Abschlussbericht der Projektgruppe 616

Johannes Bahne, Nico Bertram, Marvin Böcker, Jonas Bode,
Hermann Foot, Florian Grieskamp, Marvin Löbel, Oliver Magiera,
Rosa Pink, David Piper, Christopher Poeplau

24. April 2019

Abstract

Das Suffixarray ist eine Datenstruktur, welche zur Text-indizierung benutzt wird; es enthält die Startpositionen aller Suffixe in lexikografischer Ordnung. Die effiziente Berechnung des Suffix-Array ist nicht trivial, weshalb sich im Laufe der Jahre über ein Dutzend Algorithmen entwickelt haben, welche das Suffix-Array mit unterschiedlichen Ansätzen berechnen. Da einige der bekannten Algorithmen für die Suffix-Array-Konstruktion bereits sehr alt[1] sind, ist nicht für jeden von ihnen eine moderne Implementierung in C++ verfügbar. Im Falle des Algorithmus von Nong und Zhang[43] oder des Algorithmus von Goto[24] ist sogar noch gar keine Referenzimplementierung vorhanden. Doch auch wenn eine Implementierung vorhanden ist, kann es sein, dass diese hinsichtlich Laufzeit und Speichereffizienz nicht optimal ist. Daher versuchen wir in unserer Projektgruppe für zwölf ausgewählte Algorithmen moderne und effiziente Implementierungen zu programmieren, welche dann ohne (systembedingten) Bias verglichen werden können. Dadurch werden diese zwölf Algorithmen besser messbar, durch ein einheitliches Framework, einfach vergleichbar und dadurch insgesamt bewertbar.

[1] Der älteste hier implementierte SACA, Prefix Doubling, stammt aus dem Jahr 1990[35]

Kapitel 1

Extended Abstract

Abstract. *Efficient construction of the* suffix array *is a still ongoing research area. In this paper we introduce SACABench, a suffix array construction algorithm benchmark system for comparing the runtime and memory consumption of* suffix array construction algorithms *(SACAs). Along with this framework we include the reference implementations for many SACAs, parallel and sequential, as well as our own implementations.*

Although they are slower than their reference implementations in most cases, they can be helpful to understand the algorithms because they are written in modern C++14. In our evaluation we compare the performance of these algorithms in single-threaded and multi-threaded environments.

1.1 Introduction

The *suffix array* (SA) is a widely known text index, which can be used for various string operations, like full-text search [34] and construction of the Burrows-Wheeler Transform (BWT) [12]. It is a permutation of all indices $1 \dots n$ of a text of length n such that the ith suffix of the text T has lexicographic rank $\mathsf{SA}[i]$. While the efficient construction is still an active research area, divsufsort [23, 39] is the empirically fastest suffix array construction algorithm (SACA) since 2008, even though it has a theoretical complexity of $\mathcal{O}(n \log n)$, while there are several $\mathcal{O}(n)$ algorithms (for example SAIS[44] and DC3 [29]) available.

1.1.1 Related Work

The development of our benchmark tool as well as implementation of the SACAs is mostly independent of other work, except for the published papers on the SACAs [7, 18, 23, 29, 33, 37, 38, 42, 43, 44, 49]. For comparison of our results it is however advisable to refer to Puglisi et al. 'A Taxonomy of Suffix Array Construction Algorithms' [46].

1.1.2 Our Contribution

For the last year we (re-)implemented eleven SACAs in modern C++ for increased readability. In the process we created a sophisticated benchmark framework for SACAs, *SACABench* [5], as well as implemented parallel variants of the various SACAs.

With SACABench it is easy to implement new SACAs, because we include many of the required building blocks, for example computation of L-/S-types and radixsort. The components are also commented and documented which makes them easier to understand.

In this paper we introduce our framework and highlight its features, as well as document some of the optimiziation strategies we used to implement the SACAs. SACABench enables you to easily plot runtime and memory consumption graphs for your own SACAs as well as the included SACAs. It is therefore simple to create graphs to compare new SACAs to a wide range of known SACAs.

In the end we give an overview and performance comparison of the popular SACAs, as well as the current landshape of parallel SACAs. As far as we know, this is the first unbiased comparison of all of the included SACAs.

1.2 SACA Overview

The most naïve SACA uses a general purpose sorting algorithm to sort the suffixes of the input text. Since a string comparison is $\mathcal{O}(n)$ in the worst-case, this would give $\mathcal{O}(n^2 \log n)$ runtime. Even though this is a sub-optimal time bound, there are some algorithms[1] which don't improve on this time bound but rather use methods to speed up the real-world performance. On the other end of the spectrum there are also SACAs with an $\mathcal{O}(n)$ time bound[2]. Most SACAs share some common principles, which can be classified into the categories *inducing*, *prefix doubling* and *recursion*.

[1] Deep-Shallow [38] and mSufSort [37]

[2] SAIS [44], DC3 [29], SACA-K [42], gSACA [7] as well as nzSufSort [43]

Inducing. There are different methods of inducing but they all rely on deducing the order of suffixes from the order of other suffixes. This is possible because the ordering of suffixes that start with a common prefix is determined by the ordering of the characters that come after this prefix.

In general, this means that the ordering of two suffixes $\mathsf{T}[i..n]$ and $\mathsf{T}[j..n]$ can be used to induce the order of all suffixes that share a common prefix α, like $\alpha\mathsf{T}[i..n]$ and $\alpha\mathsf{T}[j..n]$.

Let's look at an example. Say you have the two suffixes ababc and abc. Since they both start with ab, you can induce their ordering by looking at the suffixes abc and c. Because the abc is lexicographically smaller than c, ababc is also smaller than abc.

Most algorithms that use inducing are either based on the copy technique [50] or assign types to suffixes in order to induce them later.

Prefix Doubling. Another fundamentally different approach to suffix sorting aims to double the length of sorted suffixes in every iteration. This is done by first sorting the suffixes by their first character only. To then deduce the ordering of all the elements in the b_α bucket, that is all suffixes that start with α, one can look at the relative ordering of the suffixes that start one position after the to-be-sorted suffix: their relative ordering decides the ordering of the original suffixes. In the next iteration one can look at the suffixes that start two positions later in the text, and so on. After $\mathcal{O}(\log n)$ iterations (since we double the prefix size in every iteration) all the suffixes are sorted.

Recursive. Some algorithms such as DC3 [29] and SAIS [44] require to sort a partial set of suffixes as part of their task to sort the entire suffix array. As they are able to do this by calling themselves with a different input, they are called *recursive algorithms*. This is done to achieve SACAs with liner time complexity. DC3, the first $\mathcal{O}(n)$-time SACA sorts exactly $^2/_3$ of the suffixes by recursion, while SAIS sorts all the LMS suffixes (at most $n/2$) by recursion.

See Figure 1.1 for a short summary of SACA history. The different SACAs are divided into the three classes described above. We now briefly explain the mentioned algorithms and their differences as well as in runtime and space requirements.

Prefix Doubling [18]is a SACA which works purely by the above described method *prefix doubling*. In every iteration the SA is refined and double the amount of characters are considered for sorting. Its runtime bound is $\mathcal{O}(n(\log n)^2)$. This directly influenced qSufSort[33], which improved its performance to a runtime bound by $\mathcal{O}(n \log n)$ by using the ISA (inverse suffix array). BPR[49] incorporated some ideas of inducing with prefix doubling and is therefore on the edge of those two paradigms. Its inducing is based on the copy-technique by Seward [50] and the runtime bound is at $\mathcal{O}(n^2)$. This technique is also used by Deep-Shallow [38], which uses string-sorting instead of prefix doubling. It influenced DivSufSort [23], which sorts substrings before inducing all RMS suffixes.

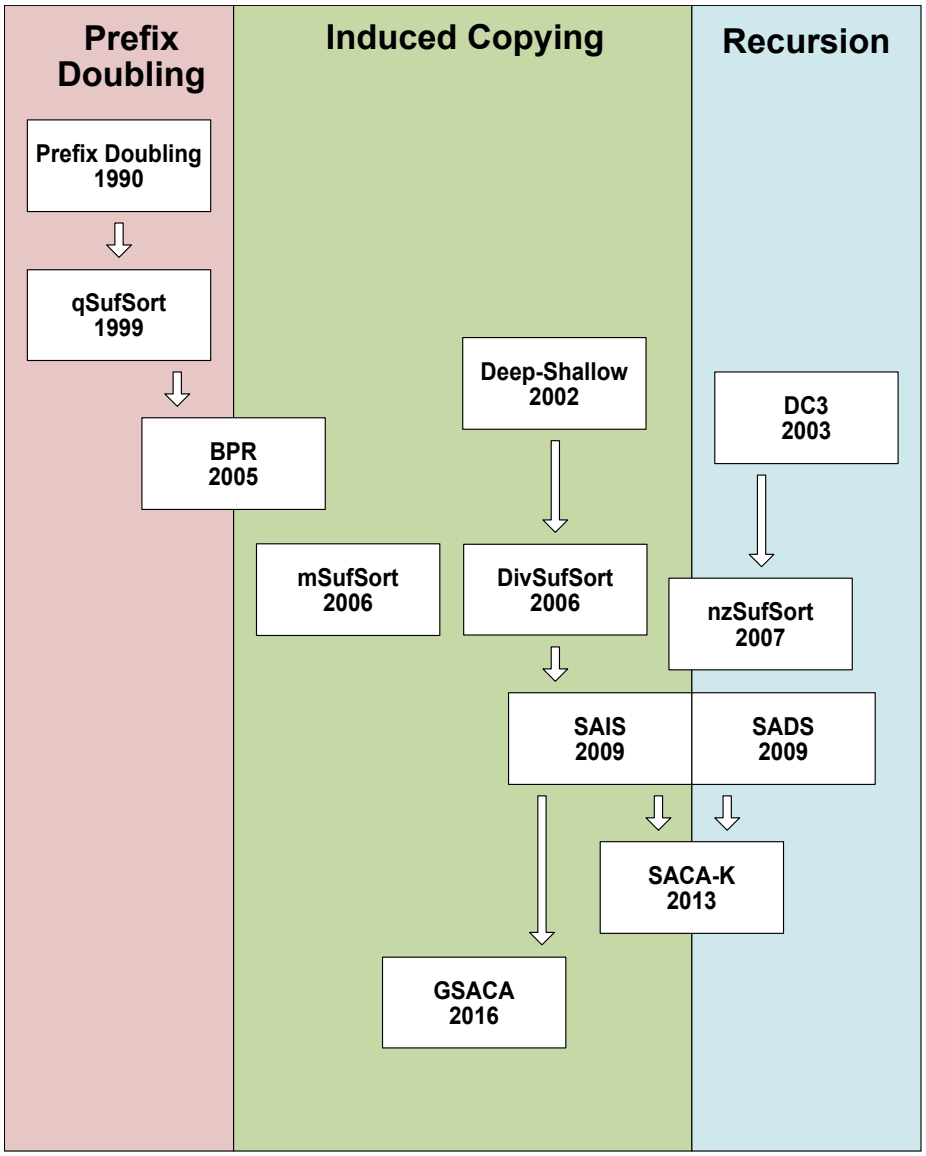


Figure 1.1: (partial) History of SACAs

All remaining suffixes can then be induced in two linear passes. It is considered the fastest algorithm so far although its runtime complexity is $\mathcal{O}(n \log n)$ and only needs constant extra space. SAIS [44] works by sorting all LMS (leftmost S-Type suffixes) in a recursion step before inducing the ordering of all suffixes in two passes similar to DivSufSort. It has linear runtime and needs $n + o(n)$ Byte extra memory. This is also done by SACA-K [43] and GSACA [7], which

are both similar to SAIS, but add some optimizations (SACA-K for example doesn't use any extra space). mSufSort [37] is another inducing-SACA, but since it constructs the ISA instead of the SA, it's not derived from any other SACA. The first linear-time SACA was DC3 [29], which utilizes the *Difference Cover* concept to sort $^2/_3$ of the suffixes in a recursion step and then induce the remaining suffixes. It can require up to $24n$ Byte extra space at maximum recursion depth. NzSufSort [42] sorts the S-Type suffixes by using the *Difference Cover* concept before inducing the L-Type suffixes similar to SAIS in one pass. It has linear runtime and does not require any extra space. For more details on the history of SACAs, see the work of Puglisi et al [46].

1.3 Benchmark Tool

SACABench is a C++14 project which contains 58[3] different SACAs. There are both sequential and parallel algorithms included and we also include the reference implementations for all of the algorithms, if one exists. You are able to run any subset of the algorithms, depending on your needs. Time and memory consumption is automatically measured and can be output as JSON. We also include tools to convert the JSON-format to graphs for a visual comparison of the algorithms.

This is the most important aspect of our framework: the possibility to easily run many SACAs on the same input text on the same hardware in order to measure and compare them fairly.

It is also possible to include a new or another existing SACA with SACABench[4].

1.3.1 Running a single algorithm

You can evaluate a single algorithm with the terminal command `sacabench construct` followed by the desired SACA and a path to an input file. To get a list of all available algorithms, you can execute the command `sacabench list`. This will output the names for all available SACAs.

While the algorithm is running, its memory consumption is measured via the tudostat library [54]. A JSON file containing detailed information about the run of the selected algorithm separated into SACA-specific phases can be genereated by the tool. This file can be converted to a plot on the tudostat website [54] (see Figure 1.2 as an example). A plot can also be generated automatically after the SACA executed by using an R-script. There is an alternative version of plots, which can be generated using LaTeX.

Multiple flags and options allow to customize the way, the SACA is executed. For example, it is possible to use only a prefix of the given input string. This enables you to use one big input file for tests with a variety of input sizes. Also it is possible to execute the selected SACA multiple times on the same input

[3] These include different variants of SACAs as well as the reference implementations.

[4] For instructions on how to do this, please consider our README in the GitHub Repository [5]

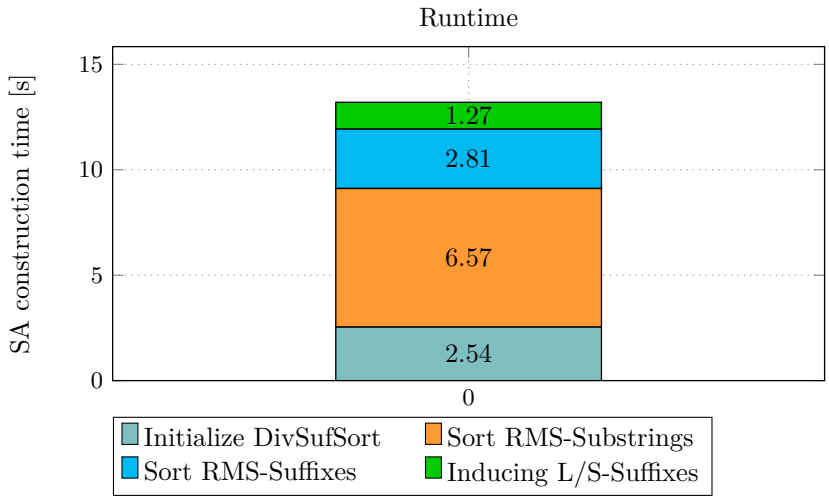


Figure 1.2: Example of runtime and memory consumption of the phases of DivSufSort, measured and visualized by SACABench.

and to combine the results. These and many more options are listed by the tool together with an explanation by adding `-h` to any subcommand.

To reduce the number of options, you can define them in a config file in INI format. An example of such a configuration can be seen in 1.3.

```
1 check = true
2 benchmark = /destination/path/to/result.json
3 force = true
4 prefix = 1K
5 repetitions = 2
6 rplot = true
7 latexplot = true
```

Figure 1.3: Example for a config file for command `sacabench construct`.

1.3.2 Comparing multiple algorithms

In order to compare different SACAs with each other, you can run a set of algorithms on a given input file. This is achieved by using the `sacabench batch` command. Most of the options available for the command `sacabench construct` are also valid for this command. By default all included algorithms are run, but you can either deselect certain algorithm by blacklisting them or run only certain algorithms by whitelisting them. These two options can also be added to the configuration file, as seen in the previous section. The selected algorithms are run sequentially on the input text and their memory consumption and construction times are measured. We also supply tools to convert the resulting JSON file into several types of plots, like bar plots, strong- and weak-scaling plots. These plots differ from the ones created by `sacabench construct`. They put the focus on comparing multiple algorithm against each other, instead of providing detailed information about the phases of the executed SACAs. These plots can be used to compare the algorithms fairly.

1.3.3 Adding additional SACAs

A standardized interface for SACAs allows to easily add new implementations. Registering the SACA within our framework allows SACABench to access the provided implementation, giving access to all features of our benchmark tool for the added algorithm. See the *README* in our GitHub repository [5] for the detailed specification of our interface.

1.4 Optimization Strategies

As with most programs, much of the performance of SACAs is dependent on efficiently implementing these algorithms. We therefore used some practical optimizations to the descriptions of the algorithms to improve performance. The following is an incomplete list of tricks one can use to do so.

1.4.1 Reducing memory consumption

We implemented our SACAs with an exchangable suffix array index type, that is a different bit length for the indices in the suffix array. With our current implementation it is possible to use 32, 40, 48 and 64 bit for the suffix array elements. Since our tool supports a different output encoding (32 or 64 bit), we can save memory during construction regardless of the desired output length.

Another method to reduce the memory footprint is to use a bitvector instead of a boolean array. This reduces the size of an array by a factor of 8 and is particularly useful when storing a boolean for every textindex, like SAIS does (L/S-type).

1.4.2 Cache-efficiency

The most crucial part of optimizing a SACA is cache-efficiency, that is aiming for time-local and space-local memory access. Since the SA is a pseudo-random mutation of numbers, it can't be written cache-efficiently. However, when implementing SACAs, avoiding unnecessary cache misses can significantly improve performance.

1.4.3 Wordpacking

To maximize throughput, multiple characters of the input (8 bit each) can be processed as a whole by interpreting them as integer numbers (64 bit, or even up to 512 bit [5]). The algorithms using wordpacking techniques included with SACABench are the GPU prefix doubler [45], Doubling [18], Discarding [18] and qSufSort [33].

1.4.4 Sorting Algorithms

Many SACAs use sorting algorithms at some point. For the benchmark a variety of sorting algorithms were implemented so that all SACAs can use them. We include some versions of quicksort, bucketsort and radixsort. This is especially interesting for naive parallelization, where one may just switch from a sequential sorting algorithm to a parallel one and thereby improve performance. Besides our own implemented sorting algorithms some external sorting algorithms are also included. Most of those external sorting algorithms are made for parallel use.

[5] This can be achieved by using CPU registers originally intended for SIMD-style vector instructions.

Binary vs. ternary comparison-based sort

Multiple versions of quicksort are implemented: Two of them are introsort [40] and ternary quicksort [8]. It has been shown by Bentley et al. [8] that the binary sorting procedure is faster if there are no equal elements in the set to be sorted [38]; otherwise, the ternary version is faster. We therefore chose the best option for the required use-case.

1.4.5 Utilization of a GPU

When implementing parallel SACAs, we also used graphics cards to improve performance. Since modern GPUs feature higher degrees of parallelism compared to CPUs due to their architecture, we include a prefix doubling SACA which uses CUDA and the CUB-Framework to run on the GPU [45]. We show that this algorithm outperformes the parallel DivSufSort implementation, although the required and available memory on current GPUs still restricts the algorithm to far smaller input sizes than CPU SACAs are capable of. Further research of GPU SACAs should be considered, as we could hardly find any (working) reference algorithms to compare our implementation with, and development of GPUs shows higher performance improvements than CPU development in certain workloads.

1.5 Evaluation

Evaluation of both the sequential SACAs, including the reference algorithms, as well as of the parallel SACAs, is done on LiDO3 [20]. We evaluate the performance of the chosen SACAs on an identical test system. LiDO3 provides us with equal compute nodes for an unbiased comparison. These nodes are all equipped with two Intel®Xeon®E5-2640 v4 with 10 cores at 2.4 GHz each, as well as 64 GB of RAM. This enables us to directly compare these algorithms in terms of runtime performance and memory consumption. We set an arbitrary limit of 2 hours of runtime as well as 60 GiB of used system memory in order to reserve some RAM for the operating system. The GPU SACAs are run on identical system with an NVIDIA®Tesla K40. Our experiment texts are `wiki.txt`, `dna.txt` and `commoncrawl.txt`:

`wiki.txt` An XML-dump of the recent wikipedia database ($|\Sigma| = 213$).

`dna.txt` A version of the 1000 Genome Project [14], which has been stripped from any characters but ACGT ($|\Sigma| = 4$).

`commoncrawl.txt` A subset of the Commoncrawl dataset [53], which has been cleaned up to contain only ASCII characters ($|\Sigma| = 242$).

1.5.1 Evaluation (sequential)

In the Figures 1.4 to 1.6 we show the performance of the implementations of different SACAs on a 1600 MiB prefix of the three listed texts.

The **runtime measurements** show that the clear winners in all three cases is DivSufSort, followed by BPR, SAIS-Lite [6] and Deep-Shallow. The algorithms MSufSort, SACA-K, qSufSort, SAIS and GSACA share the mid field. The worst measured algorithm in all three cases is the reference implementation of DC3, followed by by SADS (a variant of SAIS [44]).

The **memory consumption messurements** show that in this case the best algorithms are SACA-K and also DivSufSort, which require no extra memory, followed by Deep-Shallow, which uses only very little extra memory. Noteable is the low memory consumption of SAIS and SADS in contrast to their moderate runtime performance. As already seen in the runtime comparisons, the DC3 implementation takes the last place in terms of memory consumption, followed by BPR. However, caused by source code incompatibilities, memory utilization for BPR is not intuitively comparable to other algorithms as we are only able measure absolute memory but not extra memory for this specific algorithm[7].

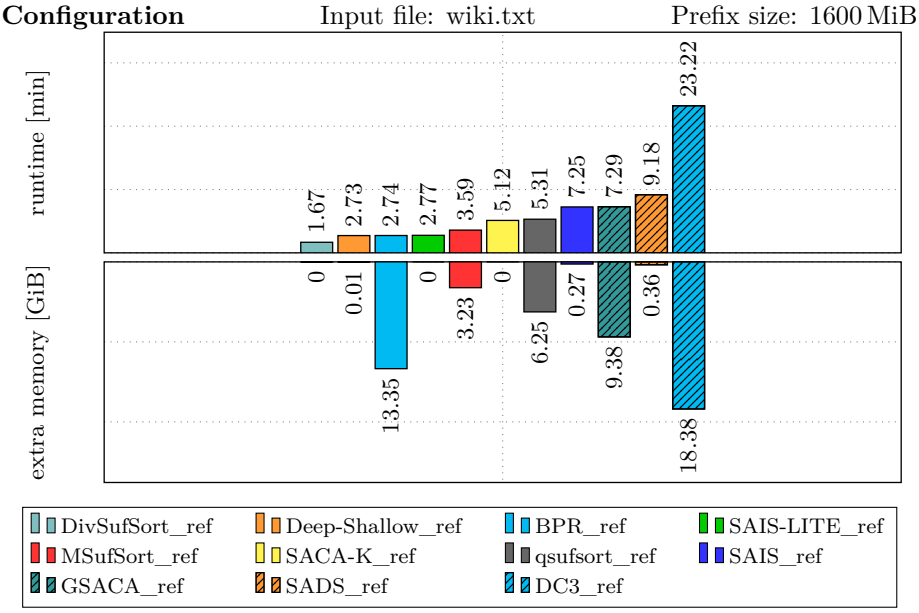


Figure 1.4: Comparison of the in SACABench included reference implementations of SACAs, including time and memory consumption: wiki.txt

[6] DivSufSort and SAIS-Lite both are optimized implementations by Yuta Mori [23].
[7] It uses an auxiliary array the same size as the suffix array.

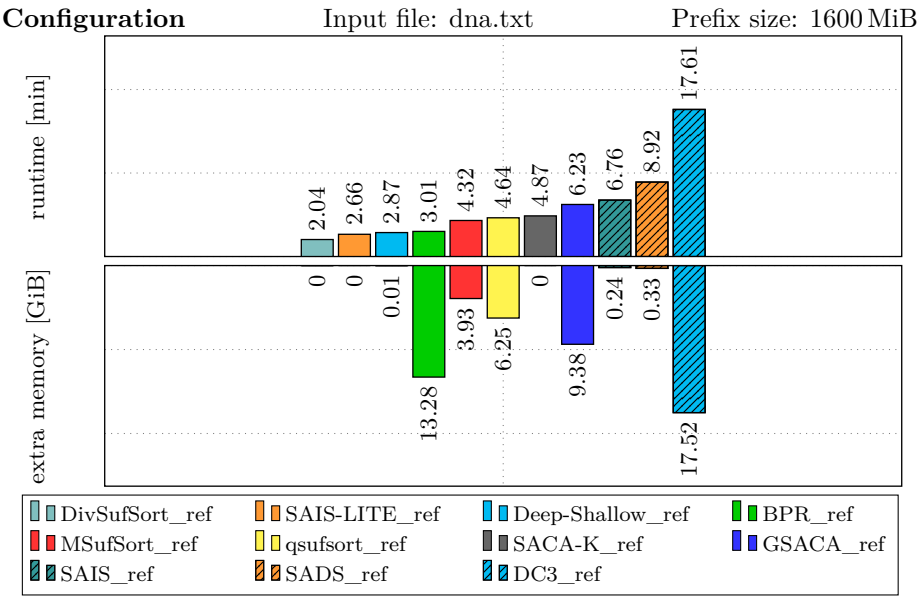


Figure 1.5: Comparison of the in SACABench included reference implementations of SACAs, including time and memory consumption: dna.txt

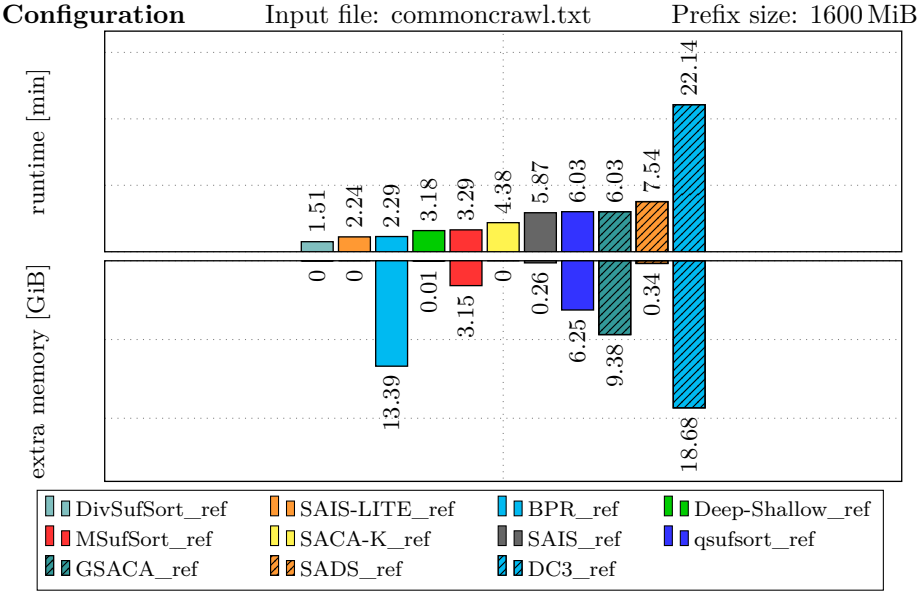


Figure 1.6: Comparison of the in SACABench included reference implementations of SACAs, including time and memory consumption: commoncrawl.txt

1.5.2 Evaluation (parallel)

In this section we will compare runtime and memory performance of the included parallel SACAs. These are variants of the sequential algorithms, that were mostly naively parallelized, with the exception of parallel SAIS[8]and DivSufSort_PARALLEL_ref which is implemented by Shun and Labeit [30, 31]. Weak-scaling experiments are conducted by scaling the size of the input text and the number of available compute cores proportionally. For each additional core, we increase the input text prefix size by 200 MB. The graph of an ideally scaling algorithm would be parallel to the x-axis.

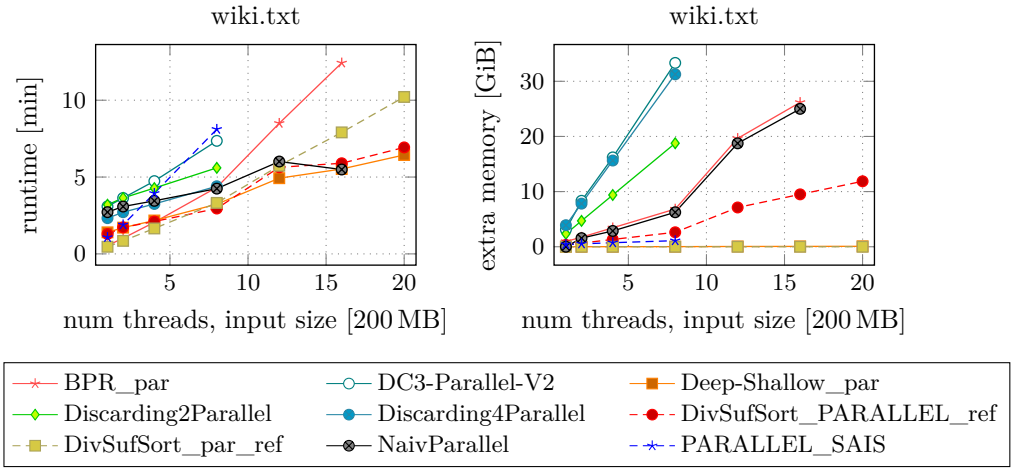


Figure 1.7: Comparison of the in SACABench included parallel implementations of SACAs, including time and memory consumption: wiki.txt

The **runtime measurements**show that, contrary to the results of the sequential algorithms, the performance of the parallel SACAs differ widely between the input text. For example, Deep-Shallow_par is the best algorithm on `wiki.txt`, as it outperformes the other algorithms in terms of runtime when more than eight cores are used, below that Mori's DivSufSort dominates all other SACAs. On `dna.txt` however, Deep-Shallow is surpassed by even the naive-parallel SACA, and fails to compute the correct SA on `commoncrawl.txt`. On these two, the parallel implementations of divsufsort are the fastest algorithms. Our implementation of parallel SAIS, both Discarding variants, as well as parallel DC3 all have problems when using more than eight cores, which is why their lines stop at 1600 MiB in all three plots. Our pSAIS can't compute the SA because of a runtime error and the other three algorithms hit the memory limit of 60 GB. The orange line for Deep-Shallow in figure 1.9 stops at 800 MiB, because it produces Segfaults from 1600 MiB on. Sadly we couldn't fix this problem in time.

[8]This algorithm uses read/write buffers and a parallel pipeline based on work of Lao et al. [32]

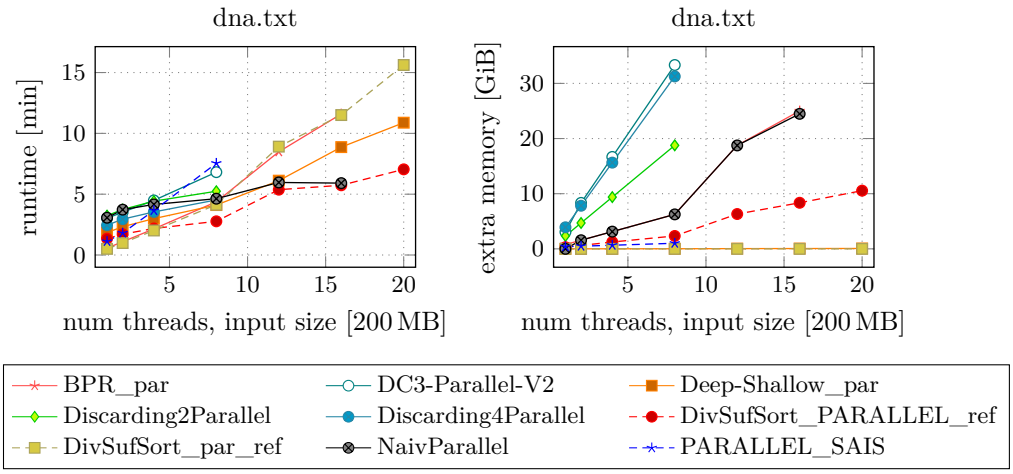


Figure 1.8: Comparison of the in SACABench included parallel implementations of SACAs, including time and memory consumption: dna.txt

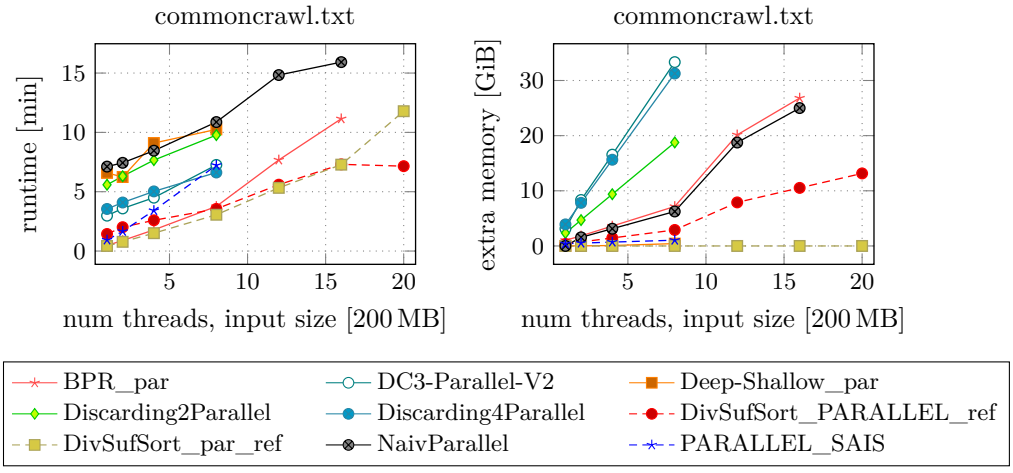


Figure 1.9: Comparison of the in SACABench included parallel implementations of SACAs, including time and memory consumption: commoncrawl.txt

The **memory consumption measurements**show that our parallel Deep-Shallow implementation outperforms most other algorithms, since it uses no extra memory, except for Mori's DivSufSort which uses no to little extra space. Both Discarding variants and DC3 have high memory consumption on all the texts, which causes them to hit the memory limit from 3200 MiB onward. This is caused by the jump in memory consumption when moving from 32-bit suffix indices to 64-bit suffix indices, which can be seen when looking at BPR_par and NaivParallel. Shun and Labeit's DivSufSort has generally higher memory

consumption than Mori's DivSufSort but trades this for better runtime performance when used with higher core-count computers. This crossing point varies between 4 (wiki.txt) and 16 cores (commoncrawl.txt).

1.5.3 Evaluation (GPU)

Configuration Prefix size: 200 MiB
Model name: Intel® Xeon® CPU E5-2640 v4 @ 2.40GHz
NVIDIA® Tesla K40 @ 2880 CUDA® cores

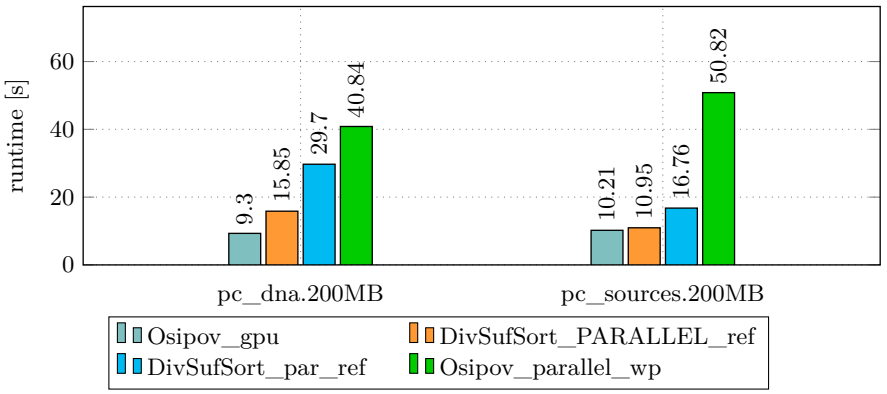


Figure 1.10: Comparison of our parallel implementations of the Osipov algorithm for GPU and CPU and DivSufSort by Mori and improvements by Shun et al. on pc_dna.200MB and pc_sources.200MB

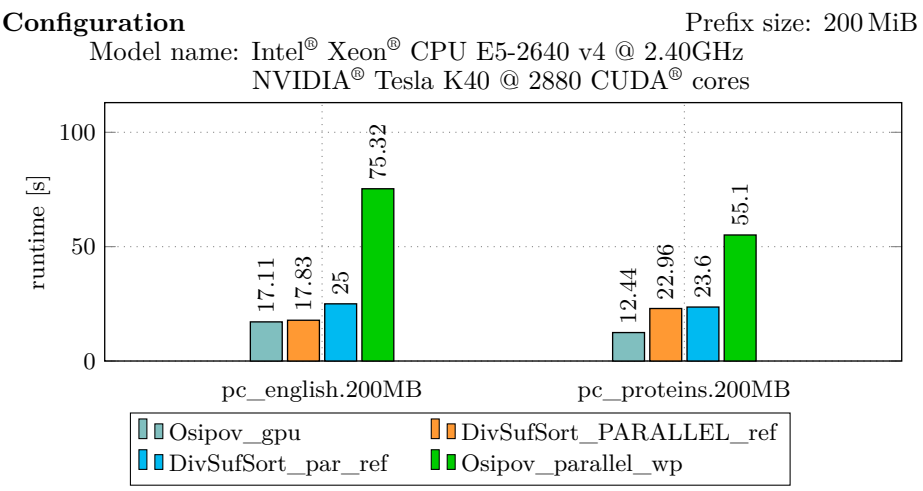


Figure 1.11: Comparison of our parallel implementations of the Osipov algorithm for GPU and CPU and DivSufSort by Mori and improvements by Shun et al. on pc_english.200MB and pc_proteins.200MB

Figures 1.10 and 1.11 show the runtime comparisons between the parallel version of Mori's DivSufSort, the parallelization of Shun et al. and the variants of the Osipov prefix doubler implemented by us, both the parallel variant for the CPU and for the GPU. All CPU implementations use the 20 available cores.

Like already seen in the last chapter, the improved Shun et al. implementation archieves better results than the one by Mori. It can also be seen that both DivSufSort implementations are in part clearly superior to the other CPU implementation in terms of runtime on all four input texts. The situation is different in comparison to the GPU implementation: on all four input texts it achieves better runtimes than all three CPU algorithms. On average this takes 12.26 seconds in our benchmarks, Shun et al.'s improved DivSufSort 16.9, Mori's original version 23.77 seconds and the CPU counterpart of the prefix doubler even 55.52 seconds. In addition, it can be observed that the GPU-Osipov with 3.02 seconds, Shun et al.'s DivSufSort with 4.3 seconds and the parallel DivSufSort with 4.63 seconds all have small standard deviations and therefore do not react disproportionately differently to different input texts. The CPU version of the Osipov algorithm, which has a standard deviation of 12.55 seconds for the four texts of the same size and thus comparatively large fluctuations in the runtime, has a different effect.

However, the high memory consumption of the Osipov algorithm is not evident from these graphs. The memory consumption of the Osipov implementations adds up to $28n$ ($20n$ for our implementation, approx. $8n$ for the radix location) and is thus significantly higher than that of the original DivSufSort, which requires additional memory depending on the alphabet size $|\Sigma|$.

1.6 Conclusion

We introduced SACABench, an extensive framework for benchmarking and comparing suffix array construction algorithms. It includes a large set of publicly available SACAs as well as re-implementations of these SACAs and simplifies the evaluation of new SACAs. These re-implementations are easier to understand than the reference implementations since they are written in modern C++.

The code is well documented and can therefore be helpful in understanding the included algorithms. We also include several parallel SACAs as well as parallel reference implementations. In our evaluation we compared the performance of the SACAs on different texts of different type and size. We can confirm divsufsort's dominance on most of the tested texts.

Our parallel implementations scale well up to 20 CPU cores and the GPU-SACA (Osipov/Prefix Doubling) is faster than the parallel divsufsort. We show that shared-memory parallelization is a fruitful approach to suffix sorting and can be used to accelerate construction of medium to large size suffix arrays. Since GPUs evolve much more quickly and stronger than modern CPUs but the amount of GPU-parallel SACAs is still small, there is much room for improvement.

Inhaltsverzeichnis

Kapitel 2

Einleitung

Im Zuge der Projektgruppe 616 in den Semestern 2018/2019 beschäftigt sich *SACABench* mit Suffix-Array-Konstruktionsalgorithmen. *SACA* ist das Akronym für **S**uffix **A**rray **C**onstruction **A**lgorithm und *Bench* die Abkürzung für Benchmark, also dem Messen von Laufzeit und Speicherplatz der Algorithmen.

Die Forschung an effizienten Konstruktionsalgorithmen für Suffix-Arrays hat in den letzten Jahren durch immer komplexer werdende Anwendungen enorm an Bedeutung gewonnen. Gerade in der Bioinformatik, in der man es im Bereich der Genomforschung mit Größenordnungen von Milliarden von Zeichen zu tun hat, sind effiziente Algorithmen in Bezug auf Zeit und Speicherplatz notwendig [44, Kap. 1].

Ein Suffix-Array repräsentiert die Startpositionen lexikographisch sortierter Suffixe eines Strings T. Sei T = suffix, dann ergeben sich für den String T folgende Suffixe:

i	$\mathsf{T}[i, n)$
0	suffix
1	uffix
2	ffix
3	fix
4	ix
5	x
6	$

Tabelle 2.1: Suffixe des Strings T

Das letzte Zeichen eines Strings ist dabei immer das Abschlusszeichen, das sogenannte Sentinel.

Sortiert man die Suffixe nun lexikografisch, ergibt sich das Suffix-Array:

i	$\mathsf{T}[i, n)$
6	\$
3	fix
2	ffix
4	ix
0	suffix
1	uffix
5	x

Tabelle 2.2: Sortiertes Suffix-Array des Strings T

Somit ergibt sich das Suffix-Array $\mathsf{SA} = [6, 3, 2, 4, 0, 1, 5]$. Ziel ist die effiziente Konstruktion dieses Arrays. Dabei gibt es Algorithmen, die den Fokus auf die Laufzeit setzen, andere wiederum auf die Speicheroptimierung und wieder andere versuchen, den besten Kompromiss aus beiden Welten zu finden. Die Aufgabe der Projektgruppe ist das Schaffen einer umfangreichen Library der bekanntesten SACAs. Diese sind eingebettet in ein Framework, das es ermöglicht, auf intuitive Art und Weise Algorithmen auf beliebigen Texten zu testen und die Performance miteinander zu vergleichen. Grundziel des Frameworks ist die Vereinheitlichung: Viele der Algorithmen existieren in einzelnen Repositorys und die Algorithmen werden in den meisten Fällen nicht auf vergleichbarer Basis getestet und analysiert. Diese Algorithmen gilt es zunächst zu verstehen und dann zu implementieren, sodass sie den Schnittstellen des Frameworks genügen. Es soll also ein erweiterbares Gesamtkonstrukt geschaffen werden, das bestehende SACAs sammelt und repräsentatives Vergleichen der Algorithmen ermöglicht.

2.1 Notation und Definitionen

Einige grundlegende Notationen und Definitionen werden hier kurz vorangestellt.

Definition 1 (Intervall)**.** *Für i, j mit $i \leq j$ definieren wir $[i, j] = i, i + 1, \ldots, j$ und $[i, j) = [i, j - 1]$ als Kurzschreibweise für Integer-Intervalle.*

Definition 2 (Arrays und Teil-Arrays)**.** *Ein Array A mit Länge $n = |\mathsf{A}|$ ist eine Sequenz von beliebigen Elementen, auf die per Index zugegriffen werden kann:*

$$\mathsf{A} := \mathsf{A}[0]\mathsf{A}[1]...\mathsf{A}[n - 1]$$

Betrachtet man ein Index-Intervall innerhalb der Array-Grenzen, bezeichnet man dies als Teil-Array von A:

$$\mathsf{A}[a, b) := \mathsf{A}[a]\mathsf{A}[a + 1]...\mathsf{A}[b - 1]$$

Definition 3 (Präfix und Suffix)**.** *Sei* A *ein Array mit Länge* n *und* $i, j \in [0, n)$*. Dann ist das Teil-Array* $\mathsf{A}[0, i)$ *ein Präfix von* A *und* $\mathsf{A}[j, n)$ *ein Suffix von* A*.*

Definition 4 (Alphabet und lexikographische Sortierung)**.** *Das konstante Alphabet ist definiert als* $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_{|\Sigma|-1}\} \cup \{\$\}$*. Es besteht aus Symbolen bzw. Zeichen* σ_i*, die im Eingabe-String vorkommen dürfen, und dem Sentinel-Symbol \$, welches das Ende des Strings markiert. Die Zeichen sind wie folgt lexikographisch (nach lexikographischer Sortierung) geordnet:* $\$ < \sigma_1 < \sigma_2 < ... < \sigma_{|\Sigma|-1}$*. Sie lassen sich durch Integer-Werte repräsentieren, indem wir \$ den Wert 0 zuweisen und* σ_i *den Wert* i*.* Σ^+ *bezeichnet weiter die Menge aller Strings über diesem Alphabet mit echt positiver Länge.*

Definition 5 (Input-String und Suffix-String)**.** *Ein Array der Länge* n*, bei dem die Elemente Zeichen aus* Σ *sind, wird* T *genannt und repräsentiert den Input-String. Das* i*-te Zeichen ist somit* $\mathsf{T}[i]$ *und das* i*-te Suffix (kurz: Suffix* i*) ist*

$$\mathsf{S}_i := \mathsf{T}[i, n) = \mathsf{T}[i]\mathsf{T}[i+1]...\mathsf{T}[n-1]$$

$\mathsf{T}[n]$ *ist das Terminalsymbol \$ (sowie alle* $\mathsf{T}[m]$ *mit* $m > n$*) und ist formal nicht Teil des Eingabe-Strings. Der Eingabe-String beginnt bei* $\mathsf{T}[0]$*.*

Definition 6 (Suffix-Array)**.** *Das Suffix-Array, kurz* SA*, bezeichnet ein Array, in dem in lexikographischer Reihenfolge die Suffix-Indizes (Positionen des Anfangsbuchstabens) gespeichert sind. Das bedeutet,* $\mathsf{SA}[j] = i$ *genau dann, wenn* $\mathsf{T}[i, n)$ *das* j*-te Suffix von* T *in lexikografischer Ordnung ist.*

Definition 7 (Bucket)**.** *Alle Suffixe, die mit demselben Zeichen* $\alpha \in \Sigma$ *beginnen, formen ein zusammenhängendes Intervall im Suffix-Array. Dieses Intervall wird* α*-Bucket genannt und mit* b_α *bezeichnet. Der* (α, β)*-Bucket* $\mathsf{b}_{\alpha,\beta}$ *bezeichnet das Intervall, dessen Suffixe mit denselben zwei Zeichen* $\alpha, \beta \in \Sigma$ *beginnen.*

Ähnlich dazu bezeichnet der Bucket b_ω *alle Suffixe im* SA*, die alle mit dem String* $\omega \in \Sigma^m$ *mit* $m > 0$ *anfangen.* b_ω *heißt dann auch Level-*m*-Bucket.*

Definition 8 (Textwiederholung)**.** *Eine Wiederholung in* T *ist ein Teilstring* $\mathsf{T}[i, i + rp]$ *mit* $r \geq 2, p \geq 0$ *und* $i, i + rp \in [0, n)$*, sodass* $\mathsf{T}[i, i + p) = \mathsf{T}[i + p, i + 2p) = \cdots = \mathsf{T}[i + (r-1)p, i + rp)$*.*

2.1.1 Praxisrelevante Grenzen

Wir gehen im Folgenden davon aus, dass für unsere Eingabe $|\Sigma| = 256$ gilt, da sich so jedes Zeichen durch ein Byte repräsentieren lässt. Dies erlaubt auch das direkte Verarbeiten von Texten in Standardkodierungen wie ASCII und UTF-8 [61], die durch Byte-Arrays ohne 0-Bytes repräsentiert werden.

Auch gehen wir davon aus, das die Länge n von Datenstrukturen, insbesondere der Eingabe und des Suffix-Arrays, durch die maximale Größe nativer Integer-Datentypen in Consumer-Computersystemen begrenzt ist. Es gilt somit in der Regel $n \leq 2^{32}$ oder $n \leq 2^{64}$, womit sich Indizes durch 4- bzw. 8-Byte Integer repräsentieren lassen. Wir betrachten außerdem $n = 2^{40}$ als 5-Byte Kompromiss zwischen den Beiden.

2.1.2 Codebeispiele

Wir geben alle algorithmischen Codebeispiele in an Python angelehnten Pseudocode an, soweit dies möglich ist. Somit ist der Code durch Einrückung strukturiert. Der Syntax wird dabei bei Bedarf mit mathematischer Notation und zusätzlichen Kontrollstrukturen erweitert.

Kapitel 3

Framework

Zur Durchführung von Benchmarks und zur besseren und leichteren Visualisierung von Laufzeit und Speicherverbrauch der Algorithmen haben wir ein Framework entwickelt. Dieses kann mit beliebigen Algorithmen für beliebige Eingaben Suffix-Arrays konstruieren und anschließend die Ergebnisse in der Konsole ausgeben, oder in einer strukturierten JSON-Datei speichern und daraus gegebenenfalls Grafiken erzeugen.

3.1 Command Line Interface

Das SACABench-Framework wurde mit dem Ziel entworfen, alle enthaltenen Funktionen flexibel aufrufen sowie Ein- und Ausgaben mit UNIX-Tools einfach verarbeiten zu können. Es können daher alle Komponenten über ein Command Line Interface (CLI) angesprochen werden, um Zugriff auf deren Funktionalitäten zu erlangen. Das Hauptprogramm unterteilt sich in fünf Bestandteile:

`sacabench list` gibt eine Liste aller verfügbaren Implementierungen aus, welche neben den im Rahmen der Projektgruppe implementierten Algorithmen auch die zugehörigen Referenzimplementierungen beinhaltet. Alle Funktionen dieser Komponente werden in Abschnitt 3.1.2 genauer erklärt.

`sacabench demo` führt alle enthaltenen Algorithmen auf einer Beispieleingabe aus, um deren Funktionalität zu demonstrieren. Alle Funktionen dieser Komponente werden in Abschnitt 3.1.3 genauer erklärt.

`sacabench construct` führt einen einzelnen Algorithmus wahlweise auf einer Eingabedatei oder auf der Standardeingabe aus und bietet unter anderem die Optionen, die Laufzeit der Verarbeitung zu messen oder die Korrektheit der Ausgabe zu testen. Alle Funktionen dieser Komponente werden in Abschnitt 3.1.4 genauer erklärt.

`sacabench batch` bietet ähnliche Funktionen wie `sacabench construct`, wendet aber mehrere Algorithmen auf den Eingabetext an. Dadurch kann die

Effizienz ausgewählter Verfahren verglichen werden. Alle Funktionen dieser Komponente werden in Abschnitt 3.1.5 genauer erklärt.

`sacabench plot` ermöglicht es, zu einer zuvor bereits durchgeführte Messung PDF-Dateien zu generieren. Dieses Kommando ist in Abschnitt 3.1.6 näher beschrieben.

3.1.1 sacabench

Das Hauptprogramm wird mit dem Befehl `sacabench` ausgeführt. Für die einzelnen Funktionen stehen entsprechende Subcommands zur Verfügung. Alle Komponenten beinhalten eine Hilfefunktion, die sich durch die Option `-h` bzw. `--help` aufrufen lässt. Die gleiche Hilfe wird außerdem ausgegeben, wenn das Programm mit ungültigen Parametern aufgerufen wird. Für präzisere Erläuterungen der Aufrufe und Optionen stehen außerdem Man-Pages für alle Subcommands zur Verfügung. Diese beinhalten eine Beschreibung der Befehlssyntax sowie Erläuterungen zur Verwendung der Optionen und Verweise auf ähnliche Befehle.

Ohne vorangegangene Installation sind die Man-Pages nur lokal aufrufbar: Aus dem SACABench Wurzelverzeichnis lässt sich beispielsweise die Man-Page für `sacabench batch` mit dem Befehl `man ./man/man1/sacabench-batch.1.gz` aufrufen. Für eine systemweite Installation steht das Shell-Skript `install-manpages.sh` zur Verfügung, welches mit entsprechenden Rechten ausgeführt werden muss, um die benötigten Dateien ins Verzeichnis `/usr/local/man/man1/` zu kopieren. Eine gekürzte Version der Man-Page für den Befehl `sacabench` ist in Anhang A.1 zu sehen.

3.1.2 sacabench list

Mit dem Befehl `sacabench list` können alle verfügbaren Algorithmen aufgelistet werden. Nach dem Aufruf erscheint eine Liste aller im Rahmen der Projektgruppe implementierten Algorithmen, sowie deren Referenzimplementierungen. Letztere sind in der angegebenen Abkürzung durch `_ref` markiert.

Über die Option `--no-description` kann außerdem die Ausgabe der Kurzbeschreibungen unterdrückt werden, sodass nur noch eine Liste aller Kürzel erscheint. Die dort aufgelisteten Kürzel entsprechen den Bezeichnungen der Algorithmen, über die sie vom Framework referenziert werden. Zusätzlich ermöglicht es die Option `-j` oder `--json`, die Namen der Algorithmen als ein JSON Array auszugeben.

3.1.3 sacabench demo

Mit dem Befehl `sacabench demo` können alle Algorithmen auf einem kurzen Beispieltext („`hello world`“) ausgeführt werden.

Diese Funktionalität erfüllt den Zweck, das Framework sowie die Algorithmen auf Lauffähigkeit auf dem verwendeten System zu testen. Auch hier ist eine Hilfeoption über `-h` bzw. `--help` erreichbar.

3.1.4 sacabench construct

Mit dem Befehl `sacabench construct` kann ein ausgewählter Algorithmus ausgeführt werden.

Der auszuführende Algorithmus wird dabei durch sein Kürzel bestimmt, wie es bei `sacabench list` angegeben ist. Gefolgt wird der Name des Algorithmus durch den Text, auf den er angewendet werden soll. Hierfür kann ein Pfad zu einer Textdatei oder alternativ `-` für STDIN angegeben werden.

Zusätzlich kann eine ganze Reihe von Optionen angegeben werden. Wie auch bei anderen Subcommands zeigen `-h` und `--help` die Hilfe an. Die Option `-c` bzw. `--check` überprüft, ob das erhaltene Suffix-Array korrekt ist. Ist dies nicht der Fall, wird eine Fehlermeldung angezeigt. Eine Alternative hierzu ist die Option `-q` oder `--quick`, welche auch die Korrektheit überprüft, jedoch einen parallelen Algorithmus verwendet. Wird die Option `-b` oder `--benchmark` gefolgt von einem Pfad angegeben, wird an diesem Pfad eine JSON-Datei mit den gemessenen Zeiten und Speicherverbrauch angelegt. Existiert an dem angegebenen Pfad bereits eine Datei, kann diese mit der Option `-f` bzw. `--force` überschrieben und durch die neue Messung ersetzt werden. Der Inhalt dieser Datei ist die Grundlage für die durch ein R-Skript erstellten Diagramme, welche mit `-z` oder `--rplot` bei der Ausführung des Algorithmus generiert werden. Als Alternative hierzu können durch die Option `--latexplot` auch Plots durch SqlPlotTools und Latex generiert werden. Weitere Informationen hierzu sind in Kapitel 3.2.5 zu finden.

Um bei den Messungen ein besseres Ergebnis zu erhalten, kann mit der Option `-r` bzw. `--repetitions` eine Anzahl an Durchführungen festgelegt werden. Hierdurch kann bei der Auswertung die Signifikanz der Messergebnisse durch die Ausblendung von Störeffekten erhöht werden. Weiterhin kann dem Befehl `-p` oder `--prefix` hinzugefügt werden. Hierbei kann die Anzahl an führenden Bytes angegeben werden, die von der Eingabe verarbeitet werden sollen. Um größere Werte leichter angeben zu können, sind die abkürzenden Schreibweisen K und M erlaubt, welche für Kilobyte bzw. Megabyte stehen. Dies sorgt dafür, dass von der übergebenen Textdatei nur so viele Bytes verarbeitet werden, wie durch diese Option angegeben werden.

Die Option `-B` oder `--binary` veranlasst das Framework zur Ausgabe in binärer Form. Anschließend werden die Einträge des Ergebnisses als Binärzahlen mit bis zu 8 Bytes ausgegeben. Die erste Zahl der Ausgabe gibt die genaue Anzahl der Stellen an. Ist eine feste Anzahl an Bits gewünscht, kann diese mit der Option `-F` bzw. `--fixed` angegeben werden. Alternativ kann das Framework

das Suffixarray als JSON-Array ausgeben, was zu Debugging-Zwecken besonders hilfreich ist.. Dies ist mit der Option `-J` oder `--json` möglich. Eine weitere Option, welche dem Subcommand `sacabench construct` übergeben werden kann, ist `-m` oder `--minimum_sa_bits` gefolgt von einem unsigned Integer. Dieser Parameter bestimmt die Anzahl der Bits, die für die Datenstrukturen während der Berechnung genutzt werden. Durch `-s` oder `--sysinfo` werden Information über das System, welches das Benchmarktool ausführt, in die Ergebnisdatei mit aufgenommen.

Anstatt eine Auswahl der genannten Optionen einzeln anzugeben, kann auch eine Konfigurationsdatei eingelesen werden, welche Werte für die zu benutzenden Optionen angibt. Diese Datei muss im im ini-Format vorliegen und mit der Optionen wird die Option `--config` gefolgt von dem Pfad zu der Konfigurationsdatei beim Aufruf von `sacabench construct` angegeben werden. Ein Beispiel für solch eine Konfigurations-Datei ist in Abbildung 3.1 zu sehen. Soll das Tool beispielsweise für den SACA `BPR` mit der angezeigten Konfigurationsdatei aufgerufen werden, sieht der entsprechende Befehl folgendermaßen aus:

```
sacabench/sacabench construct --config /pfad/zur/config BPR
/pfad/zum/input/text
```

Die gezeigte Konfigurations-Datei ist gleichbedeutentd zum Aufruf mit den Optionen `-c`, `-b`, `-f`, `-p 1K`, `-r 2`, `--rplot` und `--latexplot`. Sollen die Optionen einzeln übergeben werden, ist dies mit diesem Aufruf möglich:

```
sacabench/sacabench construct -c -b /ziel/pfad/zur/ergebnis.json
-f -p 1K -r 2 --rplot --latexplot BPR /pfad/zum/input/text
```

```
1 check = true
2 benchmark = /ziel/pfad/zur/ergebnis.json
3 force = true
4 prefix = 1K
5 repetitions = 2
6 rplot = true
7 latexplot = true
```

Abbildung 3.1: Beispiel für eine Konfigurations-Datei für den Befehl `sacabench construct`.

3.1.5 sacabench batch

Mit dem Befehl `sacabench batch` können mehrere Algorithmen auf der gleichen Eingabedatei ausgeführt werden. Die Funktionalität setzt hautpsächlich auf dem Interface von `sacabench construct` auf, weshalb ein Großteil der dort verwendbaren Optionen auch hier Anwendung findet. Zu beachten ist jedoch, dass die Ausgaben aller Algorithmen nach der Ausführung und einem optionalen Test auf Korrektheit verworfen werden. Alle das Ausgabeformat betreffenden Optionen von `sacabench construct` sind daher für `sacabench batch` ungültig.

Um nicht für jede Messung alle Algorithmen ausführen zu müssen, besteht die Option, wahlweise mit `--blacklist` einzelne Algorithmen von der Ausführung auszuschließen oder mit `--whitelist` nur explizit angegebene Algorithmen zu berechnen. Für beide Optionen entsprechen die Namen der anzugebenden Algorithmen denen aus `sacabench list` (Abschnitt 3.1.2). Benchmarks können wie genau wie zuvor mit `--benchmark` unter Angabe eines Dateinamens angelegt werden. Die mit `--rplot` oder `--latexplot` generierten Diagramme unterscheiden sich jedoch von den mit `sacabench construct` erstellten Diagrammen: Der Schwerpunkt liegt hier auf der Vergleichbarkeit der Algorithmen, weshalb in den Diagrammen gezielt die Laufzeit- und Speichermessungen aller ausgeführten Algorithmen miteinander verglichen werden. Eine ausführlichere Beschreibung der Diagramme sowie der zur Aufzeichnung der Benchmarks verwendeten Techniken ist im nächsten Abschnitt zu finden.

3.1.6 sacabench plot

Der Befehl `sacabench plot` erstellt nachträglich zu einer Messung die PDF-Dateien. Wurde beispielsweise vergessen, bei `sacabench construct` oder `sacabench batch` die Option zum Plotten anzugeben, können so die erhaltenen Messdaten dennoch ausgewerted werden. Hierzu werden dem Befehl die verwendete Messmethode, der Pfad zur Textdatei, die verarbeitet wurde, und der Pfad zur JSON-Datei mit den Ergebnissen übergeben.

3.2 Benchmark System

Ein Ziel der Projektgruppe SACABench ist der Vergleich aller selbst implementierten Algorithmen und importierten Referenzimplementierungen. Für die bessere Übersicht und Darstellung werden verschiedene Diagramme verwendet, die in diesem Kapitel näher beschrieben werden.

3.2.1 Voraussetzungen

`R-Version` Die Ergebnisse des Benchmark Systems werden mit Hilfe der Programmiersprache `R` ausgewertet. `R` ist eine *Open-Source-Software* für statistische Berechnungen und Grafiken. Daher muss auf dem auszuführenden Computer ein `R`-Interpreter installiert sein [48]. Das `R`-Skript verwendet unter anderem das Paket `rjson`, welches erst ab der Version `R` $\geq$ 3.1.0 unterstützt wird [16]. Aus diesem Grund wird die Version `R` $\geq$ 3.1.0 vorausgesetzt.

`Administrationsrechte` Das `R`-Skript benötigt neben dem Paket `rjson` auch das Paket `RColorBrewer` [41]. Beide Pakete werden automatisch installiert, wenn sie nicht vorher schon manuell installiert worden sind. Für diese Installation werden jedoch gegebenenfalls Administrationsrechte benötigt. Außerdem sind zum Beispiel für Computer mit Betriebssystem `macOS` Schreib- und Leserechte bestimmter Verzeichnisse notwendig. Daher wird empfohlen, die Befehle als Administrator auszuführen, indem zum Beispiel das Schlüsselwort `sudo` verwendet wird.

3.2.2 JSON

Wird die Option `-b` oder `--benchmark` für den Befehl `sacabench construct` und `sacabench batch` des SACABench-Frameworks angefügt, wird eine `JSON`-Datei erzeugt. `JSON` - Abkürzung für `JavaScript Object Notation` - ist ein Datenformat, das im Rahmen des SACABench-Frameworks als Speicherformat aller benötigten Benchmark-Werte verwendet wird. In dieser Datei werden unter anderem der Name, der maximale Speicherverbrauch und die Laufzeit des Algorithmus, die Größe des zu untersuchenden Ausgangstextes und die Titel jeder einzelnen Phase des Algorithmus gespeichert. Diese Datei wird anschließend mit einem `R`-Skript ausgewertet, indem die Option `-z` oder `--rplot` an die jeweiligen Befehle in der Kommandozeile angehangen werden.

3.2.3 Untersuchung einzelner Algorithmen

Wird die `rplot`-Option mit dem Befehl `sacabench construct` ausgeführt, lassen sich einzelne Algorithmen näher untersuchen. Dabei wird die Laufzeit und der maximale Speicherverbrauch des Algorithmus während der Erstellung des Suffix-Arrays gemessen. Zusätzlich gibt es die Möglichkeit, Implementierungen der Algorithmen in Phasen aufzuteilen. Dies ermöglicht eine bessere Übersicht und gegebenenfalls eine schnellere Auffindung von Schwachstellen der eigenen Implementierungen. Als Ergebnis wird eine zweiseitige `PDF`-Datei erstellt, die in dem Pfad abgespeichert wird, der hinter der Option `-b` oder `--benchmark` angegeben worden ist.

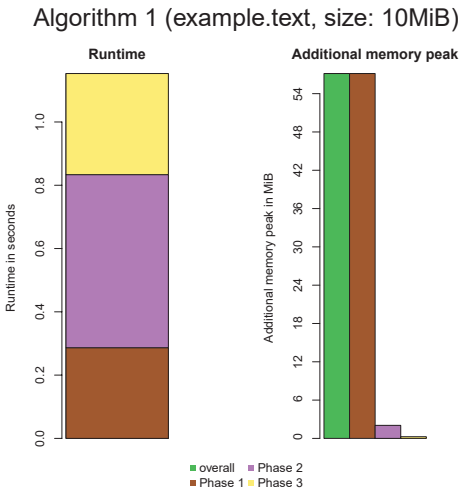


Abbildung 3.2: Messergebniss eines Beispiel-Algorithmus

Die Abbildung 3.2 zeigt die erste Seite der `PDF`-Datei. Dort ist als Überschrift sowohl der Name des Algorithmus als auch der zu untersuchende Text mit dessen Größe angegeben. Die darunter abgebildete Grafik stellt die Messergebnisse dar. Diese ist wiederum in zwei Bereiche aufgeteilt. Auf der linken Seite ist ein gestapeltes Säulendiagramm zu erkennen, das die Laufzeit repräsentiert. Jeder Stapel auf diesem Säulendiagramm entspricht der Laufzeit einer Phase. Die Zeiteinheit, die auf der Y-Achsenbeschriftung aufgetragen ist, wird dabei je nach Laufzeit angepasst, sodass die Übersicht gewahrt ist. Auf der rechten Seite befindet sich ein Säulendiagramm, das den Speicherverbrauch widerspiegelt. Auch hierbei stellt jede Säule eine Phase dar. Der Speicherverbrauch wird so gemessen, dass pro Phase jeweils der maximal gemessene Verbrauch des Speichers ohne Berücksichtigung des Ausgangstextes dargestellt ist. Welche Phase zu welchem Stapel, beziehungsweise welcher Säule gehört, ist der Legende, die sich in der Fußzeile der ersten Seite befindet, zu entnehmen.

Auf der zweiten Seite befindet sich eine Auflistung der technischen Daten des auszuführenden Systems, das sogenannte `experimental setup`. Dazu gehört der Prozessor mit der zugehörigen Frequenz, sowie die maximale Turbo-Frequenz. Ebenfalls wird die Größe des Arbeitsspeichers angezeigt. Zusätzlich wird auch auf dieser Seite erneut der Name und die Größe des Textes angegeben, zu dem das Suffix-Array erstellt worden ist.

3.2.4 Vergleich mehrerer Algorithmen

Ist jedoch nach den Messergebnissen aller vorhandenen Algorithmen in dem SACABench-Framework gefragt, wird der Befehl `sacabench batch` mit der Option `-z` oder `--rplot` verwendet. Dabei wird von jedem Algorithmus die Laufzeit und der Speicherverbrauch gemessen. Diese Messungen schaffen einen Überblick aller Algorithmen. Anders als bei den Ergebnissen einzelner Algorithmen, wird hierbei der Übersicht halber auf die Unterteilung in Phasen verzichtet. Die dabei erstellte `PDF`-Datei enthält sieben Seiten.

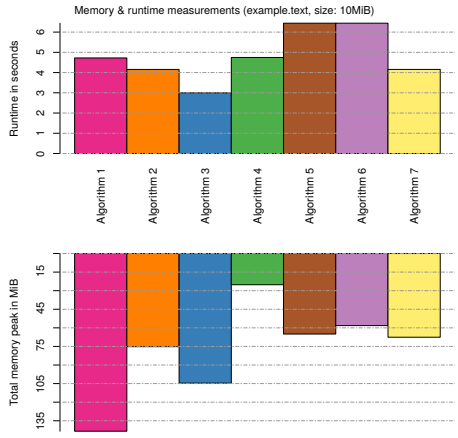


Abbildung 3.3: Messergebnisse mehrerer Algorithmen

Die Abbildung 3.3 zeigt die erste Seite der `PDF`-Datei. Auf dieser sind zwei Säulendiagramme abgebildet. Das obere Diagramm repräsentiert die jeweiligen Laufzeiten und das untere Diagramm zeigt den maximalen Speicherverbrauch inklusive des Ausgangstextes – beides jeweils in passenden Einheiten. Jede einzelne Säule eines Diagramms steht für einen Algorithmus. In der Mitte beider Diagramme sind die dazugehörigen Namen der Algorithmen wiederzufinden. Auf der ersten Seite sind diese Algorithmen aufsteigend nach Namen sortiert. Diese Reihenfolge ermöglicht einen besseren Vergleich der Messergebnisse unserer Implementierungen mit den Ergebnissen der Referenzimplementierungen, da diese bis auf das Suffix `_ref` den gleichen Namen haben und somit nebeneinander aufgelistet werden.

Auf der zweiten Seite sind die selben Messungen wiederzufinden, jedoch ist das Säulendiagramm aufsteigend nach der Laufzeit umsortiert worden. Dadurch lassen sich die Laufzeiten der Algorithmen besser miteinander vergleichen.

Auf der dritten Seite ist die Sortierung der Säulen aufsteigend nach dem maximalen Speicherverbrauch gewählt worden. Durch diese Reihenfolge lassen sich bessere Vergleiche für ähnliche Werte des Speicherverbrauchs ziehen.

Die Abbildung 3.4 zeigt die vierte Seite der `PDF`-Datei. Auf dieser ist statt eines Säulendiagramms ein Streudiagramm zum Einsatz gekommen. Hierbei handelt es sich lediglich um eine andere Darstellungsform. Auf der X-Achse befindet sich die Skala für die Laufzeit und auf der Y-Achse die Skala für den maximalen Speicherverbrauch von jedem Algorithmus für einen gegebenen Text. Rechts neben dem Diagramm ist die dazugehörige Legende wiederzufinden.

Auf der fünften Seite ist erneut ein Streudiagramm zu sehen. Jedoch ist dieses Mal lediglich die Pareto-Front eingezeichnet. In unserem Fall bedeutet das, dass ein zwei-dimensionaler Punkt dem Messergebnis eines Algorithmus entspricht, wobei eine Dimension der Laufzeit und die andere Dimension dem

maximalen Speicherverbrauch entspricht. Ein Algorithmus gehört somit zu der Pareto-Front, wenn kein anderer Algorithmus sowohl eine kürzere Laufzeit als auch einen kleineren maximalen Speicherverbrauch aufweist. In Abbildung 3.4 wären somit die Algorithmen `Algorithm 3` (blau), `Algorithm 4` (grün) und `Algorithm 7` (gelb) in der Pareto-Front. Eine solche Darstellung eignet sich, da sich somit besser abschätzen lässt, welche Algorithmen sich für welche Problemstellungen am besten eignen.

Auf der sechsten Seite befindet sich ebenfalls ein Streudiagramm. Dieses hat allerdings logarithmische Skalen, sodass die Unterschiede der Messungen einfacher zu erkennen sind. Weisen zum Beispiel viele verschiedene Algorithmen ähnliche Messergebnisse auf, so werden die Unterschiede in logarithmischen Skalen größer dargestellt und es lassen sich somit gegebenenfalls eindeutigere Aussagen treffen.

Auf der siebten und somit letzten Seite ist erneut das `experimental setup` wiederzufinden mit zum Beispiel Angaben über den verwendeten Arbeitsspeicher und Prozessor während der Ausführungen der Algorithmen.

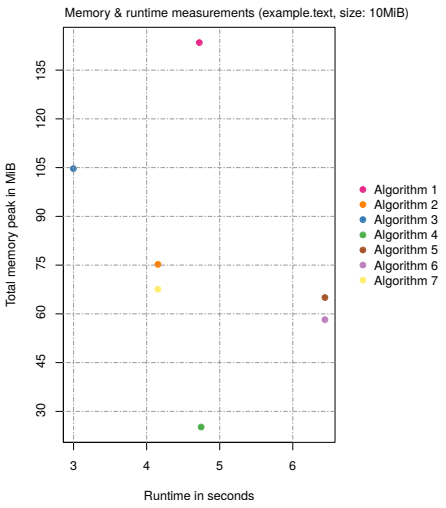


Abbildung 3.4: Messergebnisse mehrerer Algorithmen

3.2.5 Automatische PDF-Generierung mit SqlPlot-Tools

Wird `sacabench construct` zur Analyse eines einzelnen Algorithmus und dessen Phasen oder `sacabench batch` zum Vergleich mehrerer Algorithmen untereinander mit zusätzlicher Option `--latexplot` ausgeführt, werden die erhaltenen Messdaten direkt in einer PDF-Datei aufgearbeitet. Hierzu wird zunächst vom `sacabench` Haupttool eine Konfigurations-Datei mit Metainformationen über die durchgeführte Messung erstellt. Diese beinhaltet Informationen zum System, auf dem die Analyse ausgeführt wird und zu dem Text, welcher als Input für die Algorithmen verwendet wurde. Zum System werden die Anzahl der CPUs, die Anzahl an Threads pro Socket, das Modell und das verwendete Betriebssystem erfasst. Bezüglich des Textes enthält die Konfigurationsdatei den Dateinamen des Textes, Präfixgröße und Anzahl der Wiederholungen der Messung. Diese Datei wird im JSON-Format im Verzeichnis `zbmessung/sqlplot` gespeichert, welches ebenfalls ein Skript names automation.sh beinhaltet.

Im Anschluss an das Benchmarktool wird dieses Skript ausgeführt. Hierdurch wird zunächst ein temporäres Verzeichnis erzeugt, welches zur Generierung der fertigen PDF-Dateien dient. Neben der zuvor erstellten Konfigurations-Datei beinhaltet dieses Verzeichnis auch zwei LaTeX-Dateien, welche als Vorlagen für die zu erstellende PDF-Datei dienen, das für die Konvertierung des Datenformats benötigte Python-Skript `json_to_result_converter.py` und ein Makefile, welches spätere Abläufe koordiniert. Zusätzlich wird die JSON-Datei, welche bei der Messung durch das Benchmarktool erstellt wurde, in das temporäre Verzeichnis kopiert. Damit sind alle benötigten Vorbereitungen getroffen und das Makefile im Unterverzeichnis `sqlplot` wird durch das Skript ausgeführt.

Das Makefile führt das Python-Skript `json_to_result_converter.py` aus, welches als erstes die Messergebnisse unter Berücksichtigung der Metadaten in ein `RESULT`-Format konvertiert. Dieses beinhaltet die Daten in einem für *SqlPlotTools* [10] lesbaren Format. Da abhängig davon, ob `construct` oder `batch` aufgerufen wurde, unterschiedliche Daten im PDF benötigt werden, werden zwei unterschiedliche Result-Dateien erstellt: Eine Datei enthält Daten für die genauere Analyse der unterschiedlichen Phasen aller Algorithmen und die andere Datei beinhaltet die Messwerte für gesamte Algorithmen. Die Daten der einzelnen Phasen werden für die von `sacabench construct` erzeugten Diagramme benötigt, während die Messwerte der gesamten Algorithmen in den von `sacabench batch` erzeugten Diagrammen visualisiert werden. Zusätzlich generiert das Python-Skript die LaTeX-Dateien, aus denen im Anschluss die fertigen PDF-Dateien erzeugt werden. Hierzu werden die beiden LaTeX-Vorlagen im Unterordner `templates` genutzt. Nachdem all diese Dateien generiert wurden, klont das Makefile (sofern noch nicht im Verzeichnis `zbmessung/sqlplot` vorhanden) das SqlPlotTools-Repository von GitHub [10] in den temporären Ordner und baut dort das Projekt. Jede durch das Python-Skript erstellte LaTeX-Datei wird durch einen Aufruf des SqlPlotTools mit den für die Plots benötigten Daten befüllt. Dieser Aufruf wird ebenfalls von dem Makefile ausgelöst. Daraufhin können die LaTeX-Dateien gesetzt werden, wodurch die fertige PDF-Datei mit den neuen Messdaten entsteht.

Im Anschluss kopiert das Skript `automation.sh` die generierte PDF-Datei in ihr Zielverzeichnis. Zuletzt wird das temporäre Verzeichnis wieder vom System entfernt.

3.3 Code Struktur/Übersicht

Das Framework besteht hauptsächlich aus den vier Unterordnern *bigtest*, *external*, *sacabench* und *tests*.

Bigtests sind Tests auf sehr großen Texten. Die Ausführung wird mit `make bigtest` gestartet und anschließend werden verschiedene große Textdateien geladen und lokal in dem Ordner `external/datasets/` gespeichert. Die Dateien stammen von der Seite `dolomit.cs.uni-dortmund.de/` und umfassen unter anderem Quellcode, DNA-Sequenzen und Auszüge von Wikipedia. Ist der Dow-

nload abgeschlossen, werden die SACAs mit diesen heruntergeladenen Textdateien ausgeführt und das Ergebnis auf Korrektheit überprüft.

Im Ordner `external` liegen neben diesen Texten auch verschiedene Bibliotheken, welche innerhalb des Projekts eingebunden werden. Außerdem sind die Referenzimplementierungen der SACAs im Unterordner `external/reference_impls` enthalten. Diese können vom Framework mit Hilfe von Wrappern verwendet werden, welche sich im Ordner `sacabench` befinden.

Der Ordner `sacabench` ist der Hauptbestandteil des Frameworks. In diesem befindet sich der Unterordner `saca`, welcher die Implementierungen der SACAs enthält. Zusätzlich befindet sich hier der Ordner `external`, in dem die zuvor beschriebenen Wrapper zur Einbindung der externen Referenzimplementierungen vorhanden sind. Alle SACAs enthalten die Wert `EXTRA_SENTINELS`, `NAME` und `DESCRIPTION`. `EXTRA_SENTINELS` bestimmt die Anzahl der zusätzlichen Sentinels, die von dem Framework vor Aufruf des Algorithmus an den zu verarbeitenden Text angehangen werden müssen. `NAME` und `DESCRIPTION` werden bei Aufruf von `sacabench list` ausgegeben. Zusätzlich stellt jeder SACA die Methode `construct_sa` bereit, welche den Algorihtmus auf den übergebenene Text anwendet. Die Wrapper stellen diese Werte für die externen Referenzimplementierungen bereit. Neben dem Ordner `saca` befindet sich der Ordner `util`. In diesem sind verschiedenen Hilfsfunktionen und -klassen implementiert, beispielsweise finden sich hier unterschiedliche Sortieralgorithmen, die von den SACAs verwendet werden. Zuletzt befindet sich hier die Datei `sacabench.cpp`, welche das CLI implementiert. Sie enthält die `main`-Funktion, welche die eingegebenen Parameter verarbeitet und die ausgewählten Algorithmen startet.

Der Ordner `tests` umfasst Testfällen, welche mit `make check` ausgeführt werden können. Neben den Tests für die einzelnen Util-Klassen und -Funktionen stellt die Datei `saca.hpp` verschiedene Testeingaben bereit, mit denen die Korrektheit der internen und externen SACAs überprüft werden kann. Anschließend wird das Ergebnis mit einem Referenz-SACA verglichen. Die Testeingaben, auf die die SACAs angewendet werden, umfassen verschiedene Sprachen und Zeichensätze, u.a. einen All-a-Text, Smileys, Kanji und Hieroglyphen. Die Abbildung 3.5 visualisiert die beschriebenen Bestandteile der Ordnerstruktur.

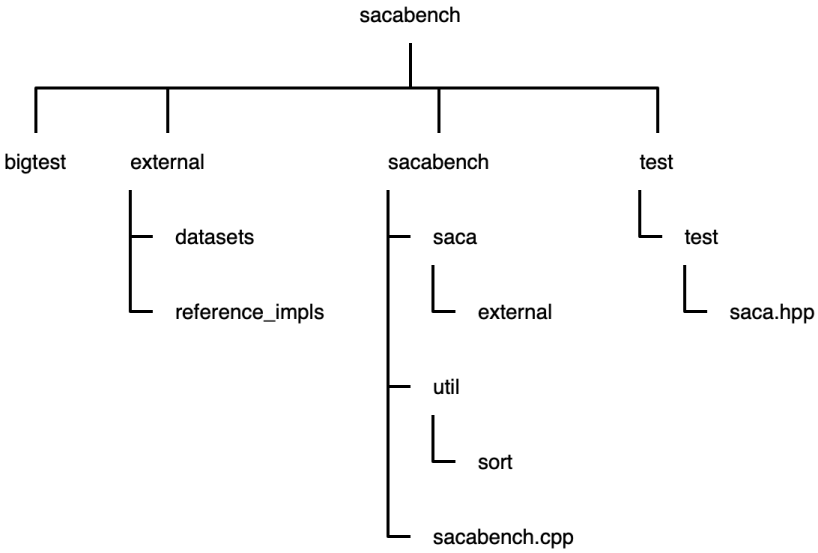


Abbildung 3.5: Ausschnitt der beschriebenen Ordnerstruktur.

Kapitel 4

Komponenten

Viele SACAs nutzen identische (oder zumindest ähnliche) Techniken oder Sortieralgorithmen. Daher ist es sinnvoll, diese Komponenten nicht mehrfach zu implementieren, sondern auf eine gemeinsame Implementierung zurückgreifen zu können, die dann von allen Algorithmen genutzt werden kann. Diese gemeinsamen Bestandteile werden hier nun beschrieben und definiert, sowie Besonderheiten bei der Implementierung hervorgehoben.

4.1 Sortieralgorithmen

4.1.1 Quicksort

Der Quicksort ist ein Sortieralgorithmus, der nach dem Teile & Herrsche Prinzip funktioniert und in der ersten Version von Hoare vorgestellt wurde [25]. Der Algorithmus erreicht zwar mit $\mathcal{O}(n^2)$ im Gegensatz zu anderen vergleichsbasierten Sortieralgorithmen eine schlechtere Worst-Case Laufzeit, aber in der Praxis ist Quicksort besser als andere Algorithmen.

Das Vorgehen ist dabei wie folgt: Sei A das zu sortierende Array. Es wird ein geeignetes Pivotelement p bestimmt und das Array durch Vertauschungen in zwei Bereiche A_{left} und $\mathsf{A}_{\text{right}}$ aufgeteilt. Dabei soll für alle i gelten, dass $\mathsf{A}_{\text{left}}[i] \leq p$ und $\mathsf{A}_{\text{right}}[i] > p$. Die beiden Teilarrays werden anschließend rekursiv sortiert. Der Rekursionsabbruch erfolgt, wenn das Eingabearray eine Größe von 0 oder 1 besitzt. In diesen Fällen ist die Eingabe bereits sortiert.

Im Folgenden stellen wir mit Ternary Quicksort eine weitere Variante des Quicksort vor. Anschließend beschreiben wir wie möglichst effizient das Pivotelement gewonnen werden kann.

Ternary Quicksort

```
def bentley_mcilroy_tq(A: Array<SuffixStartPos>):
  if |A| > 1:
    p := choose a pivot element

    # Rearrange A so, that every element in A[0, left) is
    # smaller than p, every element in A[left, right) is equal
    # to p and every element in A[right, |A|) is greater than p.
    left, right := partition(A, p)

    bentley_mcilroy_tq(A[0, left))
    bentley_mcilroy_tq(A[right, |A|))
```

Algorithmus 4.1: Bentley-McIlroy ternary Quicksort [8].

Ternary Quicksort ist eine Variante des Quicksort, die besonders geeignet ist, wenn in der Eingabe mehrere gleiche Elemente vorkommen [8]. Wir wählen hier wieder ein geeignetes Pivotelement p. Statt die Eingabe in zwei Bereiche aufzuteilen, teilen wir die Eingabe in drei Bereiche A_{left}, $\mathsf{A}_{\text{middle}}$ und $\mathsf{A}_{\text{right}}$ auf (siehe Abbildung 4.1). In A_{left} sind alle Elemente enthalten, die echt kleiner sind als p. In $\mathsf{A}_{\text{middle}}$ sind alle Elemente enthalten, die gleich p sind, und in $\mathsf{A}_{\text{right}}$ sind alle Elemente enthalten, die größer als p sind. Die Größe dieser Bereiche lässt sich durch Zählen der Elemente bestimmen. Anschließend können die Elemente durch Vertauschen in den richtigen Bereich geschrieben werden.

Nachdem wir alle Elemente in die drei Bereiche aufgeteilt haben, müssen A_{left} und $\mathsf{A}_{\text{right}}$ rekursiv sortiert werden.

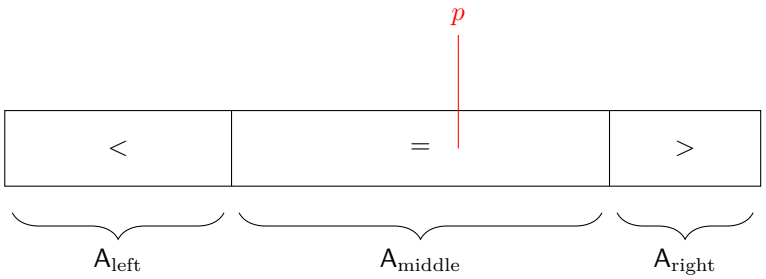


Abbildung 4.1: Partitionen bei ternary Quicksort. (Abbildung ähnlich in [8].)

Auswahl des Pivotelements

Es gibt unterschiedliche Ansätze, um in den hier beschriebenen Quicksort- Varianten das Pivotelement zu wählen. Das Problem ist der Trade-Off zwischen aufwändiger Pivot-Wahl (Pivotwahl hat hohe Laufzeit) und degeneriertem Quicksort durch eine schlechte Wahl des Pivotelements (Quicksort hat hohe Laufzeit).

Einfache Ansätze sind es das erste, mittlere oder das letzte Element der Eingabe zu wählen. Diese Ansätze sind problematisch, wenn das gewählte Pivotelement ein sehr kleines oder ein sehr großes Element ist. Dadurch wird die Eingabe nicht in zwei annähernd gleich große Bereiche aufgeteilt und die rekursiven Aufrufe werden ineffizient (degeneriertes Quicksort).

Daher verwenden wir im Quicksort und im Ternary Quicksort eine verbesserte Variante [8]. Falls die Größe des Eingabearrays kleiner als ein bestimmter Threshold ist, bestimmen wir den *Median of 3*. Das bedeutet, dass wir uns das erste, das mittlere und das letzte Element der Eingabe anschauen und den Median von diesen drei Elementen wählen. In der Implementierung haben wir als Threshold 40 gewählt. Falls die Größe des Eingabearrays größer als als der gewählte Threshold ist, bestimmen wir den *Median of 9*. Dazu teilen wir das Array in drei gleich große Teilarrays auf und bestimmen für jedes dieser Teilarrays den *Median of 3*. Unter diesen drei Elementen wird wiederum der Median gebildet, um das Pivotelement zu erhalten.

4.1.2 Introsort

Nicht immer kann eine Laufzeit von $\mathcal{O}(n \log n)$ beim Quicksort garantiert werden. Für diesen Zweck wurde der Introsort Algorithmus (auf Basis des binären Quicksort) entworfen [40]: nach einer Rekursionstiefe h durch den Quicksort Algorithmus wird Heapsort verwendet, um ein Teilarray A' von A endgültig zu sortieren. Als Wert für die maximale Tiefe h hat sich $2\lfloor \log_2 |A| \rfloor$ als empirisch geeignet herausgestellt. Eine weitere Optimierung liegt in der Verwendung von Insertionsort, wenn die Anzahl der Elemente einen Schwellwert θ unterschreitet.

Der Pseudocode in 4.2 beschreibt den Ablauf des Algorithmus [40]. Dabei befindet sich in `introsort_loop` die eigentliche Funktion: In Zeile 3 wird geprüft, ob die Größe des (Teil-)Arrays den Schwellwert unterschreitet. Ist dies der Fall, bricht die Rekursion ab. Andernfalls muss geprüft werden, ob die maximale Tiefe bereits erreicht wurde. Falls ja, wird das (Teil-)Array vollständig mit Heapsort sortiert. Tritt dies ebenfalls nicht ein, wird die verbleibende Rekursionstiefe um eins reduziert, das Pivotelement bestimmt und das Array anhand dessen partitioniert, d.h. alle größeren Elemente landen in der rechten Partition, alle Anderen in der Linken. Zuletzt wird rekursiv `introsort_loop` auf die rechte Partition ausgeführt und danach die rechte Grenze b auf die linke Partition angepasst, welche iterativ weiter sortiert wird.

Die Methode `introsort` ruft `introsort_loop` mit dem übergebenen Anfang f und Ende b sowie mit der empirisch geeigneten Maximaltiefe aufgerufen. Daraufhin wird Insertionsort auf das übergebene Array ausgeführt. Durch die vorige Vorverarbeitung über die innere Introsort-Schleife müssen nur noch In-

```
def introsort(A: Array<Integer>, f: Integer, b: Integer):
  introsort_loop(A, f, b, 2⌊log2 (b − f)⌋)
  insertionsort(A, f, b)

def introsort_loop(A: Array<Integer>, f: Integer,
                   b: Integer, depth_limit: Integer)
  while b - f > θ
    if depth_limit == 0
      heapsort(A, f, b)
      return end
    depth_limit := depth_limit - 1
    p := Choose pivot element

    # Rearrange A such that all elements ≤ p are in A[f ...i]
    # and all elements > p in A[i ...b]
    i := partition(A, f, b, p)
    # Recursively sort >-Partition
    introsort_loop(A, i, b, depth_limit)
    # Sort left partition (≤) in next iteration
    b := i
```

Abbildung 4.2: Introsort mit „≤“- und „>“-Partition.

tervalle der Länge $\leq \theta$ final sortiert werden. Dies funktioniert deshalb effizient, da z. B. Elemente im Intervall $[0, \theta)$ noch nicht endgültig sortiert sind, sie befinden sich jedoch bereits im richtigen Intervall. Somit wird kein Element aus $[\theta, 2\theta)$ oder einem hinteren Intervall in A kleiner sein als ein beliebiges Element aus dem Intervall $[0, \theta)$.

4.1.3 Multikey-Quicksort

Der Multikey-Quicksort von Bentley und Sedgewick [9] basiert auf ternary Quicksort (siehe Abschnitt 4.1.1) und sortiert Strings eines Arrays A zeichenweise. Das Funktionsargument d beschreibt, welches Zeichen momentan verglichen werden soll. Initial ist dieser Parameter $d = 0$, um das erste Zeichen von den Strings in A zu vergleichen. Wenn dieses Zeichen kleiner oder größer ist, kann der String in die „<“- bzw „>“-Partition sortiert werden.

Bei Gleichheit hingegen muss das nächste Zeichen betrachtet werden, um die Strings eindeutig zu sortieren, während für die „<“- bzw. „>“-Partitionen nach dem aktuellen Zeichen genauer sortiert wird. Abbildung 4.1 stellt die Partitionierung eines Sortierschritts bei Pivotelement p an.

Der Pseudocode 4.3 skizziert die Vorgehensweise des MK-QS Algorithmus. Es wird erst ein Pivotelement p bestimmt, welches zur Partitionierung benötigt wird. Dabei werden alle Elemente kleiner als das Pivot in $[0, i)$ einsortiert, alle gleich dem Pivotelement in $[i, j)$ und schließlich alle größeren Elemente in $[j, |\mathsf{A}|)$.

```
def mkqs(A: Array<SuffixStartPos>, d: Integer):
  if A <= 1:
    return end

  p := choose a pivot element

  # Rearrange A so, that T[A[..i]] has a character at position d
  # that is smaller than p, T[A[i..j]] has p at position d and
  # T[A[j..]] has a character at position d that is larger than p.
  i, j := partition(A, d, p)

  mkqs(A[0, i), d)
  # Sort Equal-Partition by next character.
  mkqs(A[i, j), d+1)
  mkqs(A[j, A), d)
```

Abbildung 4.3: Bentley-Sedgewick Multikey-Quicksort [9].

Zunächst wird die „<“-Partition nach dem d-ten Zeichen rekursiv weiter sortiert, bevor für die „=“-Partition zum Sortieren das $d+1$-te Zeichen verwendet wird. Zuletzt wird die Rekursion für die „>“-Partition durchgeführt.

4.1.4 Bucketsort

Der Bucketsort-Algorithmus [15, Kapitel 8.2 (dort unter dem Namen *counting sort*)] ist ein Sortierverfahren, das für Werteverteilungen über einem Alphabet einer konstanten Größe Eingaben in linearer Zeit sortieren kann. Da die Bedingung einer konstanten Alphabetgröße allerdings häufig nicht erfüllt ist, wird Bucketsort oft in Verbindung mit einer Schlüsselfunktion verwendet, um gemäß des Schlüssels „ähnliche“ Werte zu gruppieren. Diese gruppierten Bereiche werden als *Buckets* bezeichnet und im Anschluss mit einem anderen Sortierverfahren verfeinert. Bucketsort dient in dieser Variante als Mechanismus zur groben Vorsortierung.

Da im Anwendungsfeld der Suffix-Array-Konstruktion die zu sortierenden Elemente Suffixe über Σ^* sind, ist eine Schlüsselfunktion notwendig, die Wertebereich einschränkt. Hier werden als Schlüssel gemeinsame Präfixe der Länge d (für Bucketsort mit Tiefe d) verwendet.

Der Algorithmus besteht aus drei iterativ durchgeführten Schritten, die hier erläutert werden. Als begleitendes Beispiel dient die Sortierung der Suffixe des Wortes *caabaccaabacaa* mit der Tiefe 2.

Schritt 1 Es wird zunächst ein Hilfs-Array (Abbildung 4.4) angelegt, in dem zu jedem möglichen Schlüssel (hier Präfixe über Σ^2) dessen Häufigkeit in der Eingabe vermerkt wird. Dies geschieht anhand eines sequentiellen Scans der Eingabesequenz. Diese Anzahl legt die Größe des jeweiligen Buckets fest. In einem Scan des Hilfs-Arrays kann dann In-Place die kumulative Summe der Größen bestimmt werden, durch welche die Grenzen zwischen den Buckets festgelegt werden.

prefix	$$	$a	$b	$c	a$	aa	ab	ac	b$	ba	bb	bc	c$	ca	cb	cc
# in T	0	0	0	0	1	3	2	2	0	2	0	0	0	3	0	1
sum	0	0	0	0	1	4	6	8	8	10	10	10	10	13	13	14

Abbildung 4.4: Bestimmung der Bucketgrößen und Startpositionen

Schritt 2 Um die Elemente sortieren zu können, wird ein zweites Array in der Länge der Eingabesequenz angelegt. Dieses Array ist auf Basis der zuvor berechneten Grenzen (imaginär) in Buckets eingeteilt (Abbildung 4.5). In einem

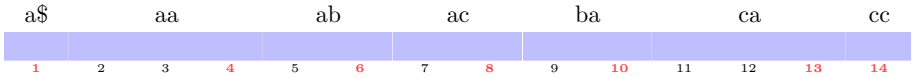


Abbildung 4.5: Buckets ohne einsortierte Suffixe

Scan der Eingabesequenz wird dann iterativ für jedes Element der zugehörige Schlüssel berechnet, um anhand der Tabelle aus Abbildung 4.4 die Position im Ausgabe-Array zu bestimmen. Nach dem Einfügen an dieser Position wird die Grenze des jeweiligen Buckets entsprechend angepasst, um bereits eingefügte Elemente im Nachhinein nicht mit neuen Elementen zu überschreiben.

Nach Abschluss dieses Schritts liegen die vorsortierten Elemente der Eingabesequenz in korrekter Reihenfolge bezüglich des gewählten Sortierschlüssels im Ausgabe-Array (Abbildung 4.6).

Schritt 3 (optional) Je nach Anwendungsfall kann es erwünscht sein, dass die Elemente in sortierter Reihenfolge direkt im Eingabe-Array liegen. Falls dies erforderlich ist, so wird in einem dritten Scan der Inhalt des Ausgabe-Arrays ins Eingabe-Array kopiert und dabei die alte Eingabe überschrieben. Die Weiterverarbeitung der einzelnen Buckets erfolgt dann mit einem anderen Sortieralgorithmus oder wahlweise auch weiter mit Bucketsort unter Verwendung einer anderen Schlüsselfunktion.

a$		aa		ab		ac		ba		ca			cc
S_{14}	S_2	S_8	S_{13}	S_3	S_9	S_5	S_{11}	S_4	S_{10}	S_1	S_7	S_{12}	S_6
1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	a	a	a	a	a	a	a	b	b	c	c	c	c
	a	a	a	b	b	c	c	a	a	a	a	a	c
	b	b		a	a	c	a	c	c	a	a	a	a
	a	a		c	c	a	a	c	a	b	b		a
	c	c		c	a	a		a	a	a	a		b
	c	a		a	a	b		a		c	c		a
	a	a		a		a		b		c	a		c
	a			b		c		a		a	a		a
	b			a		a		c		a			a
	a			c		a		a		b			
	c			a				a		a			
	a			a						c			
	a									a			
										a			

Abbildung 4.6: Buckets mit einsortierten Suffixen

4.1.5 Radixsort

In diesem Abschnitt behandeln wir das Sortierverfahren *Radixsort*. Dabei handelt es sich im Gegensatz zu den anderen Sortieralgorithmen, die in unserem Framework verwendet werden, nicht um einen vergleichsbasierten Sortieralgorithmus, sondern ist ähnlich zu Bucketsort (Abschnitt 4.1.4). Es werden stattdessen die einzelnen Stellen der zu sortierenden Elemente A in einer bestimmten Reihenfolge sortiert. Die Stellen der Elemente sind dabei Zeichen aus einem endlichen Alphabet. Da wir die Größe des Alphabets im Voraus kennen, lassen sich die Elemente in Buckets einsortieren. Wenn die maximale Anzahl der Stellen der Elemente k beträgt, erreicht dieser Algorithmus eine Worst-Case Laufzeit von $\mathcal{O}(kn)$. Falls k konstant ist, ist die Laufzeit somit linear.

Wir stellen in diesem Abschnitt verschiedene Radixsort-Varianten vor. Zum einen unterscheiden sich diese hinsichtlich der Reihenfolge, in der die Stellen der Elemente betrachtet werden. Es kann bei der Stelle mit der höchsten Wertigkeit (MSD) oder der Stelle mit der niedrigsten Wertigkeit (LSD) begonnen werden. Wenn wir die Elemente nach dem MSD sortieren, sind die Elemente nach den Zeichen an der aktuell betrachteten Stelle vorsortiert. Um Elemente mit gleichen Zeichen zu sortieren, müssen die übrigen Stellen rekursiv sortiert werden. Wenn wir die Elemente nach dem LSD sortieren, müssen die Elemente nach der aktuell betrachteten Stelle so in die Buckets einsortiert werden, dass bei gleichem Zeichen an der aktuellen Stelle die bisherige Sortierung erhalten bleibt. Beide Radixsort-Varianten wurden unter anderem auch in [15] beschrieben.

Außerdem lässt sich Radixsort mit einem zusätzlichen Hilfsarray oder inplace sortieren. Die Varianten, die ein zusätzliches Hilfsarray benötigen, sind stabile Sortierverfahren. Die inplace Varianten sind instabile Sortierverfahren.

In allen Varianten, die wir hier vorstellen, muss als Eingabeparameter eine `key_function` übergeben werden. Ist x ein Element der Eingabe und k die aktuelle Stelle, gibt `key_function`(x, k) das Zeichen an Stelle k von x zurück. Dies ist von Vorteil, wenn die Eingabearrays unterschiedliche Formen besitzen. Es kann sein, dass die Eingabe aus den tatsächlich zu sortierenden Elementen besteht, oder die Eingabe besteht wie im Abschnitt 6.8 aus Positionen von Teilstrings im Eingabetext.

Die unterschiedlichen Radixsort-Varianten verwenden alle die Hilfsfunktion *Bucketsort* (siehe Abschnitt 4.1.4). Diese erhält als Eingabe die zu sortierenden Elemente A, die aktuelle Stelle i und ein Bucketarray B mit $|\mathsf{B}| = |\Sigma|$. *Bucketsort* zählt zu jedem Zeichen des Alphabets die Häufigkeit und speichert diese in B. Anschließend wird die Präfixsumme von B gebildet. Dadurch werden in B die Startpositionen der jeweiligen Buckets berechnet.

MSD-Radixsort

Diese Variante sortiert die Elemente beginnend ab dem MSD und verwendet ein zusätzliches Hilfsarray R. Zunächst werden mit *Bucketsort* in ein Bucketarray B der Größe $|\Sigma|$ die Startpositionen der Buckets berechnet. Anschließend werden die Elemente anhand der Zeichen in die Buckets aufgeteilt. Dazu bestimmen wir mithilfe von B die Position, an der wir das Element in R schreiben müssen und inkrementieren anschließend die entsprechende Position von B. Nachdem alle Elemente in R geschrieben wurden, werden diese wieder zurück in das Eingabearray kopiert. Falls es Buckets gibt, die mehr als ein Element enthalten, werden diese Buckets rekursiv mit der nächsten Stelle weiter sortiert.

Im folgenden Beispiel werden die Elemente $\mathsf{A} = \{512, 794, 187, 394, 384\}$ mit MSD-Radixsort sortiert. Dabei sortieren wir die Elemente zunächst nach der ersten Stelle.

i	0	1	2	3	4
$\mathsf{A}[i]$	187	394	384	512	794

Da das Bucket für das Zeichen 3 mehr als ein Element enthält wird dieses rekursiv nach der zweiten Stelle sortiert.

i	0	1	2	3	4
$\mathsf{A}[i]$	187	384	394	512	794

Nun existiert kein Bucket mit mehr als einem Element. Die Eingabe wurde also sortiert. An diesem Beispiel kann man einen Vorteil des MSD-Radixsort erkennen. Man muss nicht jede Stelle betrachten, wenn die Eingabe bereits sortiert ist, und kann bereits vorher abbrechen. Dadurch ist die Laufzeit in der Praxis schneller. Jedoch muss für jeden rekursiven Aufruf das Bucketarray im Stack behalten werden. Bei großen Alphabeten kann das zu großem Speicherverbrauch führen.

LSD-Radixsort

In dieser Variante werden die Elemente beginnend ab dem LSD sortiert. Die Sortierung der Elemente anhand der aktuellen Schritte ist analog zum MSD-Radixsort. Wenn wir eine Stelle weitergehen, müssen aber alle Elemente anhand der nächsten Stelle sortiert werden.

Im folgenden Beispiel sortieren wir die Elemente $\mathsf{A} = \{512, 794, 187, 394, 384\}$. Dabei verwenden wir diesmal den LSD-Radixsort. Wir beginnen mit der Sortierung an der letzten Stelle.

i	0	1	2	3	4
$\mathsf{A}[i]$	512	794	394	384	187

Anschließend werden die Elemente nach der zweiten Stelle sortiert. Dabei bleibt die Sortierung des vorherigen Schrittes bei gleichen Zeichen erhalten.

i	0	1	2	3	4
$\mathsf{A}[i]$	512	384	187	794	394

Als Letztes sortieren wir die Elemente nach dem ersten Zeichen. Anschließend sind die Elemente fertig sortiert.

i	0	1	2	3	4
$\mathsf{A}[i]$	187	384	394	512	794

Im Gegensatz zum MSD-Radixsort reicht es hier aus nur ein Bucketarray im Speicher zu haben. Diese Radixsort-Variante ist also speichereffizienter. Da für die Sortierung jede Stelle berücksichtigt werden muss, ist diese Variante in der Praxis aber langsamer als MSD-Radixsort.

Inplace MSD-Radixsort

Der MSD-Radixsort lässt sich so modifizieren, dass kein zusätzliches Hilfsarray benötigt wird [2]. Statt die Elemente direkt in ihre richtige Buckets zu schreiben, müssen wir durch Vertauschungen die Elemente an die richtige Position bringen. Hier sei angemerkt, dass wir den LSD-Radixsort nicht auf die gleiche Art modifizieren können, da die einzelnen Stellen stabil sortiert werden müssten. Wenn wir Elemente miteinander tauschen würden, wäre diese Stabilität nicht mehr garantiert.

Wenn wir die Elemente im Eingabearray in die richtigen Buckets schreiben, lassen sich die Elemente in zwei Mengen einteilen. Es gibt Elemente, die bereits ins richtige Bucket geschrieben wurden, und Elemente, die noch nicht in das richtige Bucket geschrieben wurden. Wenn wir über die Elemente der Eingabe iterieren, müssen wir nur die Elemente betrachten, die noch nicht in das richtige Bucket geschrieben wurden. Die anderen müssen wir überspringen. Dazu benötigen wir zwei Bucketarrays $\mathsf{B}_{\text{start}}$ und B_{end}. In $\mathsf{B}_{\text{start}}$ steht für jedes Zeichen des

Alphabets die Startposition des jeweiligen Buckets und in B_{end} steht die erste freie Position des Buckets. Da zu Beginn die Startpositionen und ersten freien Positionen eines Buckets zusammenfallen, lassen sich $\mathsf{B}_{\text{start}}$ und B_{end} durch Bucketsort berechnen.

Um die Elemente in die Buckets aufzuteilen durchlaufen wir das Eingabearray von links nach rechts. Wenn wir an der Position i sind, schreiben wir $\mathsf{A}[i]$ gemäß dem Zeichen σ von $\mathsf{A}[i]$ an der aktuellen Stelle an die Position $j = \mathsf{B}_{\text{end}}[\sigma]$. Falls $i \neq j$ gilt, werden die Elemente an den Positionen i und j getauscht und für den nächsten Schritt bleibt i unverändert. Falls $i = j$ gilt, bleibt $\mathsf{A}[i]$ an der aktuellen Stelle stehen und i muss erhöht werden. Dabei müssen wir darauf achten, dass i immer auf eine Position zeigt, die noch nicht in ein Bucket einsortiert wurde. Wir müssen also i mit der Startposition s des nächsten Buckets in $\mathsf{B}_{\text{start}}$ vergleichen. Falls $i + 1 < s$, können wir i inkrementieren. Ansonsten müssen wir solange volle Buckets überspringen bis wir i auf eine freie Position setzen können.

Falls es nach der Aufteilung Buckets gibt, die mehr als ein Element enthalten, müssen diese rekursiv für die nächste Stelle sortiert werden.

Radixsort für große Alphabete

Im Folgenden nehmen wir an, dass wir Elemente sortieren, die aus maximal 3 Zeichen bestehen. Dieses Vorgehen ließe sich aber auch für längere Elemente verallgemeinern. In unserem Framework haben wir eine Radixsort-Variante entworfen, die sich besonders eignet, wenn die Größe des Alphabets durch die Größe des Eingabearrays beschränkt ist. Also für $n = |\mathsf{A}|$ gilt $|\Sigma| \leq n$. Die Inplace-Variante, die wir bereits vorgestellt haben, ist in diesem Fall nicht geeignet, da in jedem Rekursionsschritt ein neues Bucketarray erzeugt wird. Dadurch kann sich ein Speicherverbrauch von $\mathcal{O}(3n)$ ergeben.

Wir beschreiben hier, wie man die Variante aus Abschnitt 4.1.5 so modifizieren kann, dass wir genau mit einem Bucketarray B der Größe n auskommen können. Die Idee ist es das Eingabealphabet in jedem rekursiven Schritt so zu verkleinern, dass wir in allen rekursiven Schritten genug Platz in B haben.

Dazu teilen wir das Eingabealphabet in fünf Buckets der Größe $\frac{n}{5}$ auf. Jedes dieser Buckets wird nun wie in der Inplace-Variante beschrieben in die Buckets für jedes Zeichen aufgeteilt. Da wir nur noch $\frac{n}{5}$ der Zeichen sortieren müssen, benötigt die Aufteilung für $\mathsf{B}_{\text{start}}$ und B_{end} jeweils nur $\frac{n}{5}$ Speicher. Außerdem benötigen wir nach der Aufteilung nur $\mathsf{B}_{\text{start}}$ für die rekursiven Aufrufe. Das heißt es reicht aus für den ersten und zweiten rekursiven Aufruf jeweils $\frac{n}{5}$ Speicher zu verwenden und für den dritten rekursiven Aufruf benötigen wir für $\mathsf{B}_{\text{start}}$ und B_{end} insgesamt $\frac{2n}{5}$ Speicher. Insgesamt passt also für jeden rekursiven Aufruf der verwendete Speicher in B.

Optimierung

Bei den MSD-Radixsort-Varianten ist es in der Praxis meistens nicht sinnvoll kleine Buckets weiter mit Radixsort zu sortieren, da viele rekursive Aufrufe mit kleinen Bucketgrößen einen zu großen Overhead erzeugen. In diesen Fällen ist es deutlich schneller ab einer bestimmten Bucketgröße die Buckets mit einem naiven Sortieralgorithmus wie Insertionsort zu sortieren. Dazu muss in diesen Radixsort-Varianten zusätzlich eine `compare_function` übergeben werden, die als Eingabe die zu vergleichenden Elemente i und j und die aktuelle Stelle k erhält. Die `compare_function` vergleicht dann die Elemente, wobei die Zeichen, die vor der Stelle k vorkommen, abgeschnitten werden.

4.1.6 Inplace Parallel Super Scalar Samplesort (IPs^{4}o)

In SACABench verwenden wir auch andere Sortieralgorithmen, die wir nicht selbst implementiert haben. Der erste dieser Algorithmen ist IPs^{4}o [4]. Dieser existiert in einer sequentiellen und einer parallelen Variante, die mittels OpenMP parallel funktioniert.

IPs^{4}o baut auf Samplesort auf, welcher wiederrum auf Quicksort aufbaut. Statt nur ein Pivot wie bei Quicksort werden dabei aber mehrere Pivot verwendet. Dadurch erhält man einen Algorithmus „der cache-effizient ist, datenparallel arbeitet und *branch mispredictions* verhindert“ [4]. IPs^{4}o verbessert diesen Algorithmus und implementiert ihn in-place, also mit $\mathcal{O}(1)$ Extraspeicher. Da dies ein komplexes, vergleichsbasiertes Sortierverfahren ist, sei an dieser Stelle für weitere Details auf das Paper von Axtmann et al. [4] beziehungsweise das öffentliche Repository von IPs^{4}o [60] verwiesen.

4.1.7 Parallel Stable Sort

Für einige Algorithmen war es wichtig, eine stabiles Sortierverfahren zu implementieren. Wir verwenden dafür eine Implementierung von Intel® eines parallelen *Merge Sorts* [47]. Merge Sort ist ein rekursives, stabiles Sortierverfahren, welches eine theoretische Worst-Case-Komplexität von $\mathcal{O}(n \log n)$ hat. Diese Struktur macht es möglich, diesen Algorithmus effizient zu parallelisieren.

Sei A das zu sortierende Array, so werden zuerst die beiden Hälften $\mathsf{A}[0, \frac{|A|}{2})$ und $\mathsf{A}[\frac{|A|}{2}, |\mathsf{A}|)$ rekursiv sortiert. Der Rekursionsabbruch ist dabei eine zu sortierende Menge der Größe 1, welche automatisch sortiert ist. Danach kann die Sortierung der Gesamtmenge aus den sortierten Teilmengen zusammengesetzt werden, indem beide Hälften in der sortierten Reihenfolge gleichzeitig durchlaufen werden. Dabei wird immer das kleinere, aktuelle Element der beiden Partitionen verwendet, um die Gesamtsortierung zu konstruieren.

Um diesen Algorithmus zu parallelisieren, können beide Partitionen von unabhängigen Kernen sortiert werden. Der *Merge-Schritt* muss dennoch sequentiell ausgeführt werden.

4.1.8 Standardbibliothek-Sortierer

In SACABench verwenden wir außerdem die Sortierer aus der GNU Standardbibliothek von C++. Wir verwenden insbesondere die parallelen Varianten von `std::sort` und `std::stable_sort`. Diese erwiesen sich als zuverlässiger als IPs^{4}o auf vielen Kernen und werden daher für die meisten naiven Parallelisierungen der SACAs verwendet.

Im nicht-stabilen Sortierer kann entweder *Parallel Multiway Mergesort* oder *Parallel Load-Balanced Quicksort* verwendet werden. Die Mergesort-Variante teilt dabei das Problem nicht in zwei Teile sondern k Teile, wovon jeder von einem anderen Kern bearbeitet wird. Um Verlangsamungen durch kleine Partitionen zu vermeiden, teilt die Quicksort-Variante Threads, die mit ihrem Anteil des Arrays schon fertig sind, neue Teile zu, um eine höhere Effizienz zu erreichen. Da der erste Algorithmus stabil sortiert, wird er auch als stabiler Sortierer verwendet. Die Details der verwendeten Algorithmen werden im Paper von Singler und Kosnik von 2008 [51] erläutert.

4.2 Techniken

In diesem Kapitel stellen wir Techniken vor, die unabhängig vom Algorithmus genutzt werden können. Diese sollen die Berechnung des Suffix-Arrays vereinfachen und auch beschleunigen können.

4.2.1 Effektives Alphabet

Sowohl die Laufzeit als auch der Speicherverbrauch einiger der im Framework enthaltenen Algorithmen sind abhängig von der Alphabetgröße des gegebenen Strings. Da das Alphabet allerdings oft mehr Zeichen umfasst als tatsächlich im String vorkommen, kann es sinnvoll sein das vorliegende Alphabet auf die ausschließlich genutzten Zeichen zu reduzieren. Ein solches Alphabet nennen wir effektives Alphabet.

Definition 9 (Effektives Alphabet)**.** *Sei* $\mathit{eff}: \Sigma \cup \{\$\} \to \{0, \ldots, |\Sigma|\}$ *eine bijektive Funktion*[1]*, sodass*

1. $\mathit{eff}(\$) = 0$ *und damit* $\forall c \in \Sigma : \mathit{eff}(\$) < \mathit{eff}(c)$ *und*
2. $\forall c_1, c_2 \in \Sigma : c_1 < c_2 \Rightarrow \mathit{eff}(c_1) < \mathit{eff}(c_2)$.

Ziel ist es also eine Abbildung *eff* zu konstruieren, die jedes Zeichen in T auf einen Wert im Intervall $[0, |\Sigma|]$ abbildet und dabei die lexikographische Ordnung der Zeichen beibehält, wobei $|\Sigma|$ der tatsächlichen Anzahl vorkommender, unterschiedlicher Zeichen in T entspricht und $\mathit{eff}(\$) = 0$.

[1] An dieser Stelle weicht die formale Definition bewusst von der in der zugehörigen Publikation [49] ab, um Definitionslücken für erweiterte Strings zu vermeiden.

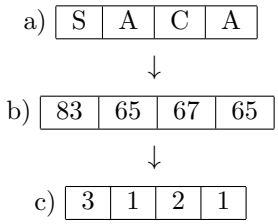


Abbildung 4.7: Beispiel für ein effektives Alphabet: a) zeigt den Eingabestring b) zeigt die entsprechenden ASCII-Werte c) zeigt die entsprechende effektive Darstellung.

Innerhalb unseres Frameworks wird das effektive Alphabet mittels des Objekts `alphabet` realisiert. Dieses nimmt dazu einen `string_span` entgegen, merkt sich welche Zeichen darin vorkommen und überschreibt schließlich die Eingabe mit den jeweiligen effektiven Werten.
Das Framework bestimmt für jede Eingabe automatisch das effektive Alphabet, welches dann an die Algorithmen übergeben wird.

4.2.2 Wordpacking

Beim Wordpacking [36] handelt es sich um eine Technik, bei der mittels Transformation der Eingabe mehrere Zeichen mit nur einer Vergleichsoperation betrachtet werden sollen. Dabei wird angenommen, dass die Alphabetgröße $|\Sigma|$ durch einen nicht-negativen Integer dargestellt werden kann.
Die Zeichen des Alphabets werden dann durch einen numerischen Wert $x \in \mathbb{N}$ entsprechend ihrer Wertigkeit im Intervall $[l, k]$, mit $l, k \in \mathbb{N}$, dargestellt, wobei das Terminalsymbol \$ die Wertigkeit 0 hat. Im Falle eines effektiven Alphabets würde dieses Intervall also $[1, |\Sigma|]$ entsprechen. Darauf aufbauend berechnen wir den maximalen Wert für $r \in \mathbb{N}$, sodass $(|\Sigma|+1)^r$ in ein Maschinenregister passt. Die Idee ist es nun den ungenutzten Platz innerhalb eines Registers zu nutzen, um dort zusätzliche Informationen über die nachfolgenden Zeichen mitzuspeichern und dabei den Stellenwert dieses Zeichens mit zu berücksichtigen. Dafür transformieren wir die Stellen des gegeben Textes T in den gepackten T_{packed} wie folgt:

$$\mathsf{T}_{packed}[i] = \sum_{j=1}^{r} x_{i+j-1} \cdot (|\Sigma| + 1)^{r-j}, \text{mit } x_i = 0 \text{ für } i \geq n \tag{4.1}$$

Dabei multiplizieren wir jede Integerrepräsentation eines Buchstabens zunächst mit $(|\Sigma| + 1)^{r-1}$, dann die darauf folgende Stelle mit $(|\Sigma| + 1)^{r-2}$ etc., wobei der Faktor $(|\Sigma|+1)^{r-j}$ mit einer Gewichtung vergleichbar ist. Dadurch erhalten wir für jede Stelle einen Wert, der die Wertigkeit von sich und seinen $r - 1$ Nachfolgern repräsentiert. Sehen wir uns dazu ein Beispiel an:
Gegeben sei der String $\mathsf{T} = abac\$$ mit $|\Sigma| = 3$. Die Wertigkeiten der Zeichen lassen sich wie folgt darstellen:

Zeichen	Wert
\$	0
a	1
b	2
c	3

Der Einfachheit halber neben wir $r = 3$ an. Daraus resultiert folgende Transformation des Textes:

$$\begin{aligned}
\mathsf{T}_{packed}[0] &= w(a) * 4^2 + w(b) * 4^1 + w(a) * 4^0 = 25 \\
\mathsf{T}_{packed}[1] &= w(b) * 4^2 + w(a) * 4^1 + w(c) * 4^0 = 39 \\
\mathsf{T}_{packed}[2] &= w(a) * 4^2 + w(c) * 4^1 + w(\$) * 4^0 = 28 \\
\mathsf{T}_{packed}[3] &= w(c) * 4^2 + w(\$) * 4^1 = 48 \\
\mathsf{T}_{packed}[4] &= w(\$) * 4^2 = 0
\end{aligned}$$

Hätten wir vor der Transformation die Stellen 0 und 2 des Textes verglichen, wäre das Resultat Gleichheit. Vergleichen wir dagegen die selben Stellen in der transformierten Eingabe, können wir anhand der zusätzlichen Informationen über die nachfolgenden Stellen sagen, dass das in 0 beginnende Suffixe lexikographisch kleiner ist als das an Stelle 2. Dadurch kann beispielsweise ein vergleichsbasierter Sortieralgorithmus mit dem selben Aufwand eine genauere Sortierung berechnen.
Das Wordpacking lässt sich zudem beschleunigen, indem bei der Berechnung das Ergebnis der vorherigen Stelle mit berücksichtigt wird. Für $i > 0$ lässt sich die Berechnung wie folgt umformulieren:

$$\mathsf{T}_{packed}[i] := (\mathsf{T}_{packed}[i-1] \% (|\Sigma| + 1)^{r-1}) * (|\Sigma| + 1) + x_{i+r} \qquad (4.2)$$

Durch die Modulo-Operation wird der Beitrag der $(i-1)$-ten Stelle aus dem Beitrag entfernt. Dazu erhöhen wir die Wertigkeit der übrigen Stellen mit der Multiplikation und erhöhen den Beitrag zudem um den Wert des neu ins Fenster gerückten Zeichens. Runden wir darüber hinaus $(|\Sigma| + 1)$ auf die nächste Zweierpotenz auf, lassen sich die Modulo- und Multiplikationsoperatoren durch schnellere *shift*- und *and*- Operationen ersetzen.
Dieses Verfahren ist vor allem bei vergleichsbasierten Sortierern hilfreich, da mit einem Vergleich direkt r Stellen berücksichtigt werden. Genutzt wird dieses beispielsweise vom qSufSort (Kapitel 5.1) oder vom Prefix Doubling Algorithmus (Kapitel 5.2).

4.2.3 ISAtoSA

Einige der im Framework enthaltenden Algorithmen, wie beispielsweise der mSufSort oder der qSufSort, konstruieren anstelle des eigentlichen Suffix-Arrays stattdessen die inverse Permutation, das inverse Suffix-Array.

Definition 10 (Inverses Suffix-Array)**.** *Das Inverse Suffix-Array, kurz* ISA*, bezeichnet ein Array, in dem zu jedem Suffix sein lexikographischer Rang gespeichert wird, der seiner Position im Suffix-Array entspricht.*
Es gilt also $\mathsf{ISA}[\mathsf{SA}[i]] = i$.

Um daraus das SA konstruieren zu können, wurden innerhalb des Frameworks drei unterschiedliche Ansätze implementiert, die wir im Folgenden vorstellen werden.

Simple Scan

Wie der Titel andeutet, handelt es sich bei dieser Variante um den straightforward Ansatz, der über die Indizes iteriert und die Einträge im SA wie folgt überschreibt:

$$\mathsf{SA}[\mathsf{ISA}[i]] = i, \text{ für } 0 \leq i < n \tag{4.3}$$

Voraussetzung für diese Methode ist, dass der Speicherplatz für sowohl das SA, als auch das ISA bereits verfügbar ist. Nachteil dieser Methode ist es, dass viele zufällige Sprünge innerhalb des Suffix-Arrays gemacht werden, worunter die Cache-Effizienz leidet.

Inplace Scan

Im Fall, dass ausschließlich Speicher für das ISA zur Verfügung steht, eignet sich der Simple Scan nicht, da dafür zusätzlicher Speicher für ein Array der Länge des Textes allokiert werden müsste. Daher wurde darüber hinaus eine Variante implementiert, die *inplace* arbeitet, also ausschließlich das gegebene ISA Array nutzt und darin auch das Ergebnis speichert [37]. Voraussetzung dafür ist, dass die Ränge im vorliegenden ISA kleiner 0 sind.
Die Idee ist, es zyklisch die Ränge im ISA durch die Positionen im SA zu ersetzen. Angenommen, wir finden Rang $r = \mathsf{SA}[s]$. Nun überschreiben wir $\mathsf{ISA}[r-1] = r$ merken uns aber den vorherigen Inhalt, um nach dem Überschreiben weiter dahin springen zu können. Auf diese Weise durchlaufen wir das Array bis jeder Rang durch einen SA-Index ersetzt wurde.
Vorteil dieser Methode ist, wie bereits erwähnt, dass kein zusätzlicher Speicherplatz für das Ergebnisarray benötigt wird. Der allerdings weiterhin bestehende Nachteil ist, dass viele zufällige, cache-unfreundliche Sprünge gemacht werden. Zudem wird das letzte Bit zur Unterscheidung zwischen Rängen und Indizes benötigt, wodurch der Wertebereich eingeschränkt wird.

Multi Scan

Die Multi Scan Variante von Maniscalco und Puglisi [37] soll im Gegensatz zu den beiden vorherigen Methoden cache-freundlich arbeiten und dabei weniger zusätzlichen Speicher benötigen als die *Simple Scan* Variante.
Wir definieren Q_i mit $i \in \{1, 2, 3, 4\}$, als i-tes Viertel des SA. Zu Beginn des Algorithmus werden zwei zusätzliche Arrays x_a und x_b mit jeweils der Größe $\frac{n}{4}$ angelegt. In der ersten Phase wird das ISA von links nach rechts durchlaufen und falls ein Rang r aus Q_1 gefunden wurde, wird dieses an die Stelle $x_a[r]$ verschoben und seine Position in ISA als leer markiert. Nach Abschluss der ersten Phase liegt Q_1 sortiert in x_a.
In der zweiten Phase scannen wir das ISA erneut von links nach rechts. Finden wir nun einen Rang $r \in Q_2$ speichern wir diesen an der jeweiligen Position in x_b und markieren auch hier die Position in ISA als leer. An jede leere Position die wir während des Scans finden oder erzeugen, verschieben wir die erste besetze Position aus x_a und markieren ihre Zugehörigkeit zu Q_1 mit Hilfe eines Bit-Tags.
Die dritte und vierte Phase laufen analog ab. Wir durchlaufen also das ISA und verschieben die jeweiligen Ränge aus Q_i in das zu dem Zeitpunkt freie Hilfsarray. Nach Ablauf der vierten Phase haben wir also die Ränge aus Q_4 in x_b gespeichert und $\frac{n}{4}$ freie Stellen am Ende des ISA Arrays. Wir kopieren x_b ans Ende von ISA, wodurch die Elemente aus Q_4 dann an der richtigen Stelle stehen und nicht weiter beachtet werden müssen.
In der nächsten Phase müssen dann also ausschließlich die Ränge aus Q_1, Q_2 und Q_3 in die richtige Reihenfolge gebracht werden. Dazu scannen wir erneut das ISA Array von links nach rechts bis wir die Position $\frac{3n}{4} - 1$ erreichen. Finden wir dabei ein Element aus Q_2, verschieben wir dieses nach x_a und Elemente aus Q_3 nach x_b. Ränge aus Q_1 werden dabei an die erste freie Position in ISA verschoben. Die Zugehörigkeit eines Ranges lässt sich dabei anhand des Tags feststellen. Zum Ende des Scans haben wir dann also Q_1 am Anfang des ISA, Q_2 in x_a und Q_3 in x_b.
In der letzten Phase wird dann x_a nach $\mathsf{ISA}[\frac{n}{4}, ..., \frac{n}{2} - 1]$ und x_b nach $\mathsf{ISA}[\frac{n}{2}, ..., \frac{3n}{4} - 1]$ verschoben. Wir erhalten also an dieser Stelle das fertige SA Array. EIn Beispiel für diese Variante zeigt Abb. 4.8
Auch wenn bei dieser Variante deutlich mehr Durchläufe durch das ISA Array gemacht werden, behaupten die Autoren, dass dadurch das es sich dabei lediglich um cache-freundliche Scans von links nach rechts handelt, Laufzeit im Gegensatz zu den cache-unfreundlichen Varianten gewinnen zu können. Nachteile dieser Variante sind jedoch zum einen die höhere Komplexität im Vergleich zu den beiden vorherigen Varianten und zum anderen, dass hierbei die drei MSB für Tags benötigt werden, wodurch hier, noch mehr als bei der inplace Variante, der Wertebereich eingeschränkt wird.

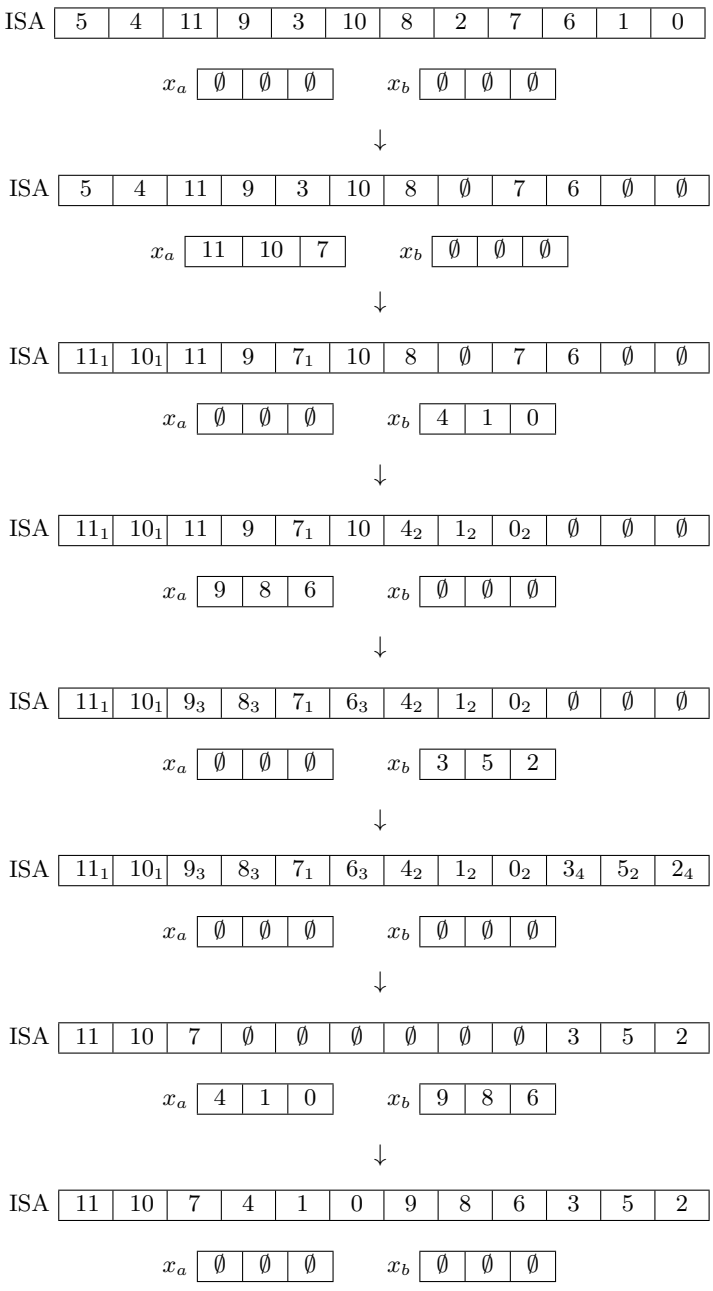


Abbildung 4.8: Beispiel für den ISAtoSA Multi Scan anhand des ISA für den Text $\mathsf{T} = MISSISSIPI\$$.

Kapitel 5

SACA Übersicht

5.1 Historie

Alle in dieser Ausarbeitung vorgestellte Algorithmen zur Konstruktion des Suffix-Arrays lassen sich in drei Gruppen einteilen, welche die grundlegenden Sortierverfahren beschreiben.

Die erste Gruppe bilden die sogenannten *prefix doubler*, zu welcher die Algorithmen Prefix Doubling und qSufSort gehören. Wie in Abschnitt 5.2.1 beschrieben wird, sortieren diese die Präfixe der Suffixe und können durch geschickte Ideen die Sortierung beschleunigen, indem die Anzahl der betrachteten Präfixe in jedem Schritt verdoppelt wird. Dabei setzte der SACA Prefix Doubling, welcher der Vorgänger des in dieser Ausarbeitung behandelten gleichnamigen SACAs ist, die Grundlagen für diese Kategorie. Darauf aufbauend ermöglichte qSufSort eine effizientere Konstruktion des Suffix-Arrays. Neben diesen beiden Algorithmen gehört auch der Algorithmus BPR teilweise in diese Gruppe, nutzt jedoch ebenfalls Ideen eines zweiten Verfahrens, dem *Induzieren*.

Zur Gruppe der Induzierer gehören auch die Algorithmen Deep-Shallow, DivSufSort, mSufSort, SAIS und GSACA. Sie sortieren Suffixe anhand bereits zuvor sortierter Suffixe. Dieses Konzept wird näher im Absatz 5.2.3 beschrieben. Deep-Shallow war von den hier behandelten Algorithmen der erste, welcher dieses Verfahren nutzte. Von ihm wurde der Algorithmus DivSufSort inspiriert, dessen Implementierungsdetails teilweise in die Entwicklung von SAIS eingingen. SAIS führte Ideen ein, welche auch von GSACA aufgegriffen wurden. Hingegen bildet mSufSort eine neu Vererbungslinie und baut auf keinem der anderen SACAs auf. Auch nzSufSort und SACA-K gehörten zu dieser Gruppe, kombinieren das induzierte Sortieren jedoch mit dem Konzept der letzten Gruppe.

Diese nutzt Rekursion, um kleinere Teilmengen von Zeichen zu Sortieren, welche dann zum finalen Suffix-Array zusammengesetzt werden. Zusätzlich zu nzSufSort und SACA-K wird dieses Sortierverfahren auch von den SACAs DC3 und SADS verwenden. Der erste Algorithmus, welcher rekursiv arbeitete, um ein Suffix-Array zu erstellen, war DC3. Der Algorithmus nzSufSort wurde unter

anderem von DC3 inspieriert, und greift manche der enthaltenen Ideen auf. Auch die beiden Suffix-Array-Konstruktionsalgorithmen SAIS und SADS gehörten zu dieser Gruppe. SACA-K, welcher sowohl induziert als auch rekursiv arbeitet, ist eine Weiterentwicklung der Ideen von SAIS und SADS und stellt dabei sowohl eine Verbesserung der Laufzeit als auch des Speicherverbrauchs dar.

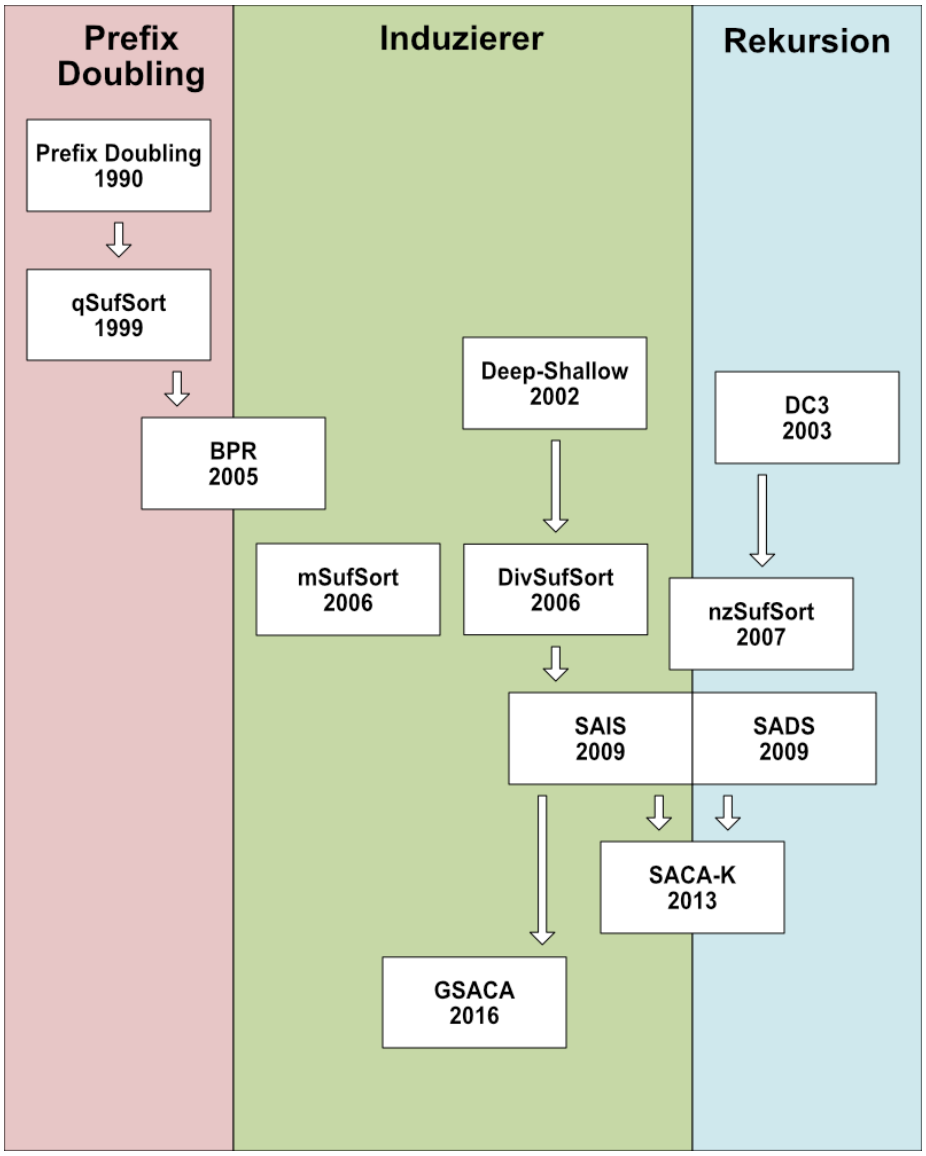


Abbildung 5.1: Geschichtliche Entwicklung von SACAs, die in dieser Ausarbeitung betrachtet werden.

5.2 Ansätze

5.2.1 Doubler

Der erste besprochene Ansatz behandelt das Doubling, auch Prefix-Doubling genannt. Die grundsätzliche Idee ist es, für jedes T_i nur einen Präfix von 2^k Zeichen zu betrachten und lexikographisch zu sortieren. Falls die so betrachteten $T[i, i+2^k)$ Strings paarweise verschieden sind, ergibt sich aus ihren Startindizen i das gesuchte Suffix-Array.

Der Prozess wird iterativ durchgeführt, und beginnt bei $k = 1$. Falls nach einer Sortierung individuelle Präfixe mehr als einmal vorkommen, sprich nicht *eindeutig* sind, wird der Vorgang mit $k + 1$ wiederholt. Erst wenn alle Suffixe eindeutig sortiert sind, endet das Prefix-Doubling.

Da die Länge und Häufigkeit von gemeinsamen Prefixen somit das Laufzeitverhalten von Doubling-Algorithmen beeinflusst, definieren wir für sie relevante Kenngrößen:

Lcp(i, j) bezeichnet die längste gemeinsamen Präfix Länge (**longest common prefix**) zwischen $\mathsf{SA}[i]$ und $\mathsf{SA}[j]$, und ist 0 wenn $i, j \notin [0, n)$ oder die beiden Suffixe keinen gemeinsamen Präfix haben. Für jedes S_i definieren wir **dps**$(i) = 1 + \max\{\mathrm{lcp}(i-1, i), \mathrm{lcp}(i, i+1)\}$ als charakteristische Präfixlänge (**distinguishing prefix size**). Wir definieren:

$$\mathrm{maxlcp} := \max_{i \in [0,n)} \mathrm{lcp}(i, i+1) \qquad \log \mathrm{dps} := \frac{1}{n} \sum_{i \in [0,n)} \log(\mathrm{dps}(i))$$

Umgangssprachlich drückt maxlcp aus, welche maximale Prefixlänge das Doubling erreichen muss, während $\log \mathrm{dps}$ die durchschnittlichen Anzahl an Doubling-Schritten ausdrückt.

Um ein Suffix-Array für T zu bestimmen, müssen alle Suffixe S_i lexikographisch sortiert werden. Wir stellen fest, dass hierfür von der Sortierfunktion jeweils nur ein Präfix von bis zu $\mathrm{dps}(i)$ Zeichen von S_i betrachtet werden muss.

5.2.2 Sortierer

Ein prinzipiell einfacher Ansatz, um SA zu konstruieren, ist selbstverständlich das Sortieren von Suffixen mittels einem allgemeinen Sortieralgorithmus, beispielsweise mit Quicksort oder Radixsort.

Für die Verwendung eines Sortieralgorithmus, um SA direkt zu konstruieren, muss eine passende Vergleichsfunktion benutzt werden: Nehmen wir an, die Menge die sortiert werden soll, ist bereits ein Array von disjunkten Suffix-Indizes [1]. In der Vergleichsfunktion vergleichen wir dann die Suffixe aus dem Text, die am entsprechenden Index starten. In Abbildung 1 findet sich Pseudocode für eine simple Vergleichsfunktion, welche die gesamten Suffixe naiv miteinander vergleicht.

[1] Dies kann zu Beginn sichergestellt werden, indem das Array mit den Zahlen von 0 bis $n-1$ (beziehungsweise n, wenn das Sentinel mitsortiert werden soll) gefüllt wird.

```
def compare(i, j):
    if T[i] == T[j]:
        return compare(i + 1, j + 1)
    return T[i] < T[j]
```

Abbildung 5.2: Pseudocode einer naiven Suffixvergleichsfunktion

Es ist klar, dass mit einem $\mathcal{O}(n \log n)$-Sortieralgorithmus wie beispielsweise Merge- oder Introsort der gesamte SACA eine Worst-Case-Komplexität von $\mathcal{O}(n^2 \log n)$ hat, da jeder Suffixvergleich eine Worst-Case-Laufzeit von $\mathcal{O}(n)$ hat. Zum Vergleich: Es gibt einige Linearzeitalgorithmen für die SA-Konstruktion, beispielsweise DC3 [29] oder SAIS [44].

Die alleinige Sortierung von ganzen Suffixen ist offensichtlich nicht effizient. Viele unserer implementierten Algorithmen verwenden allerdings zum Teil allgemeine Sortierverfahren, um verschiedene Mengen zu sortieren, beispielsweise Suffix-Indizes, eine Menge von Zeichen, ganze Suffixe oder beliebige Strings. Beispielsweise verwendet Multikey-Quicksort den Quicksort-Algorithmus, um eine Menge von Zeichen zu sortieren. Deep-Shallow verwendet Introsort, um die Suffixe nach ihrer Länge zu sortieren und DC3 benutzt Radixsort, um Triplets zu sortieren. In DivSufSort werden Stringsortierer benutzt, um bestimmte Substrings zu sortieren. Daher ist es für unser Framework wichtig, auch diese allgemeinen Sortierverfahren angepasst auf unseren Use-Case effizient zu implementieren. In Abschnitt 4.1 findet sich eine Übersicht über die von uns implementierten beziehungsweise extern eingebundenen Sortieralgorithmen.

5.2.3 Induzierer

Das Prinzip des *Induced Sortings* (zu deutsch *induziertes Sortieren*) baut darauf auf, dass das gesamte Suffixarray aus mehreren einzelnen sortierten Komponenten zusammen gesetzt werden kann. Diese Komponenten entstehen aus den Suffixen des Eingabestrings und hängen maßgeblich von dem ersten Symbol der zugehörigen Suffixe ab. Die Idee des Induced Sortings ist es, zuerst für jedes im String vorkommenden Symbol σ alle Suffixe des Eingabestrings, welche mit σ beginnen, in den σ-*Bucket* ein zu ordnen. Sind für jedes im Text vorkommende Symbol alle Suffixe in ihren Buckets korrekt geordnet, so können die Buckets aneinander gereiht werden, vorgegeben durch die Ordnung der Symbole selber, um das vollständige SA zu erhalten.

Es folgt ein Beispiel einer Zerteilung eines SA in die Buckets der jeweiligen Symbole, anhand des Textes $\mathsf{T} = abracadabra\$$, mit dem dazugehörigen Suffixarray $\mathsf{SA}(\mathsf{T}) = [11, 10, 7, 0, 3, 5, 8, 1, 4, 6, 9, 2]$. Dieses lässt sich wie folgt unterteilen:

$$\mathsf{SA}(\mathsf{T}) = \underbrace{[11]}_{\$} \cdot \underbrace{[10, 7, 0, 3, 5]}_{a} \cdot \underbrace{[8, 1]}_{b} \cdot \underbrace{[4]}_{c} \cdot \underbrace{[6]}_{d} \cdot \underbrace{[9, 2]}_{r}$$

Bei dieser Unterteilung fällt sofort auf, dass der Sentinel \$ immer einen eigenen Bucket mit nur einem Eintrag zugeteilt bekommt, welcher immer am Anfang des SAs steht. Das Alphabet eines Strings T wird hier in den meisten Fällen mit $\Sigma(\mathsf{T})$ angegeben und beinhaltet nicht den Sentinel.
Die gängigen Induzierer unterteilen jeden Bucket in zwei Sub-Buckets, in denen die Suffixe nach dem Kriterium eingeordnet werden, ob ihr zweites Symbol größer oder kleiner als ihr erste Symbol ist. Wir führen daher die folgende Notation ein für ein Suffix S_i (bzw. analog für die Stelle $\mathsf{T}[i]$):

- Wir sagen S_i ist ein *S-Typ*, falls $\mathsf{T}[i] < \mathsf{T}[i+1]$ oder $\mathsf{T}[i] = \$$ gilt.
- Wir sagen S_i ist ein *L-Typ*, falls $\mathsf{T}[i] > \mathsf{T}[i+1]$ gilt.
- Weiterhin haben S_i und S_{i+1} den selben Typ, falls $\mathsf{T}[i] = \mathsf{T}[i+1]$ gilt.
- Wir sagen S_i für $i > 0$ ist ein *LMS-Suffix* (Leftmost-S-Type-Suffix), falls S_i ein S-Typ ist und S_{i-1} ein L-Typ ist.
- Wir sagen $\mathsf{T}[i, j]$ ist ein *LMS-Substring*, falls S_i und S_j LMS-Suffixe sind oder $\mathsf{T}[i, j] = \$$ gilt.
- Wir sagen S_i für $i > 0$ ist ein *RMS-Suffix* (Rightmost-S-Type-Suffix), falls S_i ein S-Typ ist und S_{i+1} ein L-Typ ist.
- Wir sagen es gilt $\text{Type}(\mathsf{S}_i) = \text{S}$, falls S_i ein S-Typ ist (analog für L-Typen).

Analog wird das Suffix S_i in diesem Kontext auch als $\text{suf}(\mathsf{T}, i)$ bezeichnet, wenn klar gemacht wird, um welchen String T es sich handelt. Diese Unterscheidung kann bei Induzierern sehr wichtig sein, da sie für ihre Rekursionsinstanzen neue Strings erzeugen.

Induzierer unterteilen jeden Bucket nun in einen L- und einen S-Bucket. Dabei befinden sich in den L-Buckets alle Suffixe welche ein L-Typ sind – analog für S-Buckets. Für alle Indizes i, j mit $\mathsf{T}[i] = \mathsf{T}[j]$, so dass S_i ein L-Typ und S_j ein S-Typ ist, gilt, dass i vor j in SA vorkommt. Intuitiv formuliert kommt ein L-Bucket für das Symbol σ also immer vor dem S-Bucket von σ, da die Suffixe im L-Bucket ein niedrigeres zweites Symbol enthalten als die Suffixe im S-Bucket.
Dazu betrachten wir das Beispiel von oben, mit dem Text $\mathsf{T} = abracadabra\$$ und dem dazugehörigen Suffixarray $\mathsf{SA}(\mathsf{T}) = [11, 10, 7, 0, 3, 5, 8, 1, 4, 6, 9, 2]$. Der a-Bucket besteht aus den Einträgen $[10, 7, 0, 3, 5]$, wie wir oben bereits gesehen haben, setzt sich aber aus dem L-Bucket $[10]$ und dem S-Bucket $[7, 0, 3, 5]$ zusammen, da das Suffix $\mathsf{T}[10, 12) = a\$$ ein L-Typ ist, und alle anderen Suffixe S-Typen sind.
Oft wird das erste Element eines Buckets als *start* und das letzte Element eines Buckets als *end* bezeichnet.

Eine weitere gemeinsame Eigenschaft der Induzierer ist das Konzept, nach dem sie die Buckets jedes Suffixes ordnen. Sie ordnen dazu zuerst die LMS-Substrings in ihre jeweiligen Buckets ein. Wird dann erkannt, dass es zwei identische LMS-Substrings gibt, welche natürlich voneinander verschiedene LMS-Suffixe haben müssen, so werden diese mittels eines Rekursionsaufrufes und einer Umbenennung abhängig von ihren Positionen neu geordnet, so dass sich am Ende der Rekursion eine korrekte Ordnung der LMS-Substrings ergibt aus der sich auch die Ordnung der LMS-Suffixe herleiten lässt.
Im Folgenden wird erklärt, wie aus der korrekten Ordnung der LMS-Substrings bzw. der LMS-Suffixe das gesamte SA induziert werden kann:

- **Einordnung der LMS-Typen in** SA: Der Eingabestring wird von rechts nach links durchgelaufen und jeder LMS-Typ wird so weit rechts wie möglich in dem ihm zugehörigen S-Bucket platziert.
- **Erste Iterationsphase**: SA wird von links nach rechts durchgelaufen und von jedem Eintrag i wird das Suffix an der Stelle $i-1$ überprüft. Ist dieses ein L-Typ, so wird es in den Bucket passend links eingeordnet.
- **Zweite Iterationsphase**: SA wird von rechts nach links durchgelaufen und von jedem Eintrag i wird das Suffix an der Stelle $i+1$ überprüft. Ist dieses ein S-Typ, so wird es in den Bucket passend rechts eingeordnet.

Es gibt dabei viele verschiedene Implementierungen dieses Vorgangs, welche von Algorithmus zu Algorithmus unterschiedlich sein können. Die größten Unterschiede dabei liegen hauptsächlich bei der Berechnung der Typen, welche entweder dynamisch berechnet werden können oder am Anfang berechnet und dann für spätere Aufrufe gespeichert werden können. Es lässt sich zeigen:

Lemma 1 (Gemeinsamer Präfix)**.** Haben zwei Suffixe S_i und S_j einen gemeinsamen Präfix der Länge *offset*, so ist $\mathsf{S}_i \leq \mathsf{S}_j$ genau dann, wenn $\mathsf{S}_{i+offset} \leq \mathsf{S}_{j+offset}$. Dies folgt direkt aus der Definition der lexikographischen Ordnung.

Kapitel 6

Suffix-Array-Konstruktionsalgorithmen

6.1 qSufSort

In diesem Abschnitt wird der von Larson und Sadakane entworfene Algorithmus qSufSort [33] vorgestellt. Dieser garantiert für die Konstruktion des Suffix-Arrays eine Laufzeitschranke von $\mathcal{O}(n \log n)$ und benötigt dafür ausschließlich zwei Integerarrays der Länge n.

Dabei werden wir uns zunächst im folgenden Abschnitt mit der allgemeinen Funktionsweise des Algorithmus auseinander setzen. Anschließend werden die von den Autoren vorgestellten praktischen Verbesserungen präsentiert. In Kapitel 6.1.3 folgt dann ein Beispiel zur Durchführung des qSufSort. Schließlich werden wir Details zur Implementierung des Algorithmus innerhalb unseres Frameworks präsentieren.

6.1.1 Algorithmus

Grundgedanke des Algorithmus ist, dass die Suffixe eines Suffix die Präfixe der darauf im String folgenden Suffixe sind und diese bereits in einem anderen Bereich des Suffix-Arrays teils sortiert vorliegen. Anstatt also erneut Zeichen zu vergleichen, die wir bereits verglichen haben, nutzen wir vorhandene Teilresultate. Wichtig ist hierbei die Rolle der sogenannten h-Reihenfolge:

Definition 11 (h-Reihenfolge)**.** *Gegeben seien die Suffixe* S_i *eines Strings* T *mit* $i \in \{0, ..., n\}$*. Die Suffixe liegen in h-Reihenfolge vor, falls die Reihenfolge der lexikographischen Sortierung ausschließlich auf den ersten h Zeichen eines Strings beruht.*

Eine Sortierung nach ausschließlich dem Anfangszeichen eines Strings, würde demnach eine 1-Reihenfolge ergeben. Zudem muss eine h-Reihenfolge nicht

zwingend eindeutig sein. Beispielsweise sind die Positionen der Strings *aab*, *aaa* und *aaab* innerhalb einer 2-Reihenfolge untereinander nicht eindeutig.

Bezüglich der h-Reihenfolge lässt sich zudem der Begriff der Gruppe definieren:

Definition 12 (Gruppe)**.** *Sei* SA *ein Suffix-Array in h-Reihenfolge. Eine Sequenz aufeinander folgender Suffixe in* SA*, dessen ersten h Zeichen übereinstimmen, wird als Gruppe bezeichnet. Befindet sich innerhalb einer Gruppe nur ein Suffix, handelt es sich um eine sortierte Gruppe, andernfalls um eine unsortierte Gruppe. Eine Sequenz von sortierten Gruppen wird kombinierte sortierte Gruppe genannt.*

Logarithmische Schranke für Vergleiche

Grund dafür, dass der straight-forward Ansatz mittels direkter Vergleiche ähnlich wie bei der Sortierung von Zahlen nicht effizient genug arbeitet, ist, dass für den Vergleich zweier Suffixe die Anzahl der direkten Zeichenvergleiche linear in der Länge des Strings ist. Wünschenswert wäre es daher zwar ähnlich viele Vergleiche von Stellen, aber deutlich weniger Zeichenvergleiche durchführen zu müssen.

Ein Ansatz, um die Anzahl letzterer reduzieren zu können, wurde von Karp, Miller und Rosenberg [27] veröffentlicht und basiert auf der h-Reihenfolge. Folgendes Lemma von Manbers und Myers [36] ist dabei essentiell:

Lemma 2 (Manbers und Myers)**.** *Werden die Suffixe* S_i *zunächst nach ihrer Position in* SA *in der h-Reihenfolge und anschließend nach der Position des Suffixes* S_{i+h} *sortiert , erhalten wir die 2h-Reihenfolge.*

Aufbauend darauf, können wir also zunächst die 1-Reihenfolge vergleichsbasiert berechnen, indem wir nur anhand des ersten Zeichens sortieren und uns anschließend an den bereits teilweise sortierten Suffixen der Suffixe an i$+2^j-1$-ter Stelle orientieren, wobei $j \geq 1$ der Anzahl der Durchläufe entspricht.

Hierzu zur Veranschaulichung der Idee ein kleines Beispiel:

Betrachten wir den String $\mathsf{T} = babc\$$. Ein Suffix-Array in 1-Reihenfolge wäre $\mathsf{SA} = [4, 1, 2, 0, 3]$ (alternativ auch $\mathsf{SA} = [4, 1, 0, 2, 3]$, da das SA nach initialer Sortierung nicht eindeutig ist). Um jetzt beispielsweise die beiden mit b beginnenden Suffixe zu sortieren, können wir uns die Position des bei jeweils h beginnenden Suffixes im Array anschauen. Im Fall für $bc\$$ ist $\mathsf{SA}[\mathsf{T}[2+1]] = 4$ und für $babc\$$ $\mathsf{SA}[\mathsf{T}[0+1]] = 1$. Das heißt, die Suffixe, die in beiden Fällen auf das b folgen, sind bereits soweit lexikographisch im Suffix-Array sortiert, dass sich daraus eine Reihenfolge für die beiden Suffixe ablesen lässt und keine weiteren Vergleiche notwendig sind. Daraus resultiert $\mathsf{SA} = [4, 1, 0, 2, 3]$ und somit das fertig sortierte Suffix-Array.

Dadurch dass wir h in jedem Schritt verdoppeln, reduziert sich unsere obere Schranke für die Anzahl der Vergleiche auf $\mathcal{O}(n \log n)$.

```
1  def qSufSort(T)
2    Sort suffixes S_i according to their first character into SA
3    Init additional arrays V and L
4    h := 1
5    while (-L[0] != n):
6       Sort suffixes S_i in unsorted groups in SA by V[i + h]
7       Mark borders of A_left, A_middle and A_right
8       Calculate new groups
9       Update V and L
10      h = h · 2
```

Algorithmus 6.1: Pseudocode zum qSufSort

Vorgehen

Aufbauend auf den Resultaten aus dem vorherigen Abschnitt und dem des Ternary Quicksort (s. 4.1.1), lässt sich nun der qSufSort Algorithmus wie folgt konstruieren. Benötigt werden neben dem Ergebnisarray SA zunächst zwei Hilfsarrays V und L der Länge n+1.

Array V speichert für das Suffix-Array SA die Nummern der Gruppen. Die Nummer einer Gruppe ergibt sich aus dem maximalen Index im Suffix-Array, den diese Gruppe belegt. Eine Gruppe, die also die Positionen l bis k mit $l \leq k$ in SA belegt, hat demnach die Gruppennummer $V[l] = V[l+1] = ... = V[k] = k$.

Das Hilfsarray L dagegen speichert die dazu gehörigen Gruppenlängen. Um kombinierte sortierte Gruppen von unsortierten besser unterscheiden zu können, wird die Länge erst als negative Länge gespeichert. Für beliebige Gruppengrenzen l und k mit $l \leq k$ in SA ist also die Gruppenlänge, falls unsortiert, $L[l]=(k-l+1)$, anderenfalls $L[l]=-(k-l+1)$.

Der Algorithmus geht wie folgt vor:

In der ersten Zeile sortieren wir zunächst die Suffixe anhand ihres ersten Zeichens in das SA ein, wodurch dieses dann in 1-Reihenfolge vorliegt. Aufbauend darauf berechnen wir dann initial die Hilfsarrays L und V und setzen die Variable h auf unsere derzeitige h-Reihenfolge, also 1. Die nachfolgende Schleife wiederholen wir dann solange, bis unser Suffix-Array nur noch eine kombinierte sortierte Gruppe beinhaltet, also vollständig sortiert ist und somit die Länge der ersten sortierten Gruppe der Länge des Arrays entspricht. In dieser sortieren wir dann die einzelnen sortierten Gruppen, deren Startposition und Länge wir schnell mit V und L bestimmen können mit Hilfe des Ternary Quicksort auf Basis der Suffix-Array-Position des an $i + h$ startenden Suffixes. Die aus dem Ternary Quicksort resultierenden Partitionen A_{left}, A_{middle} und A_{right} werden entsprechend markiert. Damit versetzen wir das Ergebnisarray SA nach Lemma 2 in $2h$-Reihenfolge. Dementsprechend setzen wir dann h auf seinen doppelten Wert, nachdem wir in Zeile 8 und 9 die Informationen zu den Gruppen in L und V mit Hilfe der Split-Positionen des Quicksorts aktualisiert haben.

Laufzeit

Die initiale Sortierung in Schritt 1 kann in $\mathcal{O}(n \log n)$ Zeit durchgeführt werden, beispielsweise mit Hilfe des Quicksort-Algorithmus. Die Berechnung der Arrays V und L ist in linearer Zeit mittels eines Scans über das Array SA möglich, sowohl in den Zeilen 2 und 3 als auch in Zeile 8. Der für die Laufzeit ausschlaggebende Teil ist die in Zeile 4 beginnende Schleife. Wie wir bereits gesehen haben, rufen wir diese höchstens $\mathcal{O}(\log n)$ mal auf. Zwar rufen wir in jedem Durchlauf in Zeile 5 den Quicksort auf, woraus sich intuitiv eine Worst-Case Laufzeit $\mathcal{O}(n \log n) \cdot \mathcal{O}(\log n) = \mathcal{O}(n(\log n)^2)$ ergeben würde, jedoch können wir diese mittels einer genaueren Analyse deutlich schärfer wählen:

Als Sortierverfahren wurde sowohl in Zeile 1 als auch in Zeile 5 der Ternary Quicksort gewählt. Der Verlauf der Partitionierungen lässt sich implizit durch einen ternären Baum darstellen (siehe Abb. 6.1).

Die Partitionierung eines Arrays A mit $|A| = n$ lässt sich in $\mathcal{O}(n)$ berechnen. Da für die drei resultierenden Arrays A_{left}, A_{middle} und A_{right} gilt, dass diese in Summe höchstens genauso lang sind wie A, also n, können wir auch die Partitionierung der gesamten unter A liegenden Ebene in $\mathcal{O}(n)$ berechnen. Daraus folgt folgendes Lemma:

Lemma 3. *Die Partitionierung der auf einer beliebigen Ebene im ternären Baum befindlichen Teilmengen lässt sich in $\mathcal{O}(n)$ erzeugen.*

Es bleibt zu zeigen, wie hoch der ternäre Baum sein kann. Für den Pfad entlang der A_{middle}-Partitionen gilt, dass die Anzahl betrachteter Zeichen sich mit jeder Ebene verdoppelt. Damit ist die maximale Pfadlänge von der Wurzel zum Blatt entlang der A_k Mengen kleiner gleich $\log(n+1)$.

Für die Teilmengen A_{left} und A_{right} gilt, dass die Mengen in den jeweiligen Kindern höchstens halb so groß sind, da wir am Median teilen. Daraus folgt, dass auch entlang eines Pfades höchstens $\log(n+1)$ linke oder rechte Teilarrays sein können.

Daraus resultiert folgendes Lemma:

Lemma 4. *Die Höhe eines auf diese Weise erstellten ternären Baumes ist durch $\mathcal{O}(\log n)$ beschränkt.*

Fassen wir nun beide Lemmas inklusive unserer vorangegangen Überlegungen bezüglich der anderen Zeilen zusammen:

Lemma 5. *Die Berechnung des Suffix Arrays mit Hilfe des qSufSort-Algorithmus ist in $\mathcal{O}(n \log n)$ Zeit möglich.*

T	H	O	N	G	K	O	N	G	$
i	0	1	2	3	4	5	6	7	8

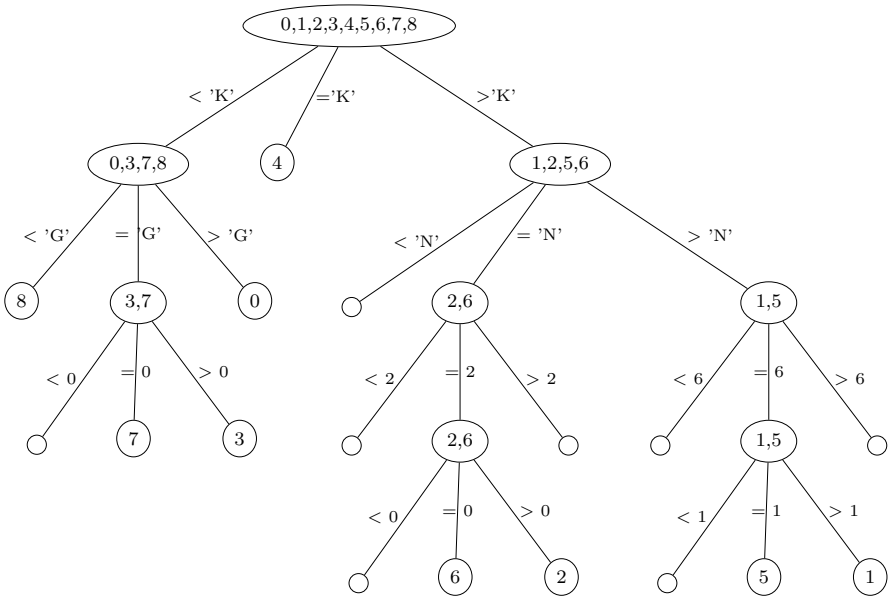


Abbildung 6.1: Der durch die ternäre Sortierung entstehende ternäre Baum anhand des Strings $\mathsf{T} = HONGKONG\$$.

6.1.2 Verbesserungen

Reduzierung auf ein Hilfsarray

Der Sinn und Zweck des Arrays L ist es vor dem tatsächlichen Aufruf des Ternary Quicksorts innerhalb der Schleife bereits sortierte kombinierte Gruppen in konstanter Zeit überspringen und zudem die Grenzen der unsortierten Gruppen für genau diesen Aufruf setzen zu können. Letzteres lässt sich allerdings implizit über über die Gruppennummern des V Arrays lösen. Können wir also zudem einen alternativen Speicherort für die Längen der kombinierten sortierten Gruppen finden, können wir uns das Hilfsarray L sparen.

Was bei den Bereichen der kombinierten sortierten Gruppen auffällt ist, dass der korrespondierende Bereich im Ergebnisarray SA nicht mehr verändert wird, da sich die sortierten Gruppen laut Definition bereits an der richtigen Stelle in SA befinden. Daher können wir genau diesen Bereich nutzen um dort die Länge der kombinierten sortierten Gruppen zu lagern.

Das Problem, das dabei entsteht ist, dass bei Terminierung keine gültige Lösung in SA steht, das diese überschrieben wurde. Jedoch können wir die Lösung mit Hilfe des Hilfsarrays V rekonstruieren, da es sich dabei um das inverse Suffix-Array handelt. Mit einer der in Kapitel 4.2.3 vorgestellten Techniken lässt

sich dieses nach Abschluss der Berechnung in das eigentliche Suffix-Array transformieren.

Um diese jetzt noch während der Berechnung unterscheiden zu können, ob es sich bei einem Eintrag in SA um die bisherige Position eines unsortierten Elements oder um die Länge einer kombinierten sortierten Gruppe handelt, negieren wir wie zuvor auch schon letzteres.
Da wir nun also alternative Wege haben, um an die Informationen aus L zu kommen und diese im Falle von kombinierten sortierten Gruppen speichern können, können wir uns dieses sparen. Dies hat in erster Linie positive Auswirkungen auf den Speicherverbrauch. Dadurch, dass wir bereits allokierten Speicher in Form der Bereiche in SA verwenden, können wir unseren Speicherverbrauch von drei Arrays der Länge $n + 1$ auf lediglich zwei reduzieren. Die benötigten Rechenoperationen erhöhen sich dagegen. Zum einen dadurch, dass das Ermitteln der Länge unsortierter Gruppen mehr Zugriffe als vorher benötigt, zum anderen durch die benötigte Invertierung des V Arrays um das Suffix-Array rekonstruieren zu können. Für eine verbesserte Laufzeit spricht allerdings, dass dadurch, das wir lediglich auf zwei Arrays arbeiten, unsere Implementierung womöglich cache-freundlicher arbeitet.

Sortieren und Updaten kombinieren

Ausgehend davon, dass wir noch das Hilfsarray L mitführen, berechnen wir dieses und das Hilfsarray V nach jedem Sortieraufruf separat mittels Durchlaufs durch das bisherige SA. Ziel ist es, diese Aktualisierungen schon bereits während der Sortierung durchführen zu können, um sich weitere Durchläufe durch die Arrays sparen zu können.

Zunächst können wir feststellen, dass das Aktualisieren der maximalen kombinierten sortierten Gruppen zum Beginn des nachfolgenden Sortieraufrufs verschoben werden kann.

Um darauf aufbauend die gesamte Aktualisierung in die Sortierung übernehmen zu können, gehen wir wie folgt vor:

1. Teile das Array A wie gewohnt nach dem Pivotelement in A_{left}, A_{middle} und A_{right} auf
2. Sortiere A_{left} rekursiv
3. Aktualisiere die Gruppennummern und -längen der Elemente in A_{middle}
4. Sortiere A_{right} rekursiv

Aktualisieren wir zuerst den „Gleich“- oder „Größer“-Teil, kann es passieren, dass ein Suffix eine niedrigere Gruppennummer zugewiesen bekommt als ein anderes Suffix vor ihm in SA, da Gruppennummern lediglich kleiner werden können. Da diese als Keys für die Sortierung dienen, hätten wir dabei einen ungültigen Zustand. Daher fangen wir zunächst beim „Kleiner“-Teil an.

Dies beeinflusst dennoch den Sortierprozess. Ohne diese Überlegung kann es nämlich sein, dass wir innerhalb der Sortierung für zwei Suffixe den selben

Key haben, diese also unsortiert bleiben und damit in der nächsten Iteration wieder aufgerufen werden, obwohl die betroffenen Stellen in der selben Iteration schon soweit verarbeitet wurden, dass eine Sortierung möglich gewesen wäre. Es würde also lediglich an der Propagierung der Ergebnisse scheitern, was mit Hilfe dieser Verbesserung für dann Fall, in welchem die betroffenen Keys schon vorher abgearbeitet worden sind, nicht passieren kann. Dadurch kann es in einigen Fällen zu Einsparungen von Sortieraufrufen kommen und dadurch zu einer Verbesserung der tatsächlichen Laufzeit.

Initiales Sortierverfahren

Bislang wurde nicht konkret festgelegt, welcher Sortieralgorithmus für die initiale Sortierung in Zeile 1 verwendet werden soll. Zwar wurde für die Analyse ein ternary Quicksort angenommen um mit Hilfe des ternären Baums eine möglichst scharfe Laufzeitschranke bestimmen zu können, dennoch können wir an dieser Stelle einen alternativen Algorithmus wählen, der womöglich effizienter arbeitet.

Eine Möglichkeit hierfür ist der Bucketsort-Algorithmus, welcher in linearer Zeit sortiert. Ist dabei $|\Sigma| \leq n + 1$, kann die Bucket-Sortierung direkt in SA stattfinden. Anderenfalls ist der Algorithmus hierbei ungeeignet.

Betrachten wir dabei zusätzlich die in Kapitel 4.2.2 vorgestellte Transformation der Eingabe mittels Wordpacking, können wir die Sortierung zusätzlich beschleunigen. Hierbei wird jedoch bei der Wahl von r vorausgesetzt, dass nicht nur $(|\Sigma| + 1)^r$ in ein Maschinenwort passt, sondern auch, dass $r \leq n$ gilt. Das daraus resultierende, neue Alphabet $|\Sigma|'$ kann zwar für einige Eingaben größer ausfallen als vorher, dennoch können wir durch die Bedingungen garantieren, dass die dazu gehörige transformierte Eingabe innerhalb des SA sortiert werden kann, ohne zusätzlichen Speicher allokieren zu müssen. Dadurch können wir eine initiale Sortierung durchführen, die das Suffix-Array in linearer Zeit in r-Reihenfolge bringt.

6.1.3 Beispiel

In diesem Abschnitt präsentieren wir den qSufSort, inklusive aller der im vorherigen Kapitel vorgestellten Verbesserungen, anhand eines Durchlaufs mit dem Eingabestring $\mathsf{T} = caabaccaabacaa\$$.
Dazu transformieren wir zunächst den gegebenen String mit Hilfe des in Kapitel 4.2.2 vorgestellten Wordpackings. Dabei gehen wir wie folgt vor:

Im Fall unseres Eingabestrings beträgt die effektive Alphabetgröße $|\Sigma| = 3$. Darauf aufbauend lassen sich folgende Wertigkeiten der Zeichen des Alphabets zuordnen:

Zeichen	Wert
\$	0
a	1
b	2
c	3

Wie in Kapitel 4.2.2 beschrieben ergibt sich der Wert für r aus der verfügbaren Registergröße und der Alphabetgröße des vorliegenden Textes. Der besseren Veranschaulichung halber nehmen wir in diesem Beispiel $r = 3$ an, das heißt, wir betrachten nach und nach drei Zeichen große Blöcke unserer Eingabe. Daraus und aus den Wertigkeiten der einzelnen Zeichen ergeben sich nach der Formel aus 4.2.2 folgende Werte für die Blöcke:

Substring	Wert
caa	53
aab	22
aba	25
bac	39
acc	31
cca	61
caa	53
aab	22
aba	25
bac	39
aca	29
caa	53
aa$	20
a$	16
$	0

Aufbauend auf den neu errechneten Wertigkeiten der einzelnen Stellen lässt sich nun die initiale Sortierung anhand des ersten Zeichens durchführen. Dafür kann beispielsweise wie im vorherigen Kapitel vorgeschlagen ein Bucketsort benutzt werden. Daraus resultiert das folgende Suffix-Array in 3-Reihenfolge:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA[i]	14	13	12	1	7	2	8	10	4	3	9	0	6	11	5
Wert	0	16	20	22	22	25	25	29	31	39	39	53	53	53	61

Aus dem vorliegenden Suffix-Array lässt sich nun das Hilfsarray V wie folgt initialisieren:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA[i]	-3	13	12	1	7	2	8	-2	4	3	9	0	6	11	-1
V[SA[i]]	0	1	2	4	4	6	6	7	8	10	10	13	13	13	14

Nach der Initialisierung beginnt nun die Prefix-Doubling Phase in der wir in jeder Iteration soweit wie möglich die unsortierten Gruppen innerhalb des Suffix-Arrays sortieren:

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	SA[i]	-3	13	12	1	7	2	8	-2	4	3	9	0	6	11	-1
	V[SA[i]]	0	1	2	4	4	6	6	7	8	10	10	13	13	13	14
$h=2$	V[SA[$i+h$]]				6	6	10	10			8	7	4	4	2	
	SA[i]	-3	13	12	1	7	2	8	-4	4	9	3	-1	0	6	-1
	V[SA[i]]	0	1	2	4	4	6	6	7	8	9	10	11	13	13	14
$h=4$	V[SA[$i+h$]]				10	9	8	7						6	6	
	SA[i]	-12	13	12	7	1	8	2	-4	4	9	3	-1	0	6	-1
	V[SA[i]]	0	1	2	3	4	5	6	7	8	9	10	11	13	13	14
$h=8$	V[SA[$i+h$]]													8	7	
	SA[i]	-15	13	12	7	1	8	2	-4	4	9	3	-1	6	0	-1
	V[SA[i]]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Die in rot markierten Zahlen stellen dabei Einträge unsortierter Gruppen dar. Zu Beginn handelt es sich dabei um die Intervalle $[3,4]$ (Suffixe *aabaccaabacaa* und *aabacaa*), $[5,6]$ (Suffixe *abaccaabacaa* und *abacaa*), $[9,10]$ (Suffixe *baccaabacaa* und *bacaa*) und $[11,13]$ (Suffixe *caabaccaabacaa*, *caabacaa* und *caa*). Wie zu erkennen ist, beginnen alle Suffixe innerhalb einer sortierten Gruppe mit den selben drei Zeichen, wodurch diese durch das Wordpacking in Kombination mit der initialen Sortierung nicht final sortiert werden konnten.

Jede dieser Gruppen versuchen wir nun mit Hilfe der Position des $(i+h)$-ten Suffix im Suffix-Array weiter zu sortieren, wobei zu Beginn $h=1$ gilt. Die Einträge der ersten beiden Gruppen zeigen dabei jeweils auf den selben Rang, wodurch hier eine feinere Sortierung nicht möglich ist. Die Einträge der Gruppe $[9,10]$ dagegen zeigen auf unterschiedliche Ränge, anhand derer wir diese Gruppe fertig sortieren können. Auch in der letzten Gruppe können wir die Sortierung verfeinern, haben allerdings auch wieder zwei gleiche Ränge, wodurch hier eine kleinere, unsortierte Gruppe übrig bleibt. Nach Abarbeiten der unsortierten Gruppen aktualisieren wir die Gruppenlängen in SA.

Für die nächste Iteration verdoppelt wir h auf 2 und wiederholen das Vorgehen für die übrigen unsortierten Gruppen, also $[3,4]$,$[5,6]$ und $[12,13]$. Die ersten beiden Gruppen lassen sich nun endgültig sortieren, wohingegen die Einträge der letzten Gruppe wieder auf den selben Rang zeigen.
Nach Aktualisierung der Gruppenlängen verdoppeln wir h erneut und versuchen in der dritten Iteration erneut die Gruppe $[12,13]$ zu sortieren. Diesmal lässt sich diese endgültig sortieren, wodurch nun das gesamte Suffix-Array ausschließlich aus sortierten Gruppen besteht. Nach Aktualisierung der Gruppenlängen terminiert die Prefix-Doubling Phase schließlich.

Um nun das finale Suffix-Array zu bekommen, müssen wir das Hilfsarray V invertieren. Dazu können wir eine der in Abschnitt 4.2.3 gezeigten Techniken verwenden. Daraus resultiert folgendes Suffix-Array:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
V[SA[i]]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA[i]	14	13	12	7	1	8	2	10	4	9	3	11	6	0	5

Damit terminiert der qSufSort und wir haben in SA das korrekte Suffix-Array gespeichert.

6.1.4 Implementierung

Naive Variante

Der qSufSort wurde zunächst in seiner naiven Variante implementiert, also ohne die von den Autoren vorgeschlagenen Verbesserungen aus Kapitel 6.1.2. Der Algorithmus bekommt, wie auch die anderen implementierten Algorithmen innerhalb des Frameworks, den Text `text`, das dazugehörige Alphabetobjekt `alphabet` und das bereits allokierte Outputarray `out_sa` übergeben. Für den übergebenen Text wird vom Algorithmus gefordert, dass dieser mit genau einem Senitel Symbol endet. Darüber hinaus wird das letzte Bit eines Wertes reserviert um auch negative Zahlen darstellen zu können.

Zu Beginn wird überprüft, ob der Text mindestens zwei Zeichen beinhaltet, falls nicht, terminiert der Algorithmus trivialerweise. Andernfalls werden zunächst die beiden Hilfsarrays `V` und `L` allokiert. Mit Hilfe des Ternary Quicksort wird dann der Text anhand des ersten Zeichens sortiert. Die daraus resultierende Sortierung wird in `out_sa` gespeichert. Darauf aufbauend werden dann die Inhalte der Hilfsarrays initialisiert. Der Präfixzähler h wird auf 1 gesetzt und die Prefix-Doubling Phase startet.

Dabei wird die Variable `counter` verwendet, die den aktuell betrachteten Index in `out_sa` speichert. Beginnt an dieser Stelle eine sortierte Gruppe, ist also der Eintrag im `L` Array negativ, überspringen wir diese, indem wir den aktuellen Counter `counter` um die Länge dieser Gruppe inkrementieren. Andernfalls handelt es sich um eine unsortierte Gruppe, die wir wieder mit Hilfe des Ternary Quicksort sortieren. Auf Basis dieser Sortierung aktualisieren wir dann die Gruppennummern.

Ist `counter` größer als die Länge von `out_sa`, terminiert die Schleife und wir aktualisieren die Gruppenlängen, verdoppeln h und überprüfen dann anhand der Länge der ersten sortierten Gruppe, ob das Array komplett sortiert ist. Ist dem so, terminiert der Algorithmus und in `out_sa` steht das fertig sortierte Suffix-Array. Andernfalls wird die Schleife erneut durchlaufen, bis das Array komplett sortiert ist.

Optimierung

Ausgehend von der naiven Implementierung, wurden die von Autoren vorgeschlagenen praktischen Verbesserungen implementiert.

Durch die erste Verbesserung, die Reduzierung auf ein Hilfsarray, konnte das zusätzliche Hilfsarray `L` eingespart werden. Stattdessen mussten die negativen Längen der sortierten Gruppen im Ausgabearray `out_sa` gespeichert werden. Da dieses innerhalb des Frameworks aus `unsigned int` besteht, wurde das letzte Bit vom Algorithmus reserviert, und falls nötig mit einer Bitmaske überschrieben, um „negative“ Werte darzustellen. Das nach Terminierung des Prefix-Doublings errechnete `ISA` wurde dann mit der Simple Scan Variante (s. Kapitel 4.2.3) ins Array `out_sa` invertiert.

Die zweite Verbesserung, die Kombination von Sortieren und Updaten, ersetzt den expliziten Aufruf des Ternary Quicksort innerhalb des Prefix-Doublings

durch eine Funktion, die selbst die Partitionsfunktion des Ternary Quicksort aufruft um die resultierenden Intervallgrenzen der Equal-Partition zum Updaten der Gruppen nutzen zu können.

Die dritte und letzte Verbesserung, die Beschleunigung der initialen Sortierung mit Hilfe von Wordpacking, transformiert nach Aufruf des Algorithmus den Text in das Ausgabearray `out_sa`, da das Textobjekt unveränderbar ist. Die resultierende Transformation wird dann, anders als von den Autoren vorgeschlagen, mit Hilfe der sequentiellen Variante des IPs^{4}o anstelle eines Bucketsort sortiert. Gründe dafür sind zum einen, das dieser unabhängig von der resultierenden Alphabetgröße durch das Wordpacking funktioniert und zum anderen Evaluierungen innerhalb unseres Frameworks gezeigt haben, dass dieser in diesem Fall die beste Laufzeit bietet.

6.2 GPU Prefix-Doubler

Mit fortlaufendem Fortschritt in der Entwicklung von General-Purpose-GPUs (GPGPUs), insbesondere innerhalb der letzten Jahre, wurde der Gebrauch von Grafikkarten als Koprozessor für grafikunabhängige Algorithmen immer populärer. Dabei ermöglicht die Bauweise der Grafikprozessoren eine hohe Parallelität und beschleunigt die Berechnungen enorm. Entwicklungsumgebungen wie das NVIDIA CUDA Toolkit werden fortlaufend weiterentwickelt und integrieren vor allem für allgemeine GPU-Berechnungen immer mehr Möglichkeiten. Zusätzlich bieten Libraries wie *Thrust* und *CUB* bereits eine Unterstützung allgemeiner Algorithmen auf der Grafikkarte wie Radixsort oder Präfixsummen an.

Auch im Bereich der Suffix-Array Konstruktionsalgorithmen entstanden für dieses Rechenmodell bereits Ansätze. Dazu gehört unter anderem der Prefix-Doubling Ansatz von Osipov [45], der Skew Ansatz von Deo und Keely [19] und ein Hybrid beider Varianten von Wang, Baxter und Owens [59]. Im Rahmen unserer Projektgruppe haben wir uns auf ersteren fokussiert und diesen innerhalb des Frameworks implementiert und evaluiert.

In diesem Kapitel werden wir dazu zunächst den sequentiellen Algorithmus von Osipov vorstellen und an einem Beispiel erläutern, bevor wir auf die Ideen der Parallelisierung und deren Implementierungen, speziell im Bezug auf die GPU, eingehen.

6.2.1 Algorithmus

Die Idee des Algorithmus von Osipov basiert sowohl auf den Prefix-Doublern von Larson und Sadakane [33], als auch von Manber und Myers [36]. In Bezug auf Parallelität hat Ersterer das Problem, dass die zu sortierenden Gruppen innerhalb einer Iteration unterschiedliche Längen haben können, wodurch unbalancierte Workloads entstehen. Der Algorithmus von Manber und Myers dagegen hat dieses Problem zwar nicht, sortiert dafür aber bereits fertig sortierte Suffixe erneut. Die Idee ist es nun, beide Ansätze zu kombinieren, also global zu sortieren, allerdings nur mit den Suffixen, die noch nicht fertig sortiert wurden oder für die Sortierung noch benötigt werden. Dabei soll folgendes Lemma helfen:

Lemma 6. *Falls in der i-ten Iteration des Manber-Myers Algorithmus gilt, dass*

- S_i *ist eine sortierte Gruppe in* SA_{2^k}
- $i < 2^{k+1}$ *oder* $\mathsf{S}_{i-2^{k+1}}$ *ist eine sortierte Gruppe*

dann gilt für alle nachfolgenden Iterationen $j > k$ *entweder* $i < 2^i$ *oder* S_{i-2^j} *ist eine sortierte Gruppe.*

Es besagt, dass sortierte Gruppen, welche im nächsten Schritt eine bereits sortierte Gruppe sortieren würden, nicht mehr betrachtet werden müssen. Dies ist möglich, da durch die Verdopplung der betrachteten Suffixlänge das hintere der beiden Suffixe immer wieder Gruppen sortieren würde, welche bereits durch

seinen Vorgänger sortiert wurden. Mit Hilfe dieses Lemmas können wir nun, falls die Bedingungen zutreffen, fertig sortierte Gruppen innerhalb des Sortierschritt nicht weiter beachten, um redundante Arbeit zu vermeiden.

In Algorithmus 6.2 können wir den Ablauf des Prefix-Doublers von Osipov sehen. In der ersten Zeile werden die Suffixe anhand ihrer ersten vier Zeichen sortiert und die daraus resultierende Reihenfolge in SA_4 gespeichert. Darauf aufbauend wird das ISA_4 konstruiert, darin bereits fertig sortierte Elemente mit einem Flag markiert und die Variablen $size$ und h initialisiert. Die folgende Schleife wird solange wiederholt, bis sich alle Elemente in sortierten Gruppen befinden. Nach der finalen Sortierung müssen wir noch das finale SA aus dem finalen ISA erzeugen.

Für die Sortierung eines Schleifendurchlaufes müssen wir zunächst die benötigten Tupel erzeugen. Dafür prüfen wir in Z. 9, ob wir das Suffix S_i mit dem Suffix S_j induzieren können, d. h. $i = \mathsf{SA}_h[j] - h$ ist ein gültiger Index und wir haben S_i noch nicht endgültig sortiert. Ist dieser Fall gegeben, erzeugen wir in Z. 10 das Tupel aus dem Suffxindex i, seinem h-Rang $\mathsf{ISA}_h[i]$ sowie seinem $2h$-Rang $\mathsf{ISA}_h[\mathsf{SA}_h[j]]$, welcher dem Rang des Suffixes S_j entspricht. Als nächstes prüfen wir, ob unser Suffix S_j für die folgenden Iterationen weiterhin benötigt wird (s. Lemma 6). Ist dies der Fall, so erzeugen wir das Tupel in Z. 14 aus dem Index, dem negierten Rang und dem Rang des Suffixes S_j. Für jedes erzeugte Tupel erhöhen wir s um eins. s entspricht somit der Anzahl an Tupeln und damit der Anzahl an Suffixen in SA_{2h}.

Um die neuen Ränge bestimmen zu können, müssen wir unsere Tupel zunächst stabil sortieren (Z. 16). Als Sortierschlüssel wählen wir hierbei den h-Rang. Daraufhin durchlaufen wir jedes Suffix in SA_{2h} und berechnen den neuen Rang. Ist der Rang des aktuell betrachteten Suffixes S_j größer als der Rang des aktuellen Kopfes, haben wir eine neue Gruppe gefunden (Z. 19) und setzen den Kopf auf den Anfang der neuen Gruppe. Andernfalls prüfen wir, ob das betrachtete Suffix j einen anderen $2h$-Rang hat als der aktuelle Kopf. In diesem Fall haben wir eine neue Gruppe durch die aktuelle Iteration und die Suffix-Länge $2h$ erhalten und können diese Suffixe zum ersten Mal eindeutig voneinander unterscheiden. Dafür setzen wir den Rang auf den Rang des Kopfes und addieren die Differenz der Positionen in SA_{2h} zwischen Suffix j und dem Kopf drauf. Sollten sowohl die h- als auch die $2h$-Ränge identisch sein, können wir die Suffixe noch nicht eindeutig unterscheiden und wir weisen S_j den Rang des Kopfes zu (Z. 26), da sich dessen Rang durch eine neue Gruppe in dieser Iteration verändert haben könnte (Z. 22–24).

Zum Abschluss der Iteration müssen wir die neuen Ränge in ISA_{2h} schreiben (Z. 27f.) und alle neuen, vollständig sortierten Gruppen mittels Negierung markieren (Z. 29). Die neue Größe für die nächste Iteration entspricht der Anzahl Tupel dieser Iteration und h wird für den nächsten Schleifendurchlauf verdoppelt (Z. 30).

```
1  osipov(T)
2  # Tupel = (SA_2h, h-rank, 2h-rank)
3   Sort suffixes S_i according to their first four characters in SA_4
4   Init ISA_4
5   Mark sorted Groups in ISA_4 by negation
6   size = n, h = 4
7   while (size > 0):
8     s = 0
9     for j in [0, size]:
10       i = SA_h[j] − h
11       if ((i > 0) ∧ (ISA_h[i] > 0)):
12          triples.add((i, ISA_h[i], ISA_h[SA_h[j]]))
13          s++
14       i = SA_h[j]
15       if ((ISA_h[i] < 0) ∧ (i − 2h ≥ 0) ∧ (ISA_h[i − 2h] ≥ 0)):
16          triples.add((i, ISA_h[i], −ISA_h[i]))
17          s++
18    sort(value(SA_2h, 2h-rank), key(h-rank))
19    head = 0
20    for j in [1, s]:
21       if (h-rank[j] > h-rank[head]):
22          head = j
23       else:
24          if (2h-rank[j] ≠ 2h-rank[head]):
25             h-rank[j] = h-rank[head] + j − head
26             head = j
27          else:
28             h-rank[j] = h-rank[head]
29    for i in [0, s]:
30       ISA_2h[SA_2h[i]] = h-rank[SA_2h[i]]
31    Mark sorted Groups in ISA_2h by negation
32    size = s, h = 2 · h
33  Calculate SA from ISA_h
```

Algorithmus 6.2: Der sequentielle Prefix-Doubling Algorithmus von Osipov.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\mathsf{SA}[j]$	14	13	1	7	12	2	8	4	10	3	9	0	6	11	5
Suffix	\$	a	a	a	a	a	a	a	a	b	b	c	c	c	c
$h = 2$		\$	a	a	a	b	b	c	c	a	a	a	a	a	c
$\mathsf{ISA}[\mathsf{SA}[j]]$	-0	-1	2	2	2	5	5	7	7	9	9	11	11	11	-14

Tabelle 6.1: Schritte vor dem Schleifendurchlauf: initiale Sortierung nach den ersten h-Zeichen (hier: $h = 2$), initiales ISA (in SA-Reihenfolge) und Markierung bereits sortierter Gruppen

j	$i = \mathsf{SA}_h[j] - h$	$i > 0 \wedge \mathsf{ISA}_h[i] > 0$	$i' = \mathsf{SA}_h[j]$	$\mathsf{ISA}_h[i'] < 0$ $\wedge(i' - 2h) \geq 0$ $\wedge\ \mathsf{ISA}_h[i' - 2h] \geq 0$
0	12	$\top \wedge \top = \top$	14	$\top \wedge \top \wedge \top = \top$
1	11	$\top \wedge \top = \top$	13	$\top \wedge \top \wedge \top = \top$
2	-1	$\bot$	1	$\bot$
3	5	$\top \wedge \bot = \bot$	7	$\bot$
4	10	$\top \wedge \top = \top$	12	$\bot$
5	0	$\top \wedge \top = \top$	2	$\bot$
6	6	$\top \wedge \top = \top$	8	$\bot$
7	2	$\top \wedge \top = \top$	4	$\bot$
8	8	$\top \wedge \top = \top$	10	$\bot$
9	1	$\top \wedge \top = \top$	3	$\bot$
10	7	$\top \wedge \top = \top$	9	$\bot$
11	-2	$\bot$	0	$\bot$
12	4	$\top \wedge \top = \top$	6	$\bot$
13	9	$\top \wedge \top = \top$	11	$\bot$
14	3	$\top \wedge \top = \top$	5	$\top \wedge \top \wedge \top = \top$

Tabelle 6.2: Bedingungen für die Erzeugung von Tupeln für $h = 2$. Die erste Bedingung (zweite Spalte) prüft, ob der Suffix-Index sortiert werden muss. Die zweite Bedingung (vierte Spalte) prüft, ob bereits sortierte Suffixe zum Sortieren anderer Suffixe in den Folgeiterationen benötigt werden (s. Lemma 6).

6.2.2 Beispiel

Im Folgenden gehen wir auf das bekannte Beispiel „caabaccaabacaa\$“ ein, um die Funktionsweise des Prefix-Doublers nach Osipov zu veranschaulichen. Um möglichst viele Schritte bzw. Verzweigungen darzustellen, setzen wir die initiale Länge der Suffixe auf $h = 2$ statt auf 4.

Zunächst sortieren wir unsere Suffixe nach den ersten zwei Zeichen und erzeugen basierend darauf das initiale ISA_2. Die Ränge bereits sortierter Gruppen werden negiert, so wie bei den Suffixen mit Index $14, 13$ und 5 zu sehen. Die Ergebnisse dieser drei Schritte sind in Tabelle 6.1 dargestellt. Als nächstes betrachten wir die erste Iteration unserer Schleife.

Zu Beginn einer jeden Iteration müssen wir die für die Sortierung benötigten Tupel erzeugen. Dafür schauen wir uns zunächst an, welche Tupel über-

Erzeugte Tupel	j	Sortierte Reihenfolge (nach h-Rang)
$(12, 2, -0)$	0	$(14, 0, -0)$
$(14, 0, -0)$	1	$(13, 1, -1)$
$(11, 11, -1)$	2	$(12, 2, -0)$
$(13, 1, -1)$	3	$(1, 2, 9)$
$(10, 7, 2)$	4	$(7, 2, 9)$
$(0, 11, 5)$	5	$(2, 5, 7)$
$(6, 11, 5)$	6	$(8, 5, 7)$
$(2, 5, 7)$	7	$(10, 7, 2)$
$(8, 5, 7)$	8	$(4, 7, 11)$
$(1, 2, 9)$	9	$(9, 9, 11)$
$(7, 2, 9)$	10	$(3, 9, -14)$
$(4, 7, 11)$	11	$(11, 11, -1)$
$(9, 9, 11)$	12	$(0, 11, 5)$
$(3, 9, -14)$	13	$(6, 11, 5)$
$(5, 14, -14)$	14	$(5, 14, -14)$

Tabelle 6.3: Die erzeugten Tupel (erste Spalte) in der Reihenfolge aus Tabelle 6.2, sowie in nach dem h-Rang sortierter Reihenfolge (dritte Spalte) und dem entsprechenden Tupelindex j.

haupt benötigt werden. Tabelle 6.2 listet den Index j der inneren Schleife, den korrespondierenden Suffix-Index $\mathsf{SA}_h[j] - h$ und die Bedingung zur Erzeugung des Tupels. Dabei muss $i > 0$ auf einen gültigen Index zeigen, dessen Suffix noch nicht eindeutig sortiert sein darf ($\mathsf{ISA}_h[i] > 0$). Ebenso haben wir den Suffix-Index $\mathsf{SA}_h[j]$ sowie die entsprechende Bedingung notiert. Die Bedingung entspricht hierbei der Negierung von Lemma 6. Die Bedingungen sind als Lazy Evaluation dargestellt, d. h. sobald ein Wert false ist, schreiben wir die Auswertung der weiteren Teilbedingungen nicht auf. In der ersten Iteration werden alle Suffixe außer S_0 und S_1 zum Erzeugen neuer Tupel verwendet. Wir erzeugen durch die zweite Bedingung darüber hinaus Tupel mit den Suffixen mit Index $14, 13$ und 5, da wir sie möglicherweise für die nächste Iteration zum weiteren Sortieren benötigen.

Die erzeugten Tupel und die daraus resultierende Sortierung sind in Tabelle 6.3 dargestellt. Dort sehen wir für die Suffix-Indizes $14, 13$ sowie 5, dass ihnen ihr ISA_2 Wert als $2h$-Rang und dessen Negierung als h-Rang zugewiesen wurden. Wichtig hierbei ist, dass für die zweite Komponente, den h-Rang, nur positive Werte gespeichert werden, damit die folgende Sortierung die Reihenfolge bereits sortierter Suffixe nicht verändert. Die dritte Komponente dient bei den drei bereits sortierten Suffixen nur als Platzhalter. Bei den übrigen Suffixen dient dieser Wert als finale Unterscheidungsmöglichkeit für die resultierende Sortierung. Da dabei auf Ungleichheit geprüft wird, stellen negative Werte, wie für die Suffix-Indizes $3, 11$ oder 12, kein Problem dar. Nach der Erzeugung und Sortierung der Tupel können wir im nächsten Schritt die neuen Ränge berechnen.

Tabelle 6.4 zeigt die Berechnung der neuen Ränge. Die ersten beiden Bedingungen prüfen, ob sich die zu vergleichenden Tupel in derselben Gruppe befin-

Indizes	h-rank$[j] >$ h-rank$[head]$	$2h$-rank$[j] \neq$ $2h$-rank$[head]$	h-rank$[j] =$ h-rank$[head]$ $+j - head$ (neue Gruppe)	h-rank$[j] =$ h-rank$[head]$ (Rang übernehmen)
$head = 2$ $j = 3$	$2 = 2$	$9 \neq -0$	h-rank[3] $= 2 + 3 - 2$ $= 3$	-
$head = 3$ $j = 4$	$2 < 3$	$9 = 9$	-	h-rank[4] $= h$-rank[3] $= 3$
$head = 3$ $j = 5$	$5 > 3$	-	-	-
$head = 5$ $j = 6$	$5 = 5$	$7 = 7$	-	h-rank[6] $= h$-rank[5] (Rang unverändert)
$head = 5$ $j = 7$	$7 > 5$	-	-	-
$head = 7$ $j = 8$	$7 = 7$	$11 \neq 2$	h-rank[8] $= 7 + 8 - 7$ $= 8$	-
$head = 8$ $j = 9$	$9 > 8$	-	-	-
$head = 9$ $j = 10$	$9 = 9$	$-14 \neq 11$	h-rank[10] $= 9 + 10 - 9$ $= 10$	-
$head = 10$ $j = 11$	$11 > 10$	-	-	-
$head = 11$ $j = 12$	$11 = 11$	$5 \neq -1$	h-rank[12] $= 11 + 12 - 11$ $= 12$	-
$head = 12$ $j = 13$	$11 < 12$	$5 = 5$	-	h-rank[13] $= h$-rank[12] $= 12$

Tabelle 6.4: Neuberechnung der Ränge mittels Tupeln aus Tabelle 6.3. Die erste Bedingung (zweite Spalte) prüft, ob eine neue h-Gruppe beginnt. Die zweite Bedingung (dritte Spalte) prüft, ob eine neue $2h$-Gruppe beginnt. Ist dies der Fall, muss der neue Rang berechnet werden. Falls die zweite Bedingung nicht erfüllt ist, so wird der Rang des Gruppenkopfes übernommen, in unserem Beispiel grün markiert. Sobald eine neue h- oder $2h$-Gruppe beginnt, muss der $head$-Zeiger auf den Anfang der neuen Gruppe zeigen. Ist eine der beiden Bedingungen erfüllt, so wird sie blau markiert.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA_4	14	13	12	1	7	2	8	10	4	9	3	11	0	6	5
ISA_4	-0	-1	-2	3	3	5	5	-7	-8	-9	-10	-11	12	12	-14

Tabelle 6.5: SA in sortierter Reihenfolge und ISA_4 (in SA_4-Reihenfolge) nach dem ersten Schleifendurchlauf.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	14	13	12	7	1	8	2	10	4	9	3	11	6	0	5
ISA	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Tabelle 6.6: Das finale SA nach einer weiteren Iteration. Alle drei unsortierten Gruppen wurden in ihrer Reihenfolge vertauscht. Alle Werte in ISA (in SA-Reihenfolge) wurden bereits negiert, sodass das SA hergeleitet werden kann.

den. Bei Ungleichheit der ersten Bedingung h-rank$[j] > h$-rank$[head]$ befinden sich die beiden Suffixe bereits in unterschiedlichen h-Gruppen. Bei der zweiten Bedingung $2h$-rank$[j] \neq 2h$-rank$[head]$ hingegen entsteht eine neue $2h$-Gruppe, da sie sich durch die Verdopplung des Suffixes zum ersten Mal unterscheiden. In diesem Fall muss der neue Rang berechnet werden. Dies geschiet in der 4. Spalte. Die blauen Markierungen in den Spalten 2 und 3 geben an, dass eine neue Gruppe beginnt, d. h. die Bedingung ist erfüllt. Wenn keine der beiden Bedingungen gilt, so befinden sich die Suffixe immer noch in derselben Gruppe. Dabei müssen wir den Rang des Gruppenkopfes übernehmen, da sich in der Zwischenzeit der Rang dieser Gruppe erhöht haben könnte. Um dies hervorzuheben, werden die Zuweisungen in der 5. Spalte grün markiert. Dies ist beispielsweise für die Gruppe $(1, 7)$ (Tupel 3 und 4) der Fall, wobei dem Index 1 im vorigen Durchlauf dieser Schleife der Rang 3 zugewiesen und somit der Rang 2 überschrieben wurde.

Zum Ende unserer Iteration übernehmen wir die finalen Werte für die h-Ränge in das ISA_4. Zuletzt markieren wir alle neuen, fertig sortierten Gruppen mittels Negierung und erhalten damit die finalen Werte dieser Iteration. Das Ergebnis findet sich in Tabelle 6.5. Da wir in dieser Iteration noch Tupel erzeugt haben und die Größe für die nächste Iteration der Anzahl erzeugter Tupel entspricht, müssen wir eine weitere Iteration durchführen. Dafür setzen wir die neue Größe auf die Anzahl unserer Tupel (identischer Wert) und wir verdoppeln h auf 4. Der Einfachheit halber überspringen wir diese Iteration jedoch, da die Gruppen $(1, 7), (2, 8)$ und $(0, 6)$ eindeutig sortiert werden. Unser finales Suffix-Array findet sich in Tabelle 6.6 wieder.

Dort sehen wir, dass die Reihenfolge für diese drei Gruppen jeweils umgedreht wurde. Wir würden daraufhin eine neue Iteration beginnen, aber bei der Erzeugung der Tupel merken, dass wir gar keine Tupel mehr erzeugen können, da entweder ungültige Indizes produziert werden oder die zu induzierenden Suffixe bereits sortiert sind. Damit wird die Größe auf 0 gesetzt und unsere Schleife ist fertig. Abschließend müssen wir aus dem ISA noch das SA berechnen, da wir

in jeder Iteration nicht in Tupel enthaltene Suffix-Indizes aus dem SA_h entfernt haben. Dabei können wir alle Ränge negieren, welche dann der Position des Suffixes im Suffix-Array entsprechen.

6.2.3 Parallelität und Implementierung

Im folgenden Abschnitt erläutern wir, wie drei Schritte des Algorithmus, die Markierung sortierter Gruppen, die Erzeugung der Tupel sowie die Aktualisierung der Ränge, parallelisiert werden können. Dabei gehen wir auch kurz auf die Implementierung in NVIDIA CUDA ein. Neben der Variante für Grafikkarten haben wir darüber hinaus eine sequentielle und eine parallele CPU Variante implementiert.

Für die Parallelisierung wird sowohl auf der CPU, als auch auf der GPU ein zusätzliches Hilfsarray für Zwischenberechnungen benötigt, welches wir mit aux bezeichnen.

Sortierte Gruppen markieren

Für die Markierung von sortierten Gruppen müssen wir ein Suffix sowohl mit seinem Vorgänger, als auch mit seinem Nachfolger vergleichen. Dafür setzen wir in einem ersten Durchlauf für jeden Index im Hilfsarray aux eine 1, wenn sich der Rang zum Rang des Vorgängers unterscheidet. Da der erste Index keinen Vorgänger besitzt, setzen wir für diesen immer eine 1. Im zweiten Durchlauf können wir den Wert in aux für Suffix j mit dem Wert seines Nachfolgers $j+1$ in SA abgleichen. Unterscheiden sich diese beiden Werte, so negieren wir den Rang des Suffixes j. Für den letzten Suffix müssen wir nur den Wert in aux betrachten, da dieser keinen Nachfolger besitzt. Ein Beispiel dafür zeigt Abb. 6.2. Die Operationen beider Durchläufe können parallel ausgeführt werden, weshalb wir diese beiden Durchläufe naiv auf zwei eigenständige Kernel-Methode ausgelagert haben.

Tupel erzeugen

Die Erzeugung der Tupel wird in drei Schritte aufgeteilt: Zunächst prüfen wir, für welche Indizes Tupel erstellt werden müssen. Dafür schreiben wir in unser Hilfsarray aux an Stelle j die Anzahl der erzeugten Tupel, wobei bis zu zwei Tupel (Suffix an Stelle j, zu sortierender Suffix i) erzeugt werden können. Wurde aux komplett befüllt, berechnen wir die exklusive Präfixsumme über alle betrachteten Suffixe dieser Iteration. Der neue Wert in aux gibt nun an, an welcher Stelle wir die durch Suffix j generierten Tupel einfügen. Im letzten Schritt können wir die Tupel parallel erzeugen und an die entsprechende Position einfügen. Für die Tupel verwalten wir zur einfacheren Nutzung drei Arrays: eines für den Index, eines für den h-Rang und das dritte Array für den $2h$-Rang. Um die Anzahl der erzeugten Tupel zu bestimmen, benötigen wir den letzten Index in aux. Da die Anzahl der durch den letzten Suffix erzeugten Tupel durch die

j	$\mathsf{ISA}_2[\mathsf{SA}_2[j]]$
0	0
1	1
2	2
3	2
4	2
5	5
6	5
7	7
8	7
9	9
10	9
11	11
12	11
13	11
14	14

$\rightarrow$

$\mathsf{aux}[j]$
1
1
1
0
0
1
0
1
0
1
0
1
0
0
1

$\rightarrow$

$\mathsf{ISA}_2[\mathsf{SA}_2[j]]$
-0
-1
2
2
2
5
5
7
7
9
9
11
11
11
-14

Abbildung 6.2: Beispiel zur parallelen Markierung von sortierten Gruppen. Wie beschrieben, werden zunächst die Indizes ermittelt, deren Rang sich von dem des Vorgängers unterscheidet. Unterscheidet dieser sich zudem noch vom Nachfolger, handelt es sich um eine fertig sortierte Gruppe.

Präfixsummenoperation überschrieben wird, berechnet sich die Gesamtzahl der Tupel aus dem Wert des letzten Indizes vor der Präfixsumme und dem Wert nach der Präfixsumme. Ein Beispiel hierfür ist in Abb. 6.3 abgebildet.

Da wir für den leichteren Zugriff auf die Ränge h- und $2h$-Ränge separat speichern, müssten wir ein Value-Paar aus Index und $2h$-Rang für alle Tupel generieren, um den Radixsort der CUB-Library nutzen zu können, welcher nur Key-Value-Paare (und einzelne Werte) sortieren kann. Wir haben uns stattdessen dafür entschieden, die $2h$-Ränge erst nach dem Radixsort in sortierter Reihenfolge zu erzeugen, da diese Ränge erst für das Aktualisieren der Ränge benötigt werden. Wir haben für das Befüllen des Hilfsarrays und für die Erzeugung der Tupel nach der Präfixsumme zwei unabhängige Kernel-Methoden geschrieben, da die Präfixsumme einen eigenen Kernel innerhalb der CUB-Library darstellt und der Aufruf von Kernel-Methoden nur durch Host-Methoden (von der CPU aus) möglich ist.

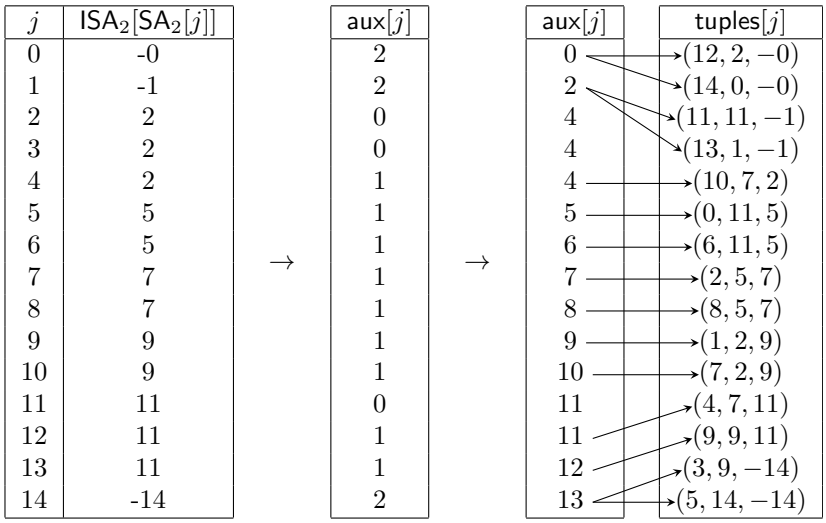


Abbildung 6.3: Beispiel zur parallelen Erzeugung der zu sortierenden Tupel. Zunächst werden dazu mit Hilfe des aux-Arrays die richtigen Position für die Tupel ermittelt und diese dann von den Threads an die jeweiligen Stellen in der Tupelliste geschrieben.

Ränge aktualisieren

Um die Aktualisierung der Ränge anhand der neuen Sortierung zu parallelisieren, gehen wir wie folgt vor: Zunächst ermitteln wir die Position der Tupel, an denen eine neue h-Gruppe beginnt. Dafür überprüfen wir für jedes Element in der sortierten Tupelliste, ob sich der Rang von dem seines Vorgängers unterscheidet. Ist dies der Fall, schreiben wir an der korrespondierenden Stelle im aux den Index des Suffixes, andernfalls eine 0. Haben wir alle Tupel überprüft, berechnen wir auf dem aux einen Prefixscan mit Maximums-Operator. Anschließend haben wir in diesem dann zu jedem Suffix die Anfangsposition der neuen h-Gruppe eingetragen. Wir prüfen hiernach mit denselben Bedingungen wie in der sequentiellen Variante (s. Algorithmus 6.2 Z.19 und 22), ob ein Tupel jetzt Kopf einer neuen $2h$-Gruppe ist. Ist dies der Fall, überschreiben wir den entsprechenden Wert in aux mit der Differenz aus der Position des Tupels in der Liste und dem aktuellen Wert in aux addiert mit dem alten h-rank, der sich aus dem Tupel entnehmen lässt. Schließlich berechnen wir erneut auf dem aux einen Prefixscan mit Maximums-Operator, erhalten dadurch die neuen $2h$-Ränge und können diese dann ins ISA übernehmen. Ein Beispiel für dieses Vorgehen, zeigt Abb. 6.4. In unserer Implementierung nutzen wir zum Befüllen des aux in beiden Fällen selbst implementierte Kernel Methoden, welche die vorher sortierte Liste der Tupel scannen. Für die Prefixscans nutzen wir dagegen, wie bereits bei der Tupelerzeugung, die in der CUB-Library implementierte Variante.

j	tuples$[j]$
0	$(14, 0, -0)$
1	$(13, 1, -1)$
2	$(12, 2, -0)$
3	$(1, 2, 9)$
4	$(7, 2, 9)$
5	$(2, 5, 7)$
6	$(8, 5, 7)$
7	$(10, 7, 2)$
8	$(4, 7, 11)$
9	$(9, 9, 11)$
10	$(3, 9, -14)$
11	$(11, 11, -1)$
12	$(0, 11, 5)$
13	$(6, 11, 5)$
14	$(5, 14, -14)$

$\rightarrow$

aux$[j]$
0
1
2
0
0
5
0
7
0
9
0
11
0
0
14

$\rightarrow$

aux$[j]$
0
1
2
2
2
5
5
7
7
9
9
11
11
11
14

$\rightarrow$

aux$[j]$
0
1
2
3
0
5
0
7
8
9
10
11
12
0
14

$\rightarrow$

aux$[j]$
0
1
2
3
3
5
5
7
8
9
10
11
12
12
14

Abbildung 6.4: Beispiel zur Berechnung der neuen Ränge im Parallelen. Dazu werden, wie beschrieben, die Positionen der Gruppenköpfe im aux berechnet, um anschließend ins ISA übertragen zu werden.

6.3 Prefix-Doubling

6.3.1 Einleitung

Viele der Algorithmen, mit denen Suffix-Arrays berechnet werden, eignen sich nicht dafür, auf externem Speicher zu arbeiten, da sie zu viele I/O-Operationen durchführen würden. Dies macht sie unbrauchbar für große Datenmengen, die nicht vollständig im Hauptspeicher verarbeitet werden können.

Dieses Kapitel präsentiert den in [18] beschriebenen **Doubling**-Algorithmus und seine verschiedenen Varianten. Er hat die Eigenschaft möglichst effizient bezüglich der I/O-Komplexität zu sein. Er eignet sich auch allgemein als ein Beispiel für einen puren Prefix-Doubling-Ansatz, weshalb wir zum Vergleich mit anderen Algorithmen eine Implementierung nutzen, die nur im Hauptspeicher arbeitet.

6.3.2 Grundlagen

Wir bauen zunächst auf den Prefix-Doubling-Definitionen in Abschnitt 5.2.1 auf.

Für jeden String T der Länge m mit beliebigem Alphabet lässt sich ein äquivalenter String T′ mit $\Sigma = [1, m]$ bilden, indem alle vorkommenden Zeichen aufsteigend sortiert, Duplikate entfernt, und die verbleibenden Zeichen durch einen aufsteigenden Zähler ersetzt werden.

Ein so **umbenannter** String hat die Eigenschaft, dass die Ordnung über dem neuen Alphabet identisch zu der über dem ursprünglichen ist und sich somit darauf dasselbe Suffix-Array ergibt. Ein dazu ähnliches Verfahren, die

lexikographische Umbenennung, wird Kern der vorgestellten Algorithmen sein und beruht darauf ganze (Teil-)Strings eines Textes durch aufsteigende Zähler zu ersetzen.

Für die Analyse der Zugriffsoperationen auf externen Speicher betrachten wir das I/O-Model [58]. Darin besteht ein Computersystem aus M Wörtern schnellen Hauptspeichers und langsameren externen Speicher der sich über I/O-Operationen in Blockgrößen B über D Platten erstreckt. Für Eingaben der Länge n nehmen wir eine Wortbreite $\geq \lceil \log n \rceil$ Bits an. Wir definieren folgende Kurzschreibweisen:

- $\text{scan}(x) = \lceil x/DB \rceil$ für sequentielles Lesen oder Schreiben von x Wörtern in externen Speicher.
- $\text{sort}(x) = \frac{2x}{DB} \left\lceil \log_{M/B} \frac{x}{M} \right\rceil$ für das I/O-effiziente Sortieren von x Wörtern unter Zuhilfenahme von externen Speicher.

6.3.3 Überblick

Wir betrachten zunächst in Abschnitt 6.3.4 den Doubling-Algorithmus [3][17]. Dieser hat eine I/O-Komplexität von $\mathcal{O}(\text{sort}(n \log \text{maxlcp}))$ und erlaubt es somit Strings der Länge 2^k in k Iterationen zu sortieren.

In den nachfolgenden Abschnitten werden wir anschließend systematisch Modifikationen einführen, die den Algorithmus bezüglich I/O-Komplexität und Rechenaufwand verbessern. Wir beginnen in Abschnitt 6.3.5 mit dem Konzept des *Pipelining*s, welches auf externen Algorithmen in der Regel eine I/O-Reduzierung um den Faktor 2 ermöglicht.

Abschnitt 6.3.6 beschreibt eine simple Methode, bereits bestimmte Suffixe aus nachfolgenden Iterationen des Doubling-Algorithmus zu entfernen (*Discarding*). Dies verbessert die I/O-Komplexität gegenüber Doubling zu $\mathcal{O}(\text{sort}(n \log \text{dps}))$ und ist besser als vorherige Discarding-Ansätze mit einer Komplexität von $\mathcal{O}(\text{sort}(n \log \text{dps})) + \text{scan}(n \log \text{maxlcp})$ [17].

Abschnitt 6.3.7 beschreibt eine Laufzeitverbesserung um einen konstanten Faktor, bei dem Strings der Länge a^k in k Iterationen sortiert werden (a-*Tupling*). Es stellen sich hierbei $a = 4$ und $a = 5$ als beste Ergebnisse heraus.

Abschnitt 6.3.8 fasst die einzelnen Varianten zusammen und gibt einen Überblick darüber welchen Nutzen sie für andere Algorithmen haben können.

6.3.4 Prefix-Doubling

```
1  def doubling(T):
2    S = [((T[i], T[i + 1]), i) for i in [0, n)]
3    for k in [1, ⌈log_2 n⌉]:
4      sort S
5      P = name(S)
6      if names in P are unique:
7        return [i for (c, i) in P]
8      sort P by (i mod 2^k, i ÷ 2^k)
9      S = []
10     for each (d, i) = P[j]:
11       if (d', i') = P[j + 1] exists and i + 2^k == i':
12         append ((d, d'), i) to S
13       else:
14         append ((d, $), i) to S
15 def name(S):
16   q = 0
17   r = 0
18   (l, l') = ($, $)
19   result = []
20   for ((c, c'), i) in S:
21     q = q + 1
22     if (c, c') != (l, l'):
23       r = q
24       (l, l') = (c, c')
25     append (r, i) to result
26   return result
```

Algorithmus 6.3: Doubling

Wie in Abschnitt 5.2.1 beschrieben, sortieren wir alle T_i anhand eines Präfixes von 2^k Zeichen. Wir vergleichen die $\mathsf{T}[i, i+2^k)$ Strings jedoch nicht zeichenweise, da dies den Rechenaufwand mit jedem Iterationsschritt verdoppeln würde. Stattdessen arbeitet der Algorithmus auf einer Sequenz S von Tupeln $((c, c'), i)$, bei der c und c' eindeutige **Namen** für $\mathsf{T}[i, i + 2^{k-1})$ und $\mathsf{T}[i + 2^{k-1}, i + 2^k)$ sind. Eine naive Implementierung dieses Ansatzes findet sich unter `doubling()` in Algorithmus 6.3.

Wir betrachten hierfür das Beispiel in Abbildung 6.5. Zu Beginn der ersten Iteration entsprechen c und c' benachbarten Zeichen im Eingabestring. S wird gemäß der (c, c')-Tupel lexikografisch sortiert (erster *sort*-Schritt im Beispiel), und anschließend gemäß ihrer neuen Position in S **lexikographisch umbenannt** (*name* im Beispiel).

Die Umbenennung erfolgt, indem die Tupel durch Zeichen d aus $[1, n]$ ersetzt werden, die derselben aufsteigenden Ordnung folgen. Siehe hierzu die `name()`-

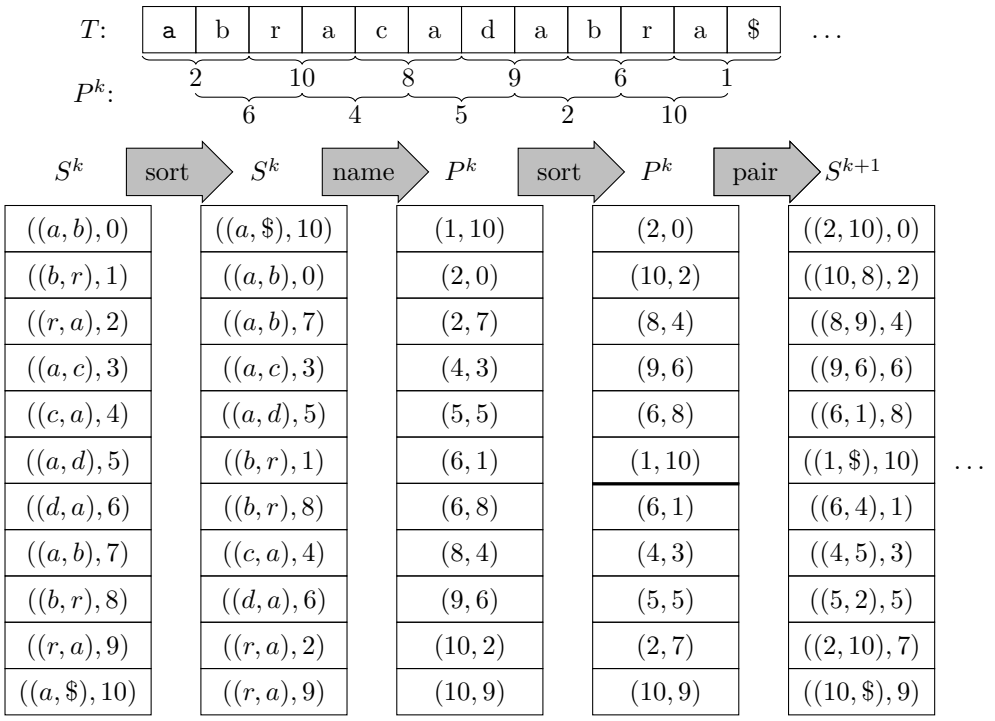


Abbildung 6.5: `doubling()` für $\mathsf{T} = abracadabra$ und $k = 1$

Funktion in Algorithmus 6.3. Die erhaltenen Tupel (d, i) werden in eine Sequenz P eingefügt und haben die Eigenschaft, dass d jeweils ein eindeutiger Name für die Zeichenfolge $\mathsf{T}[i, i+2^k)$ ist. Dies lässt sich gut an der Abbildung veranschaulichen, in der die Namen d aller $(d, i) \in P^k$ zusätzlich an ihren Textpositionen i unter der Eingabe T eingezeichnet sind. So haben zum Beispiel die Länge-2-Teilstrings an Postion 2 und 9 beide den Namen 10.

Die P-Sequenz wird anschließend gemäß des Tupels $(i \mod 2^k, i \div 2^k)$ lexikografisch sortiert (zweiter *sort*-Schritt im Beispiel). Dies führt dazu, dass Namen für direkt benachbarte Teilstrings in P nebeneinander liegen und Namen für überlappende Teilstrings in separaten Abschnitten von P liegen. Dies wird in der Abbildung durch die schwarze Trennlinie im zweiten P^k-Array deutlich.

P wird nun ähnlich zur Eingabezeichenfolge betrachtet, indem aus benachbarten Namen d und d' Tupel $((d, d'), i)$ gebildet werden und diese als Sequenz S für die nächste Iteration genutzt werden (*pair* im Beispiel).

Da immer nur zwei Namen für benachbarte Teilstrings zu neuen Namen zusammengefasst werden und der Algorithmus mit allen Zeichen und Textpositionen der Eingabe beginnt, verdoppelt sich somit in jeder Iteration die Länge der repräsentierten Präfixe des Suffix-Arrays. Da die Ordnung der darunterliegenden Strings in den neuen Namen erhalten bleibt, lässt sich aus S-Sequenzen von späteren Iterationen das korrekte Suffix-Array auslesen, sobald alle Elemente in ihnen eindeutig sind. Und da in jeder Iteration immer nur Paare von Namen betrachtet werden, steigen die Sortierkosten von S nicht mit jeder Iteration.

Wir gehen nun davon aus, dass S und P in externem Speicher liegen. Gemäß unserer Definition zur I/O-Komplexität, führt der Algorithmus pro Iteration einen konstante Anzahl von *Scanning*- und *Sort*-Operationen über n Elemente durch und hat eine obere Iterationsschranke beim Logarithmus des längsten gemeinsamen Präfixes. Es gilt somit:

Theorem 1. *Der Doubling-Algorithmus berechnet ein Suffix-Array in* $\mathcal{O}(\mathit{sort}(n)\lceil \log \mathit{maxlcp} \rceil)$ *I/O-Operationen.*

6.3.5 Pipelining

Als erste signifikante Verbesserung des Doubling-Algorithmus beobachten wir, dass sich manche Operation auf P und S beschleunigen lassen, indem sie temporäre Daten, anstatt sie erst in eine Sequenz zu schreiben und anschließend wieder auszulesen, direkt aneinander weitergeben (**Pipelining**).

Wir betrachten hierzu wieder Algorithmus 6.3. Anstatt die Tupel in Zeile (2) zunächst vollständig in S zu schreiben, können sie direkt an die Sortierfunktion in Zeile (4) übergeben werden. Ebenso können die so sortierten Tupel direkt in die Benennungsfunktion in Zeile (5), und von dort direkt in die Sortierfunktion in Zeile (8) geleitet werden, da sich die Benennungsfunktion nur jeweils das vorherige Tupel merken muss. Dasselbe gilt für die Paarbildung in den Zeilen (10)–(14), in der auch nur jeweils zwei nachfolgende Elemente betrachtet werden müssen. Die in Zeile (12) und (14) bestimmten Tupel können schließlich wie in Zeile (2) direkt in die Sortierfunktion (4) der nächsten Iteration geleitet werden.

Wir analysieren diese Modifikation nun genauer in Hinblick auf die I/O-Komplexität. Wir nehmen zunächst an, dass sich eine Sequenz aus m Wörtern mit einem geeigneten Sortieralgorithmus in $\approx 4m/DB$ I/O-Operationen und mit zwei Durchläufen sortieren lässt, falls $x \ll M^2/DB$ und $M \gg DB$ [1]. Die folgende Analyse ist dabei [18] entnommen.

Ohne Pipelining benötigt die Sortierung von n Tripeln in Zeile (4) somit $12m/DB$ I/Os, und das Lesen der Tripel und Schreiben der Paare in Zeile (5) $5m/DB$ I/Os. Der Test auf Eindeutigkeit in Zeile (6) kann während der Benennung durchgeführt werden und benötigt keine zusätzlichen I/Os. Das Sortieren der Tupel in Zeile (8) kostet $8m/DB$ I/Os, und das Lesen der Paare und Schreiben der Tripel in Zeilen (10)–(14) wiederum $5m/DB$ I/Os. Insgesamt erhalten wir somit Kosten von $(12 + 5 + 8 + 5)n/DB = 30n/DB$ I/Os.

Wir betrachten nun die I/Os für die Pipelining-Variante des Algorithmus. Zunächst können die S-Tupel direkt in die erste Phase des Sortieralgorithmus (Zeile (4)) geleitet werden, welche dabei mit $3n/DB$ I/Os in externen Speicher schreibt. Die zweite Phase liest die Tripel sortiert in ebenfalls $3n/DB$ I/Os aus, und leitet sie ohne weitere I/Os durch die Benennungsfunktion und in die erste Phase der Sortierung in Zeile (8). Diese schreibt mit $2n/DB$ I/Os, und liest sie in der zweiten Phase mit $2n/DB$ I/Os sortiert aus, leitet sie ohne zusätzliche I/Os durch die Paarbildung, und übergibt sie an die erste Phase der S-Sortierung der nächsten Iteration.

Insgesamt wurden somit innerhalb der Iterationen alle *Scan*-Operationen von S und P eliminiert und nur die Hälfte der I/Os von *Sort*-Operationen beibehalten, womit sich insgesamt eine Reduzierung auf $(3+3+2+2)n/DB = 10n/DB$ I/Os ergibt. Dies erlaubt eine genauere Spezifizierung der I/O-Komplexität des Algorithmus:

Theorem 2. *Der Doubling-Algorithmus mit Pipelining berechnet ein Suffix-Array in* $sort(5n)\lceil \log maxlcp \rceil + \mathcal{O}(sort(n))$ *I/O-Operationen.*

Der $\mathcal{O}(\text{sort}(n))$ Anteil steht hierbei für alle einmalig durchzuführenden Berechnungen außerhalb der Schleife.

6.3.6 Discarding

Für die nächste Verbesserung des Algorithmus beobachten wir Folgendes. Sei c_i^k der lexikographische Name für ein $\mathsf{T}[i, i + 2^k)$ in Iteration k. Falls c_i^k einzigartig in S ist, ist auch der zugehörige Teilstring ein einzigartiger Präfix. Es gilt somit $c_i^k = c_i^h$ für alle $h > k$, da die zusätzlichen Zeichen in $\mathsf{T}[i, i + 2^h)$ nichts mehr an der lexikografischen Sortierung und somit der Position in S ändern würden. Die Idee des **Discardings** ist es nun, in jeder Iteration Tupel mit einzigartigen Namen aus S zu entfernen, um somit nachfolgende I/Os zu verringern.

Ein Problem hierbei ist, dass sich durch das Entfernen von Elementen aus S die Benennungsfunktion `name()` anders verhalten würde, da ein Name durch die Anzahl vorheriger Tupel bestimmt wird. Dies macht eine neue Funktion `name2()` erforderlich, die bei der Benennung von allen $(c, c') \in S$ Namen relativ zum

```
def doubling_discarding(T):
  S = [((T[i], T[i + 1]), i) for i in [0, n)]
  sort S
  U = name(S) # undiscarded
  P = []      # partially discarded
  F = []      # fully discarded
  for k in [1, ⌈log_2 n⌉]:
    mark unique names in U
    sort U by (i mod 2^k, i ÷ 2^k)
    merge P into U
    P = []
    S = []
    count = 0
    for each (d, i) = U[j]:
      if (d, i) is unique:
        if count < 2:
          append (d, i) to F
        else:
          append (d, i) to P
        count = 0
      else:
        if (d', i') = U[j + 1] exists and i + 2^k == i':
          append ((d, d'), i) to S
        else:
          append ((d, $), i) to S
        count = count + 1
    if S is empty:
      sort F
      return [i for (d, i) in F]
    sort S
    U = name2(S)
def name2(S):
  q = 0
  r = 0
  (l, l') = ($, $)
  result = []
  for ((c, c'), i) in S:
    if c != l:
      (q, r) = (0, 0)
      (l, l') = (c, c')
    elif c' != l':
      r = q
      l' = c'
    append (c + r, i) to result
    q = q + 1
  return result
```

Algorithmus 6.4: Doubling+Discarding

numerischen Wert von c bildet. Siehe Algorithmus 6.4 für eine entsprechende Implementierung.

Es dürfen außerdem nicht alle einzigartigen c_i^k direkt aus S entfernt werden, da sie gegebenenfalls noch mit benachbarten Namen gepaart werden müssen. Hierzu betrachten wir die Situation für c_i^{k+1}, und unterscheiden zwei Fälle: Falls einer der vorherigen Namen $c_{i-2^k}^k$ oder $c_{i-2^{k+1}}^k$ einzigartig ist, gilt dies auch für ihre Zusammenführung in $c_{i-2^{k+1}}^{k+1}$. Somit ist c_i^{k+1} nicht notwendig, um $c_{i-2^{k+1}}^{k+2}$ als einzigartig zu identifizieren und kann aus S entfernt werden. Wir bezeichnen dies als *Fully-Discarded*. Ansonsten ist $c_{i-2^{k+1}}^{k+1}$ nicht einzigartig und wir benötigen c_i^{k+1}, um $c_{i-2^{k+1}}^{k+2}$ zu bestimmen. Wir entfernen c_i^k weiterhin aus S, aber übernehmen es zunächst noch als $d = c_i^{k+1}$ in die Liste der sortierten (d, i) Paare der nächsten Iteration. Wir bezeichnen dies als *Partially-Discarded*.

Alle Tupel, die vollständig aus S entfernt wurden, werden in einer zusätzlichen Sequenz F gesammelt und der Algorithmus endet sobald S leer ist. Wenn dies der Fall ist, enthält F einen eindeutigen Namen für jede Textposition und es lässt sich nach einer finalen lexikografischen Sortierung das Suffix-Array auslesen. Siehe hierzu `doubling_discarding()` in Algorithmus 6.4.

Wir betrachten nun wieder die I/O-Komplexität. Beim naiven Doubling-Algorithmus sortieren und verdoppeln wir so lange alle Präfixe, bis sie paarweise verschieden sind. Jeder einzelne Präfix durchläuft somit immer genau log maxlcp Iterationen. Mit Discarding hingegen wird ein einzigartiger Präfix so früh wie möglich aus der Iteration entfernt. Somit gilt für alle S_i, dass ein Präfix maximal $\mathrm{dps}(i)$ Iterationen durchläuft.

Wir stellen auch fest, dass Discarding kompatibel mit Pipelining ist. Der Algorithmus nutzt zwar mehr (externe) Sequenzen von Tupeln, aber es werden weiterhin in k Iterationen jeweils nur die zwei *Sort*-Operationen des Basisalgorithmus angewandt und alle weiteren Operationen scannen die einzelnen Elemente jeweils nur in einem Durchlauf. Dies ermöglicht somit das direkte Weiterleiten von Zwischenergebnissen in Form von Pipelining und führt zu folgender Analyse der Gesamtkomplexität:

Theorem 3. *Der Doubling-Algorithmus mit Pipelining und Discarding berechnet ein Suffix-Array in* $\mathit{sort}(5n)\lceil \log \mathit{dps} \rceil + \mathcal{O}(\mathit{sort}(n))$ *I/O-Operationen.*

6.3.7 A-Tupling

Als letzte Verbesserung lässt sich der Doubling-Algorithmus schließlich darauf generalisieren, in jeder Iteration lexikografische Namen für a^k Zeichen, anstatt 2^k zu bestimmen. Hierzu werden jeweils a benachbarte Namen in einem Tupel gesammelt. Algorithmus 6.5 zeigt den so modifizierten Code ohne Discarding oder Pipelining.

Dies wirkt sich in zwei Aspekten auf die I/O-Komplexität aus. Anstatt $\mathrm{sort}(5n)$ Sortieroperationen pro Iteration, erhöhen sich die Kosten auf $\mathrm{sort}((a+3)n)$. Andererseits verringert sich die Anzahl der Iterationen auf $\log_a n$. Setzt

```
def a_tupling(T):
  S = [((T[i], T[i + 1], ..., T[i + a - 1]), i) for i in [0, n)]
  for k in [1, ⌈log_a n⌉]:
    sort S
    P = name(S)
    if names in P are unique:
      return [i for (c, i) in P]
    sort P by (i mod a^k, i ÷ a^k)
    S = []
    for each (d, i) = P[j]:
      l = i
      T = [d]
      for q in [1, a − 1]:
        if (d', i') = P[j + q] exists and l + a^k == i':
          append d' to T
        else:
          append $ to T
          l = i'
      append (T, i) to S
```

Algorithmus 6.5: a-Tupling

a	2	3	4	5	6
$(a+3)/\log a$	5,0	3,78	3,50	3,45	3,56

Tabelle 6.7: I/O-Anforderungen für verschiedenen Varianten von a-Tupling

man beide Werte in Relation zueinander, so erhält man einen Faktor von $\frac{a+3}{\log a}$ für die gesamten I/O-Kosten des Algorithmus.

Werten wir dies für unterschiedliche Werte von a aus, so ist $a = 5$ optimal (Siehe Tabelle 6.7). In der Praxis eignet sich jedoch $a = 4$ mehr, da die I/O-Kosten nur $1,5\%$ schlechter sind und sich der Wert als Zweierpotenz besser für Berechnungen und der Implementierung eignet.

Der a-Tupling Ansatz lässt sich ohne signifikante Anpassungen in der Discarding und Pipelining-Variante verwenden, womit sich die I/O-Komplexitäten direkt ableiten lassen:

Theorem 4. *Der Doubling-Algorithmus mit Pipelining und a-Tupling berechnet ein Suffix-Array in* $sort(\frac{a+3}{\log a}n)\lceil \log maxlcp \rceil + \mathcal{O}(sort(n))$ *I/O-Operationen.*

Theorem 5. *Der Doubling-Algorithmus mit Pipelining, a-Tupling und Discarding berechnet ein Suffix-Array in* $sort(\frac{a+3}{\log a}n)\lceil \log dps \rceil + \mathcal{O}(sort(n))$ *I/O-Operationen.*

6.3.8 Zusammenfassung und Ausblick

In den vorhergehenden Abschnitten haben wir den Doubling-Algorithmus betrachtet und die darauf aufbauenden Modifikationen Pipelining, Discarding und a-Tupling. Jede dieser drei Varianten verbessert die I/O-Komplexität des Basisalgorithmus und lässt sich mit den anderen Kombinieren, wodurch sich ein guter externer Suffix-Array-Algorithmus definieren lässt, der alle Varianten miteinander vereint.

Das Pipelining beschreibt insbesondere eine allgemeine Technik, die auch für viele andere Algorithmen auf externen Speicher nützlich ist. Ebenso existieren weiterführende Algorithmen, die auf Präfix Doubling aufbauen. Ein Beispiel hierfür ist der DC3 und DCX Algorithmus [18]. Siehe auch Abschnitt 6.4.

6.3.9 Implementierung

Die Implementierung als Teil des Benchmarking-Frameworks erfolgt aus Gründen der Vergleichbarkeit nicht im externen Speicher. Stattdessen sind die in den Algorithmen benutzten P-, S-, U- und F-Arrays grundsätzlich mit der vom Framework bereitgestellten `container` Klasse implementiert.

Dies hat auch Auswirkungen auf den Sortieralgorithmus und die Anwendbarkeit des Pipelinings. Da nun davon ausgegangen wird, dass jedes Array immer in den Hauptspeicher passt, ist man somit nicht mehr auf einen effizienten externen Mergesort angewiesen und kann einen beliebigen In-Memory-Sortierer benutzen.

Dies verhindert jedoch ein direktes Anwenden des Pipelining-Ansatzes im vorgestellen Sinne, da die Sortierschritte in den Algorithmen nicht mehr als Stream-akzeptierend bzw. -erzeugend betrachtet werden können. Aufgrund des Fehlens von IO-Operationen auf externen Speicher hat dies allerdings auch geringe Auswirkungen. In Abschnitt 6.3.11 nutzen wir stattdessen die Pipelining-Idee dafür, den Speicherverbrauch der Arrays stark zu senken.

Umsetzung

Die Implementierung besteht aus einer C++-Klasse, die über dem benutzten `sa_index`-Typ und der Tupelgröße `a` gemäß des a-Tuplings templatisiert ist.

Diese stellt die statischen Memberfunktionen `doubling()` und `doubling_discarding()` bereit, die den normalen Doubling-Algorithmus bzw. seine Discarding-Variante abhängig von `a` implementieren.

Sie nutzen dabei das vom Framework vorgegebene Interface, bei dem die Eingabe als ein `string_span` repräsentiert wird, der auf einen Bereich im Speicher zeigt und das Ausgabe-Array `SA` als ein `util::span<sa_index>` repräsentiert wird, in das geschrieben werden muss.

Die Arrays enthalten Tupel der Form (c, i) und $((c_1, \ldots, c_a), i)$, wobei c ein Präfixname ist, und i ein Textindex. Da sowohl Namen als auch Indexe in $[0, n)$ liegen, passen alle Werte in eine Variable vom Typ `sa_index` (siehe

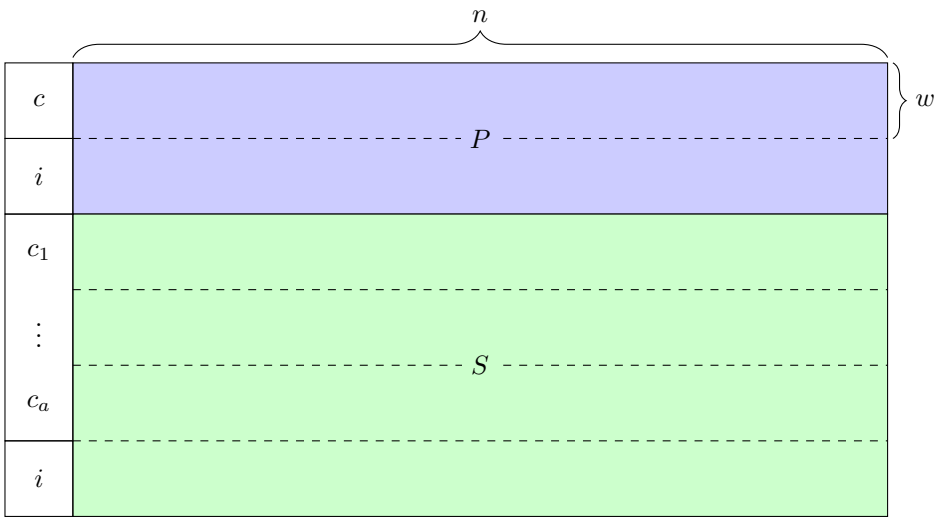


Abbildung 6.6: Doubling-Speicherlayout

Abschnitt 3.3). Die Implementierung repräsentiert deshalb alle Tupel als Arrays fester Länge `std::array<sa_index, 2>` bzw. `std::array<sa_index, a + 1>`.

Sei $w =$ `sizeof(sa_index)`. Bei einer direkten Umsetzung der Algorithmen in C++ liegt der zusätzliche Speicherverbrauch des normalen Doublings somit bei $(a+3)nw$ Bytes, und der der Discarding-Variante bei $(a+7)nw$ Bytes. Siehe Abbildung 6.6 und Abbildung 6.7.

6.3.10 Beispiel

Wir stellen nun die Funktionsweise des Discarding-Algorithmus ohne Optimierungen anhand des Beispieltextes `caabaccaabacaa$` vor:

1. Erzeuge anfängliche Zeichenpaar-Textposition-Tupel. Sie entsprechen Präfixen der Länge 2.

S	0	1	2	3	4	5	6	7	8	9	10	11	12	13
c_0	c	a	a	b	a	c	c	a	a	b	a	c	a	a
c_1	a	a	b	a	c	c	a	a	b	a	c	a	a	\$
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13

2. Sortiere S lexikografisch.

S	0	1	2	3	4	5	6	7	8	9	10	11	12	13
c_0	a	a	a	a	a	a	a	a	b	b	c	c	c	c
c_1	\$	a	a	a	b	b	c	c	a	a	a	a	a	c
i	13	1	7	12	2	8	4	10	3	9	0	6	11	5

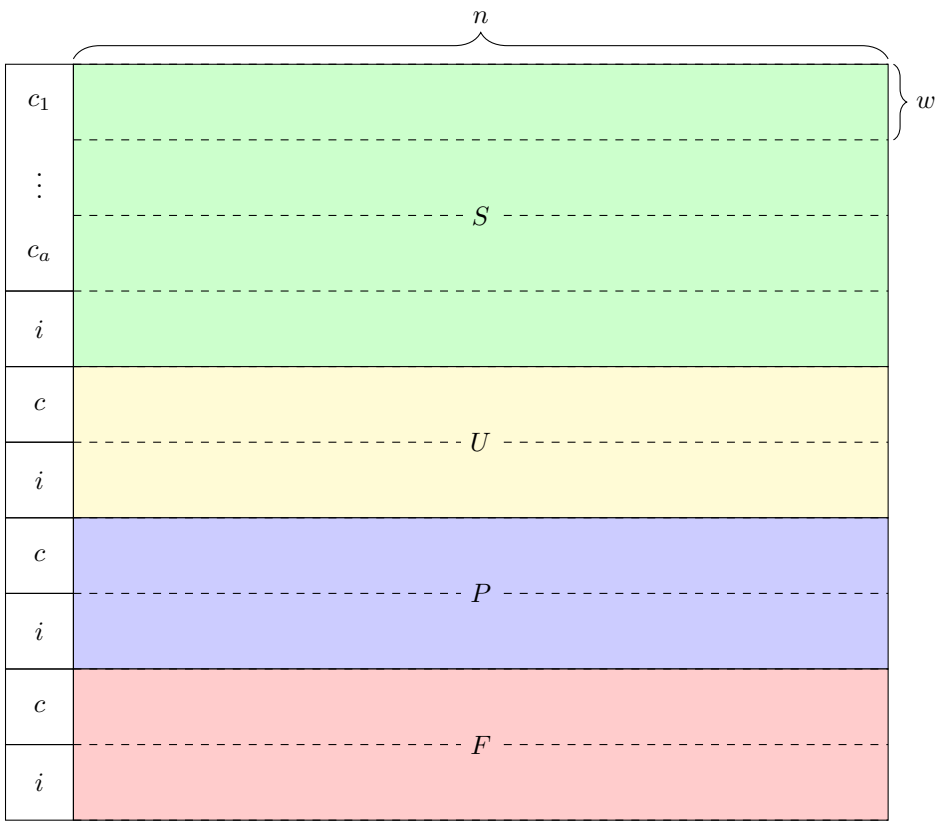


Abbildung 6.7: Discarding-Speicherlayout

3. Benenne die Paare in S lexikografisch um ($U = \text{name}(S)$).

U	0	1	2	3	4	5	6	7	8	9	10	11	12	13
c	1	2	2	2	5	5	7	7	9	9	11	11	11	14
i	13	1	7	12	2	8	4	10	3	9	0	6	11	5

4. Die anfängliche Zuordnung der Namen zu Präfixen ist somit:

Name	Präfix	Name	Präfix
1	$a\$$	9	ba
2	aa	11	ca
5	ab	14	cc
7	ac		

5. Initialisiere P und F als leere Sequenzen.

6. Iteriere bis zu $\lceil \log_2 n \rceil = 4$ mal.

7. Beginne Iteration $k = 1$.

8. Markiere alle einzigartigen Namen in U (grün in der Tabelle).

U	0	1	2	3	4	5	6	7	8	9	10	11	12	13
c	1	2	2	2	5	5	7	7	9	9	11	11	11	14
i	13	1	7	12	2	8	4	10	3	9	0	6	11	5
$Uniq.$	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

9. Merge P in die Sequenz U. Keine Änderung, da P leer ist.

10. Sortiere U anhand $(i \mod 2^k, i \div 2^k)$.

U	0	1	2	3	4	5	6	7	8	9	10	11	12	13
c	11	5	7	11	5	7	2	2	9	14	2	9	11	1
i	0	2	4	6	8	10	12	1	3	5	7	9	11	13
$Uniq.$	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓

11. Iteriere durch U.

    - Falls der Name einzigartig ist und einer der beiden Vorgängernamen einzigartig ist, füge ihn zu F hinzu.
    - Falls der Name einzigartig ist und keiner der beiden Vorgängernamen einzigartig ist, füge ihn zu P hinzu.
    - Sonst bilde aus dem Namen und seinem Nachfolger ein Paar, und füge es zu S hinzu.

U_c	U_i	U_{uniq}	S	P	F
11	0	✗	$((11,5),0)$		
5	2	✗	$((5,7),2)$		
7	4	✗	$((7,11),4)$		
11	6	✗	$((11,5),6)$		
5	8	✗	$((5,7),8)$		
7	10	✗	$((7,2),10)$		
2	12	✗	$((2,0),12)$		
2	1	✗	$((2,9),1)$		
9	3	✗	$((9,14),3)$		
14	5	✓		$(14,5)$	
2	7	✗	$((2,9),7)$		
9	9	✗	$((9,11),9)$		
11	11	✗	$((11,1),11)$		
1	13	✓		$(1,13)$	

12. Überprüfe ob S leer ist. ⇒ Nein, mache weiter.

13. Sortiere S lexikografisch.

S	0	1	2	3	4	5	6	7	8	9	10	11
c_0	2	2	2	5	5	7	7	9	9	11	11	11
c_1	0	9	9	7	7	2	11	11	14	1	5	5
i	12	1	7	2	8	10	4	9	3	11	0	6

14. Benenne die Paare in S lexikografisch um.

U	0	1	2	3	4	5	6	7	8	9	10	11
c	2	3	3	5	5	7	8	9	10	11	12	12
i	12	1	7	2	8	10	4	9	3	11	0	6

15. Beginne Iteration $k = 2$.

16. Markiere alle einzigartigen Namen in U.

U	0	1	2	3	4	5	6	7	8	9	10	11
c	2	3	3	5	5	7	8	9	10	11	12	12
i	12	1	7	2	8	10	4	9	3	11	0	6
$Uniq.$	✓	✗	✗	✗	✗	✓	✓	✓	✓	✓	✗	✗

17. Merge P in die Sequenz U und setze P zurück. Die Tupel sind dabei immer einzigartig.

P	0	1
c	14	1
i	5	13

$\Rightarrow$

U	0	1	2	3	4	5	6	7	8	9	10	11	12	13
c	1	2	3	3	5	5	7	8	9	10	11	12	12	14
i	13	12	1	7	2	8	10	4	9	3	11	0	6	5
$Uniq.$	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓	✓	✗	✗	✓

18. Die Zuordnung der Namen zu Präfixen für diesen Schritt ist somit:

Name	Präfix	Name	Präfix
1	*a*$$$	9	*baca*
2	*aa*$$	10	*bacc*
3	*aaba*	11	*caa*$
5	*abac*	12	*caab*
7	*acaa*	14	*ccaa*
8	*acca*		

19. Sortiere U anhand $(i \mod 2^k, i \div 2^k)$.

U	0	1	2	3	4	5	6	7	8	9	10	11	12	13
c	12	8	5	2	3	14	9	1	5	12	7	10	3	11
i	0	4	8	12	1	5	9	13	2	6	10	3	7	11
$Uniq.$	✗	✓	✗	✓	✗	✓	✓	✓	✗	✗	✓	✓	✗	✓

20. Iteriere durch U.

U_c	U_i	U_{uniq}	S	P	F
12	0	✗	$((12,8),0)$		
8	4	✓			$(8,4)$
5	8	✗	$((5,2),8)$		
2	12	✓			$(2,12)$
3	1	✗	$((3,14),1)$		
14	5	✓			$(14,5)$
9	9	✓			$(9,9)$
1	13	✓			$(1,13)$
5	2	✗	$((5,12),2)$		
12	6	✗	$((12,7),6)$		
7	10	✓		$(7,10)$	
10	3	✓			$(10,3)$
3	7	✗	$((3,11),7)$		
11	11	✓			$(11,11)$

21. F nach diesem Schritt:

F	0	1	2	3	4	5	6
c	8	2	14	9	1	10	11
i	4	12	5	9	13	3	11

22. Überprüfe, ob S leer ist. $\Rightarrow$ Nein, mache weiter.

23. Sortiere S lexikografisch.

S	0	1	2	3	4	5
c_0	3	3	5	5	12	12
c_1	11	14	2	12	7	8
i	7	1	8	2	6	0

24. Benenne die Paare in S lexikografisch um.

U	0	1	2	3	4	5
c	3	4	5	6	12	13
i	7	1	8	2	6	0

25. Beginne Iteration $k = 3$.

26. Markiere alle einzigartigen Namen in U.

U	0	1	2	3	4	5
c	3	4	5	6	12	13
i	7	1	8	2	6	0
$Uniq.$	✓	✓	✓	✓	✓	✓

27. Merge P in die Sequenz U und setze P zurück.

P	0
c	7
i	10

$\Rightarrow$

U	0	1	2	3	4	5	6
c	3	4	5	6	7	12	13
i	7	1	8	2	10	6	0
$Uniq.$	✓	✓	✓	✓	✓	✓	✓

28. Die Zuordnung der Namen zu Präfixen für diesen Schritt ist somit:

Name	Präfix	Name	Präfix
3	*aabacaa*\$	7	*acaa*\$\$\$\$
4	*aabaccaa*	12	*caabacaa*
5	*abacaa*\$\$	13	*caabacca*
6	*abaccaab*		

29. Sortiere U anhand $(i \mod 2^k, i \div 2^k)$.

U	0	1	2	3	4	5	6
c	13	5	4	6	7	12	3
i	0	8	1	2	10	6	7
$Uniq.$	✓	✓	✓	✓	✓	✓	✓

30. Iteriere durch U.

U_c	U_i	U_{uniq}	S	P	F
13	0	✓			$(13, 0)$
5	8	✓			$(5, 8)$
4	1	✓			$(4, 1)$
6	2	✓			$(6, 2)$
7	10	✓			$(7, 10)$
12	6	✓			$(12, 6)$
3	7	✓			$(3, 7)$

31. F nach diesem Schritt:

F	0	1	2	3	4	5	6	7	8	9	10	11	12	13
c	8	2	14	9	1	10	11	13	5	4	6	7	12	3
i	4	12	5	9	13	3	11	0	8	1	2	10	6	7

32. Überprüfe, ob S leer ist. ⇒ Ja, beende den Algorithmus.

33. Sortiere F lexikografisch.

F	0	1	2	3	4	5	6	7	8	9	10	11	12	13
c	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i	13	12	7	1	8	2	10	4	9	3	11	6	0	5

34. Gib finales Suffix Array aus:

Index	Suffix
13	*a*
12	*aa*
7	*aabacaa*
1	*aabaccaabacaa*
8	*abacaa*
2	*abaccaabacaa*
10	*acaa*
4	*accaabacaa*
9	*bacaa*
3	*baccaabacaa*
11	*caa*
6	*caabacaa*
0	*caabaccaabacaa*
5	*ccaabacaa*

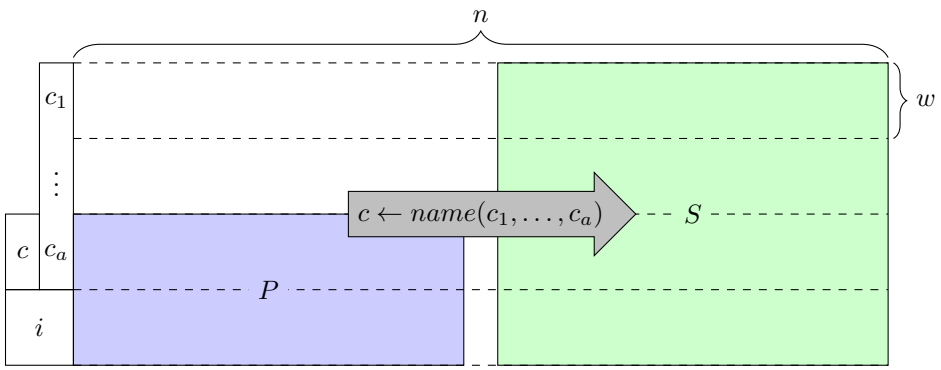


Abbildung 6.8: Doubling-Speicherlayout bei Array-Überlagerung

6.3.11 Optimierungen

Es wurden eine Reihe von Optimierungen durchgeführt, um den Speicherverbrauch und die Laufzeitperformance der anfänglichen Implementierung zu verbessern:

Array-Überlagerung beim Doubling

Die ausschlaggebendste Speicheroptimierung basiert auf der Beobachtung, dass die Algorithmen nie gleichzeitig auf alle Elemente der benutzen Arrays zugreifen müssen.

Betrachten wir hierzu zunächst P und S des einfachen Doubling-Algorithmus (Listing 6.3). Aus den Überlegungen zum Pipelining wissen wir, dass das Erzeugen der umbenannten Tupel in Zeile 5 online während der Iteration über die Elemente von S geschehen kann. Dasselbe gilt umgekehrt für die Paarbildung in Zeile 11–14, bei der wir für das i-te Paar nur jeweils den i-ten und $i+1$-ten Eintrag aus P benötigen.

Wir beobachteten außerdem, dass sich die Tupel $(c, i) \in P$ vollständig als Tupel $((c_1, \ldots, c_a), i) \in S$ darstellen lassen, indem wir zB. c immer durch c_1 repräsentieren.

Wir repräsentieren P und S nun nicht mehr durch zwei getrennte Arrays der Länge $(a+1)nw$ für S und $2nw$ für P, sondern nutzen ein einzelnes Array SP der Größe $(a+1)nw$ bei dem ein Eintrag *entweder* ein Tupel $(c, i) \in P$ *oder* ein Tupel $((c_1, \ldots, c_a), i) \in S$ enthält. Die Umbenennung bzw. Paarbildung im Algorithmus iteriert somit durch ein Array aus anfänglich S bzw. P Elementen und ersetzt dabei sequentiell jeden Eintrag durch das errechnete P bzw. S Element. Siehe Abbildung 6.8.

Array-Überlagerung beim Discarding

Die vorherige Optimierung lässt sich zunächst direkt auf die S und U Arrays der Discaring-Variante 6.4 anwenden, um den Speicherverbrauch auf $(a+5)nw$ Bytes zu senken.

Wir können jedoch noch weiter gehen. Hierzu schauen wir uns an, wie genau die Arrays P und F benutzt werden. In den Zeilen 14–26 wird für jeden Eintrag, den wir aus U entfernen, entweder ein Element zu F, P, oder S hinzugefügt. Es gilt somit zu jedem Zeitpunkt $|S| + |U| + |F| + |P| = n$. Dies gilt ebenso in den Zeilen 8–12 und 31, da die beteiligten Arrays immer um dieselbe Anzahl von Elementen erweitert und verringert werden.

Es ist somit erneut möglich, alle Arrays durch ein kombiniertes Array $SUPF$ der Größe $(a+1)nw$ zu repräsentieren, indem jeder Eintrag entweder S, U, P oder F zugeordnet ist. Das Problem hierbei ist, dass wir nun in jedem Schritt des Algorithmus jeden Eintrag des kombinierten Arrays eindeutig einen der vier ursprünglichen Arrays zuordnen müssen und sie in denselben Laufzeitschranken wie zuvor iterieren und erweitern bzw. verkürzen wollen.

Wir lösen diese Problem in der Implementierung wie folgt:

1. Alle Elemente der Teil-Arrays P, S und U liegen immer konsekutiv in $SUPF$. Dies erlaubt es, die Arraygrenzen mit konstanten Platzverbrauch zu speichern und auf jedes Element in konstanter Zeit zuzugreifen.

2. Die Elemente von F sind aufgeteilt in ein Teil-Array aller Elemente von vergangenen Iterationen und der der aktuellen Iteration. Beide Arrays werden nach jeder Iteration ohne zusätzliche Kosten vereint.

3. Teil-Arrays werden verkürzt, indem das erste oder letzte Element entfernt wird. Dies lässt eine leere Stelle zurück.

4. Teil-Arrays werden erweitert, indem ein Element in eine anliegende Stelle eingefügt wird. Liegt dort ein Element eines Nachbararrays, wird dieses rekursiv entfernt und an der anderen Seite eingefügt. Dies verschiebt das Nachbararray in konstanter Zeit um ein Element, aber erhält nicht seine Reihenfolge.

5. Teil-Arrays, die vereinigt oder ineinander umgewandelt werden liegen nebeneinander oder nehmen denselben Platz ein.

Daraus folgt folgendes Layout von $SUPF$:

1. P liegt am linken Rand, da dort die Reihenfolge wichtig ist und äußere Teil-Arrays nicht verschoben werden können.

2. S liegt neben P, da es in U umbenannt und danach mit P vereinigt wird.

3. Neue F Elemente liegen neben S

4. U liegt rechts von neuen F Elementen und links neben F-Elemente der vorherigen Iteration.

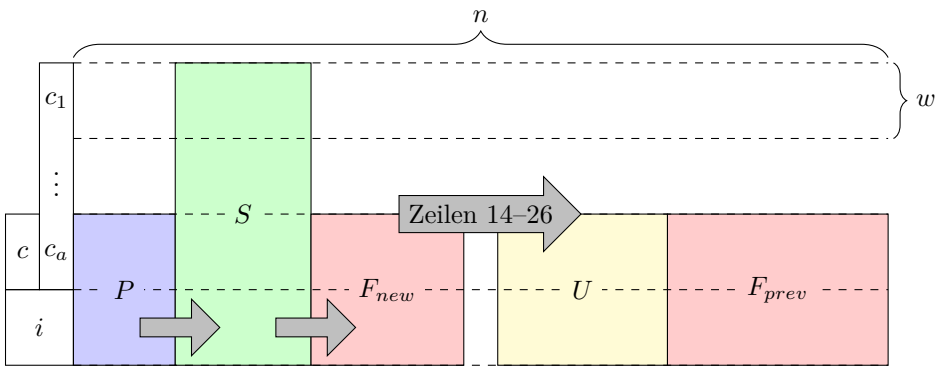


Abbildung 6.9: Discarding-Speicherlayout bei Array-Überlagerung

5. Das finale F sammelt sich iterativ am rechten Rand von $SUPF$ und wird in jeder Iteration nach links um mögliche neue F Elemente erweitert.

Siehe Abbildung 6.9 für ein Beispiel für den Zustand von $SUPF$ während Zeile 14–26 des Algorithmus. Diese Optimierung ermöglicht uns, den Speicherverbrauch von Discarding auf denselben Wert $(a+1)nw$ wie für das normale Doubling zu reduzieren.

Wordpacking

Sowohl Doubling und Discarding erstellen eine initiale Version des S-Arrays, indem benachbarte Zeichen der Eingabe in den Tupeln $(c_1, \ldots, c_a)$ als Namen benutzt werden. Da die Zeichen der Eingabe jedoch maximal ein Byte belegen und `sa_index` aus ≥ 4 Bytes besteht, enthält jeder c_x in der ersten Iteration ≥ 3 Nullbytes.

Die Idee des Wordpackings ist es, den Wertebereich der Namen c_x voll auszunutzen, indem man sie aus der Konkatenation von benachbarten Eingabe-Bytes bildet. Die Namenstupel aus S repräsentieren somit nicht mehr a benachbarte Zeichen, sondern aw. Da gilt $w \geq 4$, erlaubt uns dies in der Regel die ersten Iterationen der Algorithmen zu überspringen. Da diese gerade beim Discarding auch die aufwändigsten sind, reduziert dies die Rechenzeit um einen signifikanten Anteil.

Schnellerer Sortierschlüssel

Alle Varianten der Algorithmen enthalten in der k-ten Iteration einen Sortierschritt, der Tupel (c, i) gemäß des Sortierschlüssels $(i \mod 2^k, i \div 2^k)$ sortiert. Experimente basierend auf Profiling des kompilierten C++-Codes ergaben folgende Erkenntnisse:

1. Die Modulorechnung macht einen signifikanten Anteil der Sortierlaufzeit aus.
2. Ersetzt man sie für $a = 2$ oder $a = 4$ durch Bitshifting Operationen reduziert sich die Laufzeit auf $< 50\%$, aber sie machen immer noch einen signifikanten Anteil aus.

Dies liegt darin begründet, dass wir neben den $\mathcal{O}(n \log n)$ Vergleichsoperationen auch $\mathcal{O}(n \log n)$ Rechenoperationen durchführen müssen, um die zu vergleichenden Integer-Werte zu erhalten. Eine Verringerung der Rechenkosten würde sich somit vorteilhaft auf die Laufzeit auswirken. Wir stellen nun folgendes fest:

1. Unmittelbar vor dem Sortierschritt wird eine Benennungsphase durchgeführt, die sequentiell alle (c, i) Tupel erzeugt.
2. Unmittelbar nach dem Sortierschritt findet entweder eine Paarbildungs- oder eine Discarding-Phase statt, die sequentiell alle sortierten (c, i) Tupel verarbeitet.
3. Die Tupel $(i \mod 2^k, i \div 2^k)$ lassen sich für $a = 2$ oder $a = 4$ als eine Bitrotation des Integers i in der gleichen Anzahl von Bits repräsentieren und erhalten dabei dieselbe Ordnung untereinander. Wir bezeichnen diese Repräsentation als i_{rot}

Dies ermöglicht es uns die Rechenoperationen auf $\mathcal{O}(n)$ zu senken, indem wir die Tupeltransformation nicht während der Sortierung durchführen, sondern als Vor- bzw. Nachbearbeitungsschritt während der direkt anliegenden Phasen. Konkret führen wir folgende Modifikation durch:

- Für jedes (c, i) Tupel, das die Benennungsphase erzeugt, berechnen und speichern wir (c, i_{rot}) im zugehörigen Array.
- Die Sortierphase sortiert die (c, i_{rot}) Tupel anhand i_{rot} gemäß der natürlichen Ordnung auf $\mathbb{N}$.
- Die anschließende Paarbildungs- bzw. Discarding-Phase transformiert die sortierten (c, i_{rot}) Tupel zurück zu den ursprünglichen (c, i) Tupeln und verarbeitet sie dann normal weiter.

In-Place-Merging

Der Discarding-Algorithmus 6.3 führt in Zeile 10 einen Merge-Schritt zwischen den P- und U-Arrays durch, nachdem U gemäß des Sortierschlüssels für k sortiert wurde.

Da ein einfacher Merge-Schritt zusätzlichen Speicher benötigt, erfolgt dies in der anfänglichen Implementierung durch eine simple Konkatenation von P mit U *vor* der Sortierung. Die Laufzeit beträgt somit $\mathcal{O}((|P| + |U|) \log(|P| + |U|))$, anstatt $\mathcal{O}(|U| + |P| + |U| \log |U|)$.

Als Optimierung fand die Implementierung deshalb in Form eines In-Place Merge-Algorithmus statt. Es wurde dabei ausgenutzt, dass durch die Array-Überlagerung in den zu vereinigenden Arrays U und P unbenutzte freie Tupelkomponenten vorhanden sind.

Diese werden als zusätzliches Array der Länge $|U| + |P|$ mit Platz für $a - 1$ Elemente pro Position interpretiert und genutzt um die (c, i)-Tupel von P und dem sortierten U in $\mathcal{O}(|U| + |P|)$ zu vereinigen. Für $a = 2$ sind dabei zwei Scans notwendig, da nicht beide Komponenten der Tupel gleichzeitig an eine freie Stelle kopiert werden können.

Schnellerer Sortieralgorithmus

Da unsere Implementierung nur im Arbeitsspeicher arbeitet, sind wir nicht an einen bestimmten extern arbeitenden Sortieralgorithmus gebunden, weshalb es uns frei steht einen Sortierer zu nutzen, der das beste Laufzeitverhalten zeigt.

Wir haben hierzu den Discarding-Algorithmus mit allen im Framework verfügbaren Sortierern auf einer Reihe von Testdaten angewandt, um dem sich empirisch am besten verhaltenden Algorithmus zu wählen. Es stellte sich hierbei der externe IPs^{4}o Algorithmus (siehe Abschnitt 4.1.6) als mit Abstand bester Algorithmus heraus.

Ausgelassene Optimierungsansätze

Da der Ausgabe-`span` zu einer Allokation von mindestens $n \log n$ Bits zeigt, würde er sich im Prinzip als temporären Speicher für eine der Komponenten der von dem Algorithmus benutzten Tupel-Arrays benutzen lassen, um den Speicherverbrauch während der Berechnung um denselben Wert zu senken.

Dies würde jedoch die Implementierung wesentlich komplexer machen, da die Komponenten eines Tupels nun über mehrere disjunkte `span`s verteilt wären. Da das Interface der Sortierer-Implementierungen darauf beruht, nur vollständige Elemente eines Arrays zu sortieren, wäre somit also auch eine Anpassung der benutzten Sortieralgorithmen notwendig.

6.3.12 Parallelisierung

Alle vorgestellten Varianten des Algorithmus lassen sich trivial parallelisieren, indem ein paralleler Sortieralgorithmus benutzt wird. Wir nutzen hierzu den parallelen IPs^{4}o Algorithmus aus Abschnitt 4.1.6.

In anderen Aspekten ist eine parallelisierung der Algorithmen nicht effektiv möglich, da sowohl die Umbenennungsphase als auch die Paarbildungsphase inhärent sequentielle Abläufe beinhalten.

6.4 DC3

Im Jahre 2003 veröffentlichten Juha Kärkkäinen und Peter Sanders einen der ersten Algorithmen, der ein Suffix-Array in linearer Laufzeit $\mathcal{O}(n)$ erstellen kann [29]. Dieser Algorithmus heißt *DC3*, wird aber auch *skew* genannt. Der *DC3*-Algorithmus wird in den meisten wissenschaftlichen Veröffentlichungen gerne als Vergleichsalgorithmus angesehen, sodass dieser auch im Rahmen der Projektgruppe SACABench nicht fehlen darf. Außerdem dient der *DC3* als Grundlage für den Algorithmus *nzSufSort*, der im Kapitel 6.8 näher erläutert wird.

6.4.1 Einführung

Der *DC3*-Algorithmus wird in der wissenschaftlichen Ausarbeitung von Juha Kärkkäinen und Peter Sanders wiederholt als simpler linearer Algorithmus zur Konstruktion eines Suffix-Arrays dargestellt. Der Wortlaut *simpel* bezieht sich jedoch auf die Implementierung des Algorithmus. Diese benötigt nämlich lediglich 50 Zeilen Code in der Programmiersprache *C++*. Die Theorie, die sich dabei im Hintergrund abspielt, ist jedoch nicht ganz trivial. Daher werden wir uns mit der Theorie näher auseinandersetzen und diese anhand von Beispielen besser verdeutlichen. Dabei gehen wir am Ende auch auf Erweiterungen und Verbesserungsvorschläge ein und bewerten diese anhand von Laufzeit- und Speicherplatz-Messungen.

6.4.2 Vorüberlegungen

Der *DC3*-Algorithmus baut auf dem *Divide-and-Conquer* Prinzip auf. Diese Art von Algorithmen teilen das Hauptproblem rekursiv in kleinere Teilprobleme auf, bis diese leichter zu lösen sind. Im Anschluss werden die Teilprobleme zu einer Gesamtlösung zusammengefügt.

Das zweite Prinzip, das sich hinter dem *DC3*-Algorithmus verbirgt, ist das sogenannte *Difference Cover*. Diesem Prinzip verdankt der Algorithmus auch seinen Namen.

Definition 13 (*Difference Cover*)**.** *Eine Menge $D \subseteq [0, v)$ ist ein Difference Cover modulo v, wenn die Werte in $[0, v)$ als Differenz zweier Werte aus $D \subseteq [0, v)$ ausgedrückt werden können. Mathematisch dargestellt:*

$$\{(i - j) \text{ modulo } v \mid i, j \in D\} = [0, v)$$

Beispiel 1. *Ein Beispiel für das Difference Cover* modulo 7*: D = {1, 2, 4}*

$$\begin{array}{lll} 1 - 1 = 0 & 1 - 4 = -3 \equiv 4 & (\mod 7) \\ 2 - 1 = 1 & 2 - 4 = -2 \equiv 5 & (\mod 7) \\ 4 - 2 = 2 & 1 - 2 = -1 \equiv 6 & (\mod 7) \\ 4 - 1 = 3 & & \end{array}$$

Das bedeutet, dass wir mit der Menge D = {1, 2, 4} die Zahlen aus $[0, 7)$ *mit der Hilfe von Differenzen zweier Werte aus D ausdrücken können.*

Ein Beispiel für das Difference Cover modulo 3*: D = {1, 2}*

6.4.3 Algorithmus

In dem vorherigen Kapitel 6.4.2 ist die Definition des *Difference-Cover*-Prinzips näher erläutert worden. Nun wird beschrieben, wie dieses Verfahren eingesetzt werden kann, um ein Suffix-Array zu erhalten – wobei wir uns in diesem Kapitel nur mit dem *Difference Cover* modulo 3 beschäftigen, da dieser auch in der wissenschaftlichen Ausarbeitung von Juha Kärkkäinen und Peter Sanders behandelt wird. Dieser wird dabei in drei Phasen aufgeteilt. Allgemein ausgedrückt werden in der ersten Phase die Suffixe mit Startpositionen aus dem *Difference Cover* $D = \{1, 2\}$ sortiert. In der zweiten Phase werden die restlichen Suffixe unter Verwendung des Ergebnisses des vorherigen Schrittes sortiert. Und – wie im *Divide-and-Conquer* Prinzip üblich – werden in der dritten Phase die beiden sortierten Lösungen aus der ersten und zweiten Phase zusammengeführt, sodass am Ende eine sortierte Menge aller Suffixe des Ausgangstextes entsteht – das Suffix-Array.

Erste Phase - Sortierung des *Difference Covers*

In dem ersten Schritt des Algorithmus werden die Substrings $\mathsf{T}[i, i+2]$ sortiert, wobei i ein Element aus dem *Difference Cover* $D = \{1, 2\}$ ist. Das bedeutet, es werden alle Substrings der Länge drei – auch Triplets genannt – startend in den Positionen i modulo $3 = 1$, also $i = 1, 4, 7, 10, ...$, und den Positionen i modulo $3 = 2$, also $i = 2, 5, 8, 11, ...$, des Ausgangstextes T aufsteigend sortiert. Anschließend werden den Triplets der Reihenfolge nach lexikographische Namen $t_i \in [0, \lceil 2n/3 \rceil)$ mit der Eigenschaft, dass $t_i < t_j$ wenn $\mathsf{T}[i, i+2] < \mathsf{T}[j, j+2]$ und $t_i = t_j$ wenn $\mathsf{T}[i, i+2] = \mathsf{T}[j, j+2]$, zugewiesen. Das bedeutet, dass das kleinste Triplet den lexikographisch kleinsten Namen 0 erhält, das zweitkleinste Triplet die 1 und so weiter. Sind mehrere Triplets gleich, das heißt, sowohl die erste, zweite und dritte Stelle des Substrings sind jeweils gleich, dann erhalten diese Triplets die gleichen lexikographischen Namen. Wenn jedem Triplet ein lexikographischer Name zugeordnet worden ist, wird überprüft, ob diese Namen eindeutig sind. Wenn in dem Ausgangstext keine gleichen Triplets mit den beschriebenen Eigenschaften vorkommen, sind wir mit dem ersten Schritt des *DC3*-Algorithmus fertig und können mit der zweiten Phase fortfahren. Kommen in dem Ausgangstext T jedoch gleiche Triplets startend in den Positionen i modulo $3 \neq 0$ vor, so können wir aktuell noch keine eindeutige Aussage über die Reihenfolge der gleichen Suffixe treffen. Um diese eindeutige Aussage treffen zu können, schreiben wir die lexikographischen Namen so um, dass die Ordnung der Triplets beibehalten wird. Dabei werden die Namen t_i in die Mengen i modulo $3 = 1$ und i modulo $3 = 2$ aufgeteilt und diese zu einem neuen String T_{12} konkateniert. Mathematisch dargestellt, bedeutet das, dass der String

$$\mathsf{T}_{12} = [t_i \mid i \text{ modulo } 3 = 1] \circ [t_i \mid i \text{ modulo } 3 = 2]$$

als neuer Ausgangstext angesehen und der *DC3*-Algorithmus auf diesem String ausgeführt wird, sodass wir nach erneutem Ausführen des Algorithmus die endgültige Sortierung der Triplets erhalten und mit dem zweiten Schritt fortfahren können.

Es gibt jedoch einen Spezialfall, bei dem dieser Schritt zu einem falschem Suffix-Array führen kann. Der Spezialfall tritt auf, wenn folgende drei Punkte gleichzeitig zutreffen:

1. die Textlänge ist n modulo $3 = 1$
2. das Triplet an der Position $n - 3$ kommt in dem Text mehrmals vor
3. das Triplet an der Position $n - 2$ ist nicht das kleinste Triplet

Denn dann ist der String mit den lexikographischen Namen falsch, weil der Algorithmus nicht weiß, dass mit dem Triplet an der Position $n-3$ das Ende des Textes erreicht ist. Der Spezialfall tritt zum Beispiel bei dem Text $\mathsf{T} = aabcabc$ auf. Der String mit den lexikographischen Namen wäre $t_{12} = [1, 1] \circ [3, 2]$, also $\mathsf{T}_{12} = 1132$. Wird der String in dieser Form rekursiv aufgerufen, wird am Ende des Algorithmus das Suffix $\mathsf{T}[1, n)$ kleiner sein als das Suffix $\mathsf{T}[4, n)$, weil das Triplet 113 in der Rekursion kleiner ist als 132. Um dagegen vorzubeugen, wird vor Beginn des Algorithmus dem Ausgangstext ein sogenanntes Dummy-Triplet angehangen, das aus Sentinels besteht. Dadurch wird dafür gesorgt, dass das Ende des Textes mit in die Sortierung eingeht.

Zweite Phase - Induzierung

Wir haben nun in der ersten Phase die eindeutige Reihenfolge der Suffixe aus dem *Difference Cover* bestimmt. Jetzt können wir das Prinzip des *Difference Covers* anwenden und die Reihenfolge der restlichen Suffixe beginnend in Position i modulo $3 = 0$ induzieren. Dafür benötigen wir die Ränge der Suffixe aus dem *Difference Cover*, die sich aus den lexikographischen Namen der ersten Phase ableiten lassen. Also berechnen wir das inverse Suffix-Array ISA_{12}, das die jeweiligen Ränge repräsentiert.

$$\mathsf{SA}_{12}[i] = j \text{ genau dann, wenn } \mathsf{ISA}_{12}[j] = i$$

Jetzt lassen sich Paare aufstellen, die sich aus einem Zeichen $\mathsf{T}[i]$ des Ausgangstextes und dem Rang des darauffolgenden Suffixes zusammensetzen, wobei i modulo $3 = 0$ ist. Nachdem alle Paare aufgestellt sind, können diese ebenfalls aufsteigend sortiert werden und anschließend die jeweiligen Positionen i der Paare in ein Array SA_0 abgespeichert werden. Somit haben wir auch die Suffixe aufsteigend sortiert, die nicht in dem *Difference Cover* sind, und können zur dritten Phase übergehen.

Dieses Prinzip funktioniert, da die Suffixe startend in Position i modulo $3 = 0$ nur ein Zeichen am Anfang des Suffixes mehr aufweisen als die Suffixe beginnend in i modulo $3 = 1$ und der Rest gleich ist. Und die Suffixe in i modulo $3 = 1$ sind bereits in der ersten Phase eindeutig sortiert worden. Dieses hilft uns in der zweiten Phase weiter, denn dann reicht es aus, die jeweiligen Zeichen in i modulo $3 = 0$ zu vergleichen und bei Gleichheit die Ränge der Suffixe einer Position hinter den jeweiligen Zeichen anzuschauen. Dadurch können zwar Paare

das gleiche Zeichen $\mathsf{T}[i]$ aber niemals den gleichen Rang aufweisen und das führt zu einer eindeutigen Sortierung der Suffixe startend in Position i modulo $3 = 0$.

Dritte Phase - Merge

Wir haben aus den ersten beiden Phasen die jeweils sortierten Suffixe des *Difference Covers* SA_{12} und diejenigen, die nicht in dem *Difference Covers* SA_0 sind, vorliegen. Im letzten Schritt müssen diese beiden Mengen vereinigt werden, um an das Suffix-Array SA zu gelangen. Dabei nutzen wir aus, dass die jeweiligen Mengen bereits sortiert sind. Somit vergleichen wir immer das kleinste Suffix aus der Menge SA_{12} mit dem kleinsten aus der Menge SA_0. Dabei können bei dem Vergleich von $\mathsf{SA}_{12}[i]$ und $\mathsf{SA}_0[j]$ folgende vier Fälle auftreten.

1. $\mathsf{SA}_{12}[i]$ modulo $3 = 1$
2. $\mathsf{SA}_{12}[i]$ modulo $3 = 2$
3. $i >$ length(SA_{12})
4. $j >$ length(SA_0)

Tritt Fall 1 ein, bedeutet das, dass ein Suffix aus SA_0 mit einem Suffix aus SA_{12}, dessen Startposition $\mathsf{SA}_{12}[i]$ modulo $3 = 1$ ist, miteinander verglichen wird. Für diesen Vergleich benötigen wir die Paare $(\mathsf{T}[\mathsf{SA}_{12}[i]], \mathsf{ISA}_{12}[\mathsf{SA}_{12}[i]+1])$ und $(\mathsf{T}[\mathsf{SA}_0[j]], \mathsf{ISA}_{12}[\mathsf{SA}_0[j]+1])$. Wenn wir diese zwei Paare ermittelt haben, lassen sie sich miteinander vergleichen. Der kleinere Index von beiden wird dann dem Suffix-Array SA hinzugefügt und der andere Index wird mit dem nächsten verglichen. Bei Fall 1 wird ein Zeichen und der Rang des darauffolgenden Suffix miteinander verglichen. Dies funktioniert, da sowohl der Rang der Position nach i modulo $3 = 0$ als auch der Rang des Suffixes nach i modulo $3 = 1$ bekannt ist. Tritt Fall 2 ein, bedeutet das, dass ein Suffix aus SA_0 mit einem Suffix aus SA_{12}, dessen Startposition $\mathsf{SA}_{12}[i]$ modulo $3 = 2$ ist, miteinander verglichen wird. Für diesen Vergleich stellen wir – anders als in Fall 1 – die Triplets $(\mathsf{T}[\mathsf{SA}_{12}[i]], \mathsf{T}[\mathsf{SA}_{12}[i]+1], \mathsf{ISA}_{12}[\mathsf{SA}_{12}[i]+2])$ und $(\mathsf{T}[\mathsf{SA}_0[j]], \mathsf{T}[\mathsf{SA}_0[j]+1], \mathsf{ISA}_{12}[\mathsf{SA}_0[j]+2])$ auf. Wie zuvor werden diese beiden Triplets nun wieder verglichen und der Index des kleineren Triplets dem Suffix-Array angehangen. Bei Fall 2 werden zwei Zeichen und der Rang des Suffix, das zwei Zeichen später beginnt, miteinander verglichen. Dies funktioniert, da sowohl der Rang des Suffixes zwei Positionen nach j modulo $3 = 0$ als auch der Rang zwei Positionen nach i modulo $3 = 2$ bekannt ist. Ein Zeichen und der darauffolgende Rang – wie in Fall 1 – reicht für einen Vergleich nicht aus, da der Rang nach i modulo $3 = 2$ eine Position startend in $(i + 1)$ modulo $3 = 0$ ist und somit nicht miteinander vergleichbar ist.
Bei den Fällen 3 und 4 sind eines der beiden Mengen SA_0 oder SA_{12} abgearbeitet und der Rest des Suffix-Arrays kann mit den jeweils restlichen Indizes der übrig gebliebenen Menge aufgefüllt werden.
Sind beide Mengen durchlaufen worden, haben wir am Ende ein vollständiges Suffix-Array und der Algorithmus ist terminiert.

Laufzeit

In diesem Kapitel untersuchen wir die *Worst-Case*-Laufzeit des *DC3*-Algorithmus, indem wir zuerst die Laufzeiten der einzelnen Phasen betrachten und diese am Ende zu einer Gesamtlaufzeit zusammenführen. In der ersten Phase werden Triplets aufgestellt und diese anschließend sortiert. Für die Sortierung könnte zum Beispiel das Sortierverfahren *Radix-Sort* verwendet werden. Der *LSD-Radix-Sort*, der im Kapitel 4.1.5 näher erläutert worden ist, benötigt eine Laufzeit von $\mathcal{O}(kn)$, wobei k für die Länge der zu vergleichenden Werte steht. In unserem Fall ist $k = 3$. Für die Vergabe der lexikographischen Namen wird ebenfalls eine lineare Laufzeit benötigt. Zusammengefasst benötigt die erste Phase eine Laufzeit von $T(n) = \mathcal{O}(n)$.

Die zweite Phase stellt Paare auf, welche erneut mit Hilfe von *Radix-Sort* sortiert werden. Dies benötigt eine Laufzeit von $T(n) = \mathcal{O}(n)$.

In der dritten Phase werden beide Mengen SA_0 und SA_{12} jeweils einmal durchlaufen und einfache Vergleiche ausgeführt. Dies führt ebenfalls zu einer Laufzeit von $T(n) = \mathcal{O}(n)$.

Also weist jede Phase eine Laufzeit von $\mathcal{O}(n)$ auf. Im *Worst-Case* sind die lexikographischen Namen aus der ersten Phase jedoch nicht eindeutig, sodass ein rekursiver Aufruf erfolgt. Dabei wird der Algorithmus mit einer Textlänge von $\frac{2n}{3}$ aufgerufen. Zusammengefasst erhalten wir somit für den gesamten Algorithmus eine Laufzeit von $T(n) = \mathcal{O}(n) + T(\frac{2n}{3})$. Da der Faktor $\frac{2}{3}$ kleiner als 1 ist, lässt sich diese Rekursionsgleichung zu $T(n) = \mathcal{O}(n)$ auflösen. Somit ist die Laufzeit des *DC3*-Algorithmus linear.

Beispiel

In diesem Kapitel wollen wir die Theorie des *DC3*-Algorithmus anhand eines Beispieles verdeutlichen. Dazu wenden wir den Algorithmus auf den String T = caabaccaabacaa\$ an, um das gesuchte Suffix-Array zu erhalten.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\mathsf{T}[i]$	c	a	a	b	a	c	c	a	a	b	a	c	a	a	\$

Bei der ersten Phase werden alle Triplets beginnend in Positionen
i modulo $3 = 1$ und i modulo $3 = 2$ aufgestellt, wie in Tabelle 6.8 zu erkennen.

i	1	4	7	10	13	2	5	8	11	14
$\mathsf{T}[i, i+2]$	aab	acc	aab	aca	a\$\$	aba	cca	aba	caa	\$\$\$

Tabelle 6.8: Triplets i modulo $3 \neq 0$

Dabei fällt auf, dass die Triplets an der Stelle $i = 13$ und $i = 14$ mit Sentinels aufgefüllt werden, damit sie besser mit den anderen zu vergleichen sind. Nun können wir diese Triplets zum Beispiel mit Hilfe des Sortieralgorithmus *Radix-Sort* aufsteigend sortieren. Das Ergebnis ist in der Tabelle 6.9 festgehalten.

i	14	13	1	7	2	8	10	4	11	5
$\mathsf{T}[i, i+2]$	\$\$\$	a\$\$	aab	aab	aba	aba	aca	acc	caa	cca

Tabelle 6.9: sortierte Triplets i modulo $3 \neq 0$

Nun können wir die lexikographischen Namen vergeben. Dafür werden die Positionen aufsteigend nummeriert. Falls mehrere Triplets gleich sind, erhalten sie den gleichen lexikographischen Namen. Anschließend werden diese lexikographischen Namen so umsortiert, dass wir die Namen für die Positionen i modulo $3 = 1$ und die Positionen i modulo $3 = 2$ konkatenieren können. Das Ergebnis dieser zwei Schritte ist in der Tabelle 6.10 zu finden.

i	1	4	7	10	13	2	5	8	11	14
T_{12}	2	5	2	4	1	3	7	3	6	0

Tabelle 6.10: Vergabe der lexikographischen Namen

Hier ist zu erkennen, dass mehrere Triplets den gleichen lexikographischen Namen erhalten haben. Somit haben wir noch nicht die endgültige Reihenfolge ermitteln können. Dafür rufen wir den Algorithmus rekursiv mit diesen lexikographischen Namen $\mathsf{T}_{12} = 2524137360$ auf. Als Ergebnis der Rekursion erhalten wir $\mathsf{SA}_{12} = [9, 4, 2, 0, 7, 5, 3, 1, 8, 6]$. Das Ergebnis lässt sich so deuten, dass sich das kleinste Triplet an Stelle 9 von T_{12} befindet, das der Position $i = 14$ aus dem Ausgangstext T entspricht, also $\mathsf{SA}_{12} = [14, 13, 7, 1, 8, 2, 10, 4, 11, 5]$.

Damit ist die erste Phase des Algorithmus abgeschlossen und wir gehen über zur zweiten Phase. Dafür werden zunächst die Ränge bestimmt mit Hilfe des inversen Suffix-Arrays, das sich wie folgt berechnen lässt: $\mathsf{SA}_{12}[i] = j$ genau dann, wenn $\mathsf{ISA}_{12}[j] = i$.

i	1	4	7	10	13	2	5	8	11	14
ISA_{12}	4	8	3	7	2	6	10	5	9	1

Tabelle 6.11: Ränge der Suffixe i modulo $3 \neq 0$

Im Anschluss lassen sich die Suffixe i modulo $3 = 0$ mit Hilfe des Ausgangstextes T und der Ränge ISA_{12} sortieren. An diesem Beispiel kann man die Theorie gut verdeutlichen. Denn anstatt sich – wie in der ersten Phase – erneut Triplets anzuschauen und möglicherweise vor dem Problem zu stehen, gleiche Triplets zu erhalten, wird die eindeutige Sortierung aus der ersten Phase genutzt. Schaut man sich das Zeichen 'c' an der ersten und siebten Position von T an, muss man sich nur die Ränge der Suffixe an der jeweils nächsten Position anschauen. Daraus lässt sich erschließen, dass das Suffix an Position $i = 6$ kleiner als das Suffix an Position $i = 0$ ist, weil der Rang des Suffixes an $i = 7$ kleiner ist als das an $i = 1$.

i	12	9	3	6	0
$(\mathsf{T}[i], \mathsf{ISA}_{12}[i+1])$	(a, 2)	(b, 7)	(b, 8)	(c, 3)	(c, 4)

Tabelle 6.12: Sortierte Paare $(\mathsf{T}[i], \mathsf{ISA}_{12}[i+1]), i$ modulo $3 = 0$

Als Ergebnis erhalten wir $\mathsf{SA}_0 = [12, 9, 3, 6, 0]$ - wie in der Tabelle 6.12 nachzuvollziehen ist.

Nun können wir in der dritten Phase die bereits sortierten Mengen SA_0 und SA_{12} mergen. Dafür stellen wir – wie im vorherigen Kapitel 6.4.3 beschrieben – die benötigten Paare und Triplets auf, je nachdem, welcher der Fälle 1 bis 4 zutrifft. Als erstes vergleichen wir die beiden Suffixe beginnend in $\mathsf{SA}_{12}[i = 0] = 14$ und $\mathsf{SA}_0[j = 0] = 12$. Da 14 modulo $3 = 2$ ist – also Fall 2 zutrifft –, stellen wir die Triplets $(\$, \$, 0)$ und $(a, a, 1)$ auf. Der Vergleich ergibt, dass das Suffix startend in $\mathsf{SA}_{12}[i = 0] = 14$ kleiner ist. Somit wird dem Suffix-Array SA der Index 14 hinzugefügt und als nächstes $\mathsf{SA}_{12}[i = 1] = 13$ mit $\mathsf{SA}_0[j = 0] = 12$ verglichen. 13 modulo 3 ergibt 1. Das heißt, der Fall 1 ist eingetreten und wir vergleichen die Paare $(a, 1)$ und $(a, 2)$. So werden beide Mengen einmal durchlaufen und wir erhalten am Ende das endgültige Suffix-Array $\mathsf{SA} = [14, 13, 12, 7, 1, 8, 2, 10, 4, 9, 3, 11, 6, 0, 5]$.

6.4.4 Erweiterungen

DC7

Wie in Abschnitt 6.4.2, ist nicht nur ein *Difference Cover* modulo 3 möglich, sondern auch zum Beispiel modulo $7, 13, 21$ und 31. Im Rahmen der Projektgruppe *SACABench* wurde zusätzlich zum *DC3* auch der *DC7* implementiert. Dafür werden jedoch ein paar Änderungen an dem Algorithmus vorgenommen, die der wissenschaftlichen Arbeit von Juha Kärkkäinen, Peter Sanders und Stefan Burkhardt entnommen worden ist [26]. Für den *DC7*-Algorithmus verwenden wir das *Difference Cover* $D = \{1, 2, 4\}$.

Erste Phase Die erste Phase ist analog zu der ersten Phase des *DC3*-Algorithmus. Hier werden die Septets $\mathsf{T}[i, i+6]$ an den Positionen i modulo $7 \in \{1, 2, 4\}$ aufsteigend sortiert. Anschließend werden lexikographische Namen vergeben und diese so umsortiert, dass die Ordnung der Septets beibehalten wird.

$$\mathsf{T}_{124} = [t_i \mid i \text{ modulo } 7 = 1] \circ [t_i \mid i \text{ modulo } 7 = 2] \circ [t_i \mid i \text{ modulo } 7 = 4]$$

Wenn die lexikographischen Namen eindeutig sind, kann mit der zweiten Phase fortgefahren werden, ansonsten wird der *DC7*-Algorithmus erneut mit dem String T_{124} der Länge $\mathcal{O}(\frac{3n}{7})$ ausgeführt.

Zweite Phase In der zweiten Phase werden die Suffixe an den Positionen i modulo $7 \notin \{1, 2, 4\}$ sortiert, indem die Tupel $(\mathsf{T}[i, i+5], \mathsf{R}[i+6])$ aufgestellt und aufsteigend sortiert werden, wobei $\mathsf{R}[i+6]$ den Rang des Suffixes beginnend in Position $i+6$ repräsentiert. Dafür müssen wir eine bestimmte Reihenfolge einhalten. Demnach werden zuerst die Suffixe an den Positionen i modulo $7 = 3$ und i modulo $7 = 5$ bestimmt, indem die Ränge der Positionen $(i+6)$ modulo $7 = 2$ beziehungsweise $(i+6)$ modulo $7 = 4$ zu Hilfe genommen werden. Anschließend werden die Ränge der Positionen i modulo $7 = 5$ bestimmt, da diese für die Sortierung der Suffixe an i modulo $7 = 6$ benötigt werden. Danach werden die Ränge von i modulo $7 = 6$ bestimmt, um damit die noch übrig gebliebenen Suffixe an i modulo $7 = 0$ zu sortieren.

Dritte Phase Für die dritte Phase gibt es zwei verschiedene Ansätze. Zuerst betrachten wir den naiven Ansatz. Hierbei wird – ähnlich wie bei dem *DC3*-Algorithmus – der jeweils kleinste Werte der Mengen SA_{124}, SA_0, SA_3, SA_5 und SA_6, der noch nicht in dem endgültigen Suffix-Array einsortiert worden ist, miteinander verglichen. Dabei werden bei den Vergleichen zwischen i und j jeweils ein l gesucht, sodass $(i+l)$ modulo 7 und $(j+l)$ modulo 7 aus dem *Difference Cover* $D = \{1, 2, 4\}$ sind. Zur Bestimmung der Länge l kann die Tabelle 6.13 verwendet werden.

i/j	0	1	2	3	4	5	6
0	0	1	2	1	4	4	2
1	1	0	0	1	0	3	3
2	2	0	0	6	0	6	2
3	1	1	6	0	5	6	5
4	4	0	0	5	0	4	5
5	4	3	6	6	4	0	3
6	2	3	2	5	5	3	0

Tabelle 6.13: Merge-Tabelle

Nun werden Tupel der Länge $l+1$ aufgestellt, die aus l Zeichen beginnend von i beziehungsweise j und dem Rang R der Position $i+l$ beziehungsweise $j+l$ bestehen, also $(\mathsf{T}[i, i+l], \mathsf{R}[i+l+1])$ und $(\mathsf{T}[j, j+l], \mathsf{R}[j+l+1])$. Wird dieser Ansatz verwendet, muss ein 7-Wege-Merge mit jeweils $\mathcal{O}(7)$-Vergleichen durchgeführt werden, was zu einer *Worst-Case*-Laufzeit von $\mathcal{O}(7n \log(7)) = \mathcal{O}(n)$ führt.

Der zweite Ansatz verfolgt eine theoretisch bessere Idee im Bezug auf die Laufzeit. Dafür wird die Menge SA_{124} in die drei Mengen SA_1, SA_2 und SA_4 aufgeteilt, wobei die jeweilige Sortierung beibehalten wird. Anschließend werden alle Mengen zu einer Menge $\mathsf{SA}_{0:6} = \mathsf{SA}_0 \circ \mathsf{SA}_1 \circ \mathsf{SA}_2 \circ \mathsf{SA}_3 \circ \mathsf{SA}_4 \circ \mathsf{SA}_5 \circ \mathsf{SA}_6$ konkateniert. Nun wird diese Menge stabil nach den ersten sieben Zeichen sortiert. Die Eigenschaft *stabil* ist in diesem Fall sehr wichtig, da wir die eindeutige vorherige Sortierung aus den ersten zwei Phasen nicht durcheinander bringen

dürfen. Als nächstes müssen die noch gleichen Tupel mit einem vergleichsbasiertem 7-Wege-Merge sortiert werden. Dies passiert, indem wir die Ränge von $(i+l)$ und $(j+l)$ miteinander vergleichen. Dadurch erhalten wir das endgültige Suffix-Array. Der zweite Ansatz hat in der Theorie eine bessere *Worst-Case*-Laufzeit von $\mathcal{O}(7n) = \mathcal{O}(n)$.

DC3-Lite

Es ist möglich den Speicherverbrauch des *DC3* so weit zu reduzieren, dass nur ein zusätzliches Hilfs-Array U der Länge n verwendet wird. Diese Variante des *DC3* wurde in [43] vorgestellt und wird als Komponente im *nzSufSort* Abschnitt 6.8 verwendet. Wir bezeichnen diese Variante in unserem Framework als *DC3-Lite*.

Im Folgenden gehen wir auf die Unterschiede der einzelnen Phasen zwischen dem *DC3* und dem *DC3-Lite* ein. Da das Induzieren analog zum *DC3* ist, wird diese Phase hier nicht genauer betrachtet.

Erste Phase Ähnlich wie im *DC3* werden in der ersten Phase die Triplets $\mathsf{T}[i, i+2]$ sortiert und lexikographische Namen vergeben. Im Unterschied zum *DC3* werden aber alle Positionen i von T sortiert. Dies machen wir, um den Eingabetext mit den lexikographischen Rängen zu überschreiben, damit für T_0 und T_{12} kein zusätzlicher Speicherbereich benötigt wird. Formell berechnet sich der überschriebene Text T_{new} durch

$$\mathsf{T}_{\text{new}} = [t_i | i \text{ modulo } 3 = 0] \circ [t_i | i \text{ modulo } 3 = 1] \circ [t_i | i \text{ modulo } 3 = 2]$$

Dadurch bleiben die Informationen des ursprünglichen Textes erhalten, da in jedem lexikographischen Rang die Information des Zeichens an der betrachteten Position berücksichtigt wurde. In jedem Zeichen sind sogar mehr Informationen enthalten, da die beiden darauf folgenden Zeichen ebenfalls berücksichtigt werden.

Zunächst werden die Positionen von T in das Array U geschrieben und mithilfe der Inplace-Variante des Radixsort für große Alphabete aus Abschnitt 4.1.5 sortiert. Dabei wird der Speicherbereich für das SA als Bucket-Array verwendet.

Anschließend werden die lexikographischen Ränge vergeben und damit T_{new} berechnet. Damit für den rekursiven Aufruf die lexikographischen Ränge durch die Textlänge beschränkt sind, berechnen wir parallel auch T_{12} und schreiben dies in die Positionen von T_{new}, welche den Positionen i mit i modulo $3 = 1$ und i modulo $3 = 2$ entsprechen. Die Positionen i von T_{new} mit i modulo $3 = 1$, werden vorher im ersten Drittel von U und die Positionen mit i modulo $3 = 2$ im ersten Drittel von SA zwischengespeichert.

Der rekursive Aufruf erfolgt dann mit T_{12} und dem nicht verwendeten Speicher von U und SA. Dadurch wird das Suffix-Array SA_{12} von T_{12} berechnet und die zwischengespeicherten Positionen von T_{new} werden wieder zurückgeschrieben.

Dritte Phase Durch das Induzieren in der zweiten Phase wurde das SA_0 berechnet. In dieser Phase werden nun SA_0 und SA_{12} vereinigt. Dazu werden das ISA_0 und das ISA_{12} hintereinander in U berechnet. Im Unterschied zum *DC3* wird das vereinigte Suffix-Array nicht direkt in einen neuen Speicherbereich geschrieben, sondern die Einträge von SA_0 und SA_{12} werden mit den Positionen im vereinigten Suffix-Array überschrieben.

Beim Vereinigen werden jeweils die Suffixe an den Positionen $\mathsf{SA}_0[i]$ und $\mathsf{SA}_{12}[j]$ miteinander verglichen. Wenn $\mathsf{T}[\mathsf{SA}_0[i]]$ und $\mathsf{T}[\mathsf{SA}_{12}[j]]$ ungleich sind, kann die Position im vereinigten Suffix-Array direkt bestimmt werden. Ansonsten kann die Ordnung der Suffixe durch Nachschauen in ISA_0 und ISA_{12} bestimmt werden. Falls $\mathsf{SA}_{12}[i]$ modulo $3 = 1$ gilt, kann die Ordnung der Suffixe durch einen Vergleich von $\mathsf{ISA}_{12}[\mathsf{SA}_0[i]]$ und $\mathsf{ISA}_{12}[p_2 + \mathsf{SA}_{12}[j]]$ bestimmt werden, wobei p_2 die Startposition der Menge aller Positionen i mit i modulo $3 = 2$ in ISA_{12} ist. Falls $\mathsf{SA}_{12}[i]$ modulo $3 = 2$ gilt, wird die Ordnung der Suffixe durch einen Vergleich von $\mathsf{ISA}_{12}[p_2 + \mathsf{SA}_0[i]]$ und $\mathsf{ISA}_{12}[\mathsf{SA}_{12}[j] - p_2 + 1]$ bestimmt.

Anschließend werden ISA_0 und ISA_{12} mit den berechneten Positionen im vereinigten Suffix-Array, die in SA_0 und SA_{12} stehen, verknüpft. Genauer gesagt wird $\mathsf{ISA}_0[i] = \mathsf{SA}_0[\mathsf{ISA}_0[i]]$ und $\mathsf{ISA}_{12}[i] = \mathsf{SA}_{12}[\mathsf{ISA}_{12}[i]]$ gesetzt. Da ISA_0 und ISA_{12} hintereinander in U gespeichert wurden, steht in U nun das inverse Suffix-Array $\mathsf{ISA}_{\text{new}}$ von T_{new}. Um SA_{new} zu berechnen wird das Inverse von U in SA berechnet.

Bis jetzt haben wir nur das Suffix-Array SA_{new} des überschriebenen Eingabetextes berechnet. Da wir die Positionen des ursprünglichen Eingabetextes umsortiert haben, müssen wir diese wieder in die korrekte Reihenfolge bringen. Dies lässt sich durch einen Durchlauf mit der folgenden Funktion erreichen, wobei $m_0 = |\mathsf{T}_0|$ und $m_{12} = |\mathsf{T}_{12}|$.

$$\mathsf{SA}[i] = \begin{cases} 3\mathsf{SA}_{\text{new}}[i] & \text{für } \mathsf{SA}_{\text{new}}[i] \in [0, m_0) \\ 3(\mathsf{SA}_{\text{new}}[i] - m_0) + 1 & \text{für } \mathsf{SA}_{\text{new}}[i] \in [m_0, m_0 + \lceil \frac{m_{12}}{2} \rceil)) \\ 3(\mathsf{SA}_{\text{new}}[i] - m_1) + 2 & \text{für } \mathsf{SA}_{\text{new}}[i] \in [m_0 + \lceil \frac{m_{12}}{2} \rceil), m_0 + m_{12}) \end{cases}$$

Parallel

In diesem Kapitel wird der *DC3* - Algorithmus auf Möglichkeiten zur Parallelisierung untersucht. Dabei gehen wir auf jede einzelne Phase des Algorithmus ein, wobei das Augenmerk auf die dritte Phase – dem Mergen – gelegt wird.

Phase 1 In der ersten Phase werden die Substrings $\mathsf{T}[i, i+2]$ sortiert, wobei i ein Element aus dem *Difference Cover* $D = \{1, 2\}$ ist. Diese Phase haben wir in unserer Implementierung in zwei Teile aufgeteilt. Zuerst wird eine Liste von Triplets aufgestellt und diese wird dann im zweiten Teil der Phase sortiert. Diese zwei Teile lassen sich naiv parallelisieren. Die Aufstellung der Triplets kann mithilfe von *OpenMP* parallelisiert werden und für die Sortierung dieser Triplets wird statt eines sequentiellen Sortieralgorithmus ein paralleler Sortier-Algorithmus verwendet. Wir haben uns für den parallelen Standard-Sortieralgorithmus der *C++*-Bibliothek entschieden, da dieser im Vergleich zum parallelen IPs^{4}o auf allen Eingabetexten ohne Fehler durchgelaufen ist.

Phase 2 In der zweiten Phase des Algorithmus wird die Reihenfolge der Suffixe S_i bestimmt, wobei i kein Element aus dem *Difference Cover* $D = \{1, 2\}$ ist. Wie bereits im Kapitel 6.4.3 beschrieben worden ist, kann dieser Schritt mithilfe der Induzierung durchgeführt werden. Dafür werden Tupel bestehend aus einem Zeichen $\mathsf{T}[i]$ und dem Rang des nachfolgenden Suffixes S_{i+1} aufgestellt. Die jeweiligen Ränge sind dem zuvor berechneten inversen Suffix-Array zu entnehmen. Diese Tupel lassen sich anschließend sortieren. Dementsprechend lässt sich die zweite Phase ebenfalls naiv parallelisieren. Für die Berechnung des inversen Suffix-Arrays lässt sich die Schleife mit *OpenMP* parallelisieren. Die Schleife für die anschließende Aufstellung der Tupel parallelisieren wir ebenfalls mithilfe von *OpenMP*. Für die Sortierung dieser Tupel verwenden wir – wie bereits in der ersten Phase – den parallelen Standard-Sortieralgorithmus der *C++*-Bibliothek.

Phase 3 Die dritte Phase des *DC3* - Algorithmus lässt sich hingegen nicht naiv parallelisieren, da die Reihenfolge der Ausführungen in der Variante, die im Kapitel 6.4.3 vorgestellt worden ist, vorgegeben ist. Daher werden wir uns in diesem Kapitel mit verschiedenen Merge-Verfahren auseinandersetzen, die parallelisiert werden können.

Verallgemeinerter Ansatz Ein Merge-Verfahren, das sich gut parallelisieren lässt, ist bereits im Kapitel 6.4.4 vorgestellt worden. Der zweite Ansatz in dem Kapitel lässt sich jedoch nicht nur in dem *DC7* – Algorithmus, sondern in allen *Difference Cover* – Algorithmen anwenden. Dafür ziehen wir das Beispiel T = caabaccaabacaa\$ ran.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\mathsf{T}[i]$	c	a	a	b	a	c	c	a	a	b	a	c	a	a	\$

Aus den ersten zwei Phasen sind die jeweiligen Reihenfolgen der Suffixe S_i, wobei i ein Element aus dem *Difference Cover* $D = \{1, 2\}$ ist, und S_j, wobei j kein Element aus dem *Difference Cover* $D = \{1, 2\}$ ist, bekannt: $\mathsf{SA}_0 = [12, 9, 3, 6, 0]$ und $\mathsf{SA}_{12} = [14, 13, 7, 1, 8, 2, 10, 4, 11, 5]$. Das Mergen dieser beiden Mengen in diesem Verfahren unterteilen wir in drei Schritte. Zuerst wird die Menge SA_{12} in die zwei Mengen SA_1, SA_2 aufgeteilt und anschließend alle Mengen zu einer Menge $\mathsf{SA}_{012} = \mathsf{SA}_0 \circ \mathsf{SA}_1 \circ \mathsf{SA}_2$ konkateniert:

$\mathsf{SA}_{012} = [12, 9, 3, 6, 0, 13, 7, 1, 10, 4, 14, 8, 2, 11, 5]$. Diese Menge wird im zweiten Schritt dieses Verfahrens stabil nach den ersten drei Zeichen mit dem parallelen Standard-Sortieralgorithmus der *C++*-Bibliothek sortiert:

$\mathsf{SA}_{012} = [14, 13, 12, 7, 1, 8, 2, 10, 4, 9, 3, 6, 0, 11, 5]$. Zuletzt können die nun noch gleichen Triplets miteinander verglichen werden. Die Triplets $\mathsf{T}[7, 9]$ und $\mathsf{T}[1, 3]$ beinhalten jeweils die gleichen Triplets. Daher müssen wir die Ränge der Suffixe S_8 und S_2 miteinander vergleichen und erhalten dadurch, dass $\mathsf{S}_7 < \mathsf{S}_1$. Unter anderem sind auch die Triplets $\mathsf{T}[6, 8]$, $\mathsf{T}[0, 2]$ und $\mathsf{T}[11, 13]$ gleich. Auch hier werden dann die Ränge untereinander verglichen und erhalten, dass $\mathsf{S}_{11} < \mathsf{S}_6 < \mathsf{S}_0$. Somit muss die Reihenfolge dieser drei Indizes in SA_{012} getauscht werden. Die Vergleiche der jeweils gleichen Triplets können unabhängig voneinander durchgeführt werden, sodass sich dieser Schritt gut parallelisieren lässt. Die vorhandenen Threads können sich die Bereiche der gleichen Triplets aufteilen, sodass diese gleichzeitig verglichen und die zugehörigen Indizes gegebenenfalls vertauscht werden. Am Ende des Verfahrens erhalten wir das korrekte Suffix-Array $\mathsf{SA} = [14, 13, 12, 7, 1, 8, 2, 10, 4, 9, 3, 11, 6, 0, 5]$.

Parallele Suche In diesem Absatz stellen wir die *parallele Suche* auf sortierten Arrays vor. [28] Dabei handelt es sich um eine Verallgemeinerung der binären Suche auf eine beliebige Anzahl p an Threads.

Sei A das sortierte Eingabe-Array mit $n = |\mathsf{A}|$ und k der Schlüssel, nach dem wir suchen. Ähnlich wie bei der binären Suche wählen wir bestimmte Elemente x in A und vergleichen diese mit k, um zu entscheiden, ob wir links oder rechts von x weitersuchen. Um dies effizient zu parallelisieren, wählen wir p Elemente x_i an den Positionen $\frac{n(i+1)}{p+1}$ aus mit $i = 0, ..., p-1$. Dadurch werden $p+1$ Segmente in A induziert. Wir prüfen für jedes Element x_i, ob $k < x_i$ gilt und speichern die Ergebnisse dieser Vergleichsoperationen in ein Array C der Größe p. Nun wollen wir entscheiden, in welchem Segment von A sich der Schlüssel k befindet. Für dieses Segment muss gelten, dass das Element an der rechten Grenze größer als k ist und das Element an der linken Grenze kleiner als k ist. Wir müssen in C also nach der Position i suchen für die gilt $\mathsf{C}[i-1] = 0$ und $\mathsf{C}[i] = 1$, falls $1 \leq i \leq p-1$. Dann ist das Segment durch das Intervall $[\frac{ni}{p+1}, \frac{n(i+1)}{p+1}]$ gegeben. Zusätzlich müssen noch die Sonderfälle für das erste und letzte Segment betrachtet werden: Ist $\mathsf{C}[0] = 1$, ist das Segment, in dem sich k befindet, durch $[0, \frac{n}{p+1}]$ gegeben und ist $\mathsf{C}[p-1] = 0$, ist das Segment durch $[\frac{n(p-1)}{p+1}, |\mathsf{A}| - 1]$ gegeben. Diese Bedingungen lassen sich parallelisiert testen. Auf dem Segment lässt sich nun rekursiv die parallele Suche fortsetzen bis das Segment maximal p Elemente enthält. Dann lässt sich parallelisiert jedes Element mit k vergleichen und die Position von k in A wird zurückgegeben.

Da wir in jedem Rekursionsschritt das Eingabe-Array in $p+1$ Segmente aufteilen, bricht die Suche nach $log_{p+1}(n) = \frac{log(n)}{log(p+1)}$ Rekursionsschritten ab. Da in jedem Rekursionsschritt jeder Thread eine konstante Anzahl an Schritten ausführt, ist die Laufzeit in $O(\frac{log(n)}{log(p+1)})$.

Merge ab einer großen Anzahl von Prozessoren (Theorem 2) Nach einer wissenschaftlichen Arbeit von Kruskal [28, p. 943,944] unter dem zweiten Theorem gibt es einen Ansatz für ein paralleles Merge-Verfahren, das für eine große Anzahl von Prozessoren geeignet ist. Dieses Verfahren baut auf dem Algorithmus in [55] auf. Da dieses Verfahren von Kruskal sehr theoretisch beschrieben worden ist, demonstrieren wir dieses an dem Beispiel T = caabaccaabacaa$. Aus den ersten beiden Phasen haben wir die bereits sortierten Mengen $\mathsf{SA}_0 = [12, 9, 3, 6, 0]$ und $\mathsf{SA}_{12} = [14, 13, 7, 1, 8, 2, 10, 4, 11, 5]$ gegeben. Der Algorithmus besteht aus vier Abschnitten. Außerdem werden drei Variablen am Anfang des Verfahrens gesetzt.

1. M : Anzahl der Elemente aus SA_0
2. N : Anzahl der Elemente aus SA_{12}
3. k : beeinflusst, in wieviele Bereiche wir die Mengen aufteilen

In unserem Fall bedeutet das, dass $M = 10$ und $N = 5$ ist. Außerdem sieht Kruskal $k = 3$ als Richtwert an.

In der ersten Phase dieses Algorithmus werden alle Indizes der Menge SA_0 markiert mit $i \cdot \lceil M^{1/k} \rceil$, wobei $i = 1, 2, ...$. In unserem Fall werden daher das zweite und vierte Element der Menge SA_0 markiert.

In Phase 2 wird die Anzahl der benötigten Prozessoren bestimmt. In unserem Fall werden $\lfloor N^{1/k} \rfloor = 2$ Prozessoren pro markiertes Element bereitgestellt.

In der nachfolgenden dritten Phase werden die Positionen ermittelt, die die jeweiligen markierten Elemente aus der ersten Phase in der Menge SA_{12} hätten. Wichtig ist in diesem Fall, dass dies unabhängig voneinander erfolgt. Da die Menge SA_{12} bereits sortiert ist, kann zum Beispiel mithilfe der parallelen Suche aus dem vorherigen Kapitel die jeweilige Position ermittelt werden. In unserem Beispiel müssen wir das zweite und vierte Element der Menge SA_0 untersuchen. Das zweite Element der Menge SA_0 beschreibt das Suffix S_9. In der Menge SA_{12} hätte dieses Suffix die Position 9. Das Suffix S_6 an vierter Stelle der Menge SA_0 hätte die Position 10 in der Menge SA_{12}. Da die jeweilige Suche der Position unabhängig voneinander erfolgt, kann diese Aufgabe problemlos auf verschiedene Prozessoren aufgeteilt werden.

In der vierten Phase des Merge-Verfahrens werden sogenannte Segmente aufgestellt. Die Segmente stellen die Bereiche dar, in der die nun noch übrigen Elemente aus der Menge SA_0 einsortiert werden müssen. Aus der vorherigen Phase erhalten wir die Segmente $Y_1 = [0, 8]$, $Y_2 = [9, 9]$, $Y_3 = [10, 10]$. Dieser Schritt wird rekursiv aufgerufen, da es sich erneut um einen Merge handelt. Die Abbruchbedingung der Rekursion ist erfüllt, wenn mindestens eine der Mengen nur noch ein Element enthält. Dann genügt es, dieses eine Element in die andere Menge zu sortieren. In unserem Beispiel erhalten wir die Tupel ($\{12\}$, $\{14, 13, 7, 1, 8, 2, 10, 4\}$), ($\{3\}$, $\{11\}$) und ($\{0\}$, $\{5\}$). Die jeweiligen Mengen an erster Stelle der Tupel müssen nun in die jeweiligen Mengen an zweiter Stelle einsortiert werden. Dementsprechend ist jeweils die Abbruchbedingung erfüllt. Das Suffix S_{12} gehört zwischen die Suffixe S_{13} und S_7. Das Suffix S_3 gehört vor das Suffix S_{11} und das Suffix S_0 gehört vor S_5. Letztendlich können wir alle Schritte zusammenfügen und erhalten das Suffix-Array $\mathsf{SA} = [14, 13, 12, 7, 1, 8, 2, 10, 4, 9, 3, 11, 6, 0, 5]$.

Da diese Schritte unabhängig voneinander sind, kann dies auf verschiedene Prozessoren verteilt werden.

Merge von zwei Seiten In diesem Abschnitt beschreiben wir eine einfache Parallelisierung des naiven Merge-Algorithmus mit zwei Threads. Sei dazu n die Größe des Ausgabearrays C. Damit beide Threads unabhängig voneinander arbeiten, vereinigt der erste Thread von links insgesamt x Schritte und der andere Thread vereinigt von rechts insgesamt $n - x$ Schritte. Da der erste Thread in die ersten x Positionen von C und der zweite Thread an die letzten $n - x$ Positionen schreibt, arbeiten beide Threads unabhängig voneinander.

Theorem 7 - alles zusammen In diesem Abschnitt stellen wir einen parallelen Merge-Algorithmus vor, der effizient für eine konstante Anzahl an Threads p arbeitet. Dieser Algorithmus wurde zuerst in [55] beschrieben und die Beschreibung wurde in [28] weiter ergänzt. Dazu seien A und B die Eingabe-Arrays, die in das Ausgabe-Array C vereinigt werden sollen, mit $|\mathsf{A}| = n$, $|\mathsf{B}| = m$ und $|\mathsf{C}| = n + m$.

Wir werden den Algorithmus so erläutern, dass er mit einem beliebigen Vergleichsoperator arbeiten kann. Zur besseren Verständlichkeit verwenden wir in den Beispielen den Vergleichsoperator für natürliche Zahlen. In dem parallelen DC3 verwenden wir dann den bereits in Abschnitt 6.4.3 beschriebenen Vergleichsoperator.

Im Folgenden wird zunächst eine Übersicht über die Schritte des Algorithmus gegeben. Dieser Algorithmus arbeitet in sieben Schritten. In den ersten beiden Schritten werden insgesamt p Elemente in A und B markiert und die Reihenfolge dieser Elemente in C werden durch Mergen bestimmt. Durch die Wahl der markierten Elemente werden A und B in einzelne Segmente aufgeteilt. Um die markierten Elemente an die korrekten Positionen in C schreiben zu können, benötigen wir für jedes markierte Element x die Positionen in A und in B. Diese Positionen werden in den Schritten drei und vier bestimmt und die markierten Elemente an die korrekten Positionen in C geschrieben.

In den letzten drei Schritten werden die übrigen Elemente in A und B vereinigt, indem zunächst die Subsegmente in A und B bestimmt werden, die an die freien Stellen in C vereinigt werden sollen. Diese werden dann mit der Parallelisierung aus Abschnitt 6.4.4 vereinigt.

Nun werden die einzelnen Schritte genauer beschrieben. Im ersten Schritt markieren wir höchstens p Elemente in A und B in jeweils gleichen Abständen. Als Abstand d setzen wir $d = \lceil \frac{m+n}{p} \rceil$ und die markierten Elemente x_i sind dann an den Positionen $id - 1$.

Markierte Elemente werden durch Tupel (p_a, b, c) repräsentiert und in Arrays `marked_element_A` geschrieben, wenn x_i in A steht, bzw. `marked_element_B`, wenn x_i in B steht.

Mit p_a repräsentieren wir die Position des markierten Elements im Ursprungs-Array. b ist ein Indikator, wobei $b = 0$, wenn x_i in A steht und $b = 1$, wenn x_i in B steht und c repräsentiert die Nummer des markierten Elements im Urspungs-Array. Diese Werte werden benötigt, damit diese Informationen nach dem Mergen der markierten Elemente direkt bereit stehen.

In unserem Beispiel wollen wir die Arrays $\mathsf{A} = [2, 3, 5, 8, 13, 21, 34]$ und $\mathsf{B} = [4, 9, 16, 25, 36]$ mit vier Threads mergen. Der Abstand zwischen den Elementen ist $d = \frac{12}{4} = 3$. Wir markieren also in A die Elemente 5 und 21 und in B das Element 16. Abgespeichert als Tupel sehen die markierten Elemente wie folgt aus:

`marked_element_A`= [(2,0,0), (5,0,1)]
`marked_element_B`= [(2,1,0)]

Im zweiten Schritt werden die markierten Elemente in `marked_element_A` und in `marked_element_B` in ein Array `marked_element` vereinigt. Dabei wird der Merge-Algorithmus für eine große Anzahl an Prozessoren verwendet. Da es

höchstens p markierte Elemente gibt, haben wir genug Prozessoren, um diesen Algorithmus verwenden zu können.

In unserem Beispiel werden `marked_element_A` und `marked_element_B` vereinigt und als Ausgabe ergibt sich `marked_element` $= [(2,0,0),(2,1,0),(5,0,1)]$. Dies entspricht der sortierten Reihenfolge der markierten Elemente $[5,16,21]$.

Jetzt müssen wir zu jedem markierten Element x die Positionen p_A in A und p_B in B bestimmen. Um die folgenden beiden Schritte einfacher beschreiben zu können, nehmen wir an, dass x ein markiertes Element aus A ist. Für markierte Elemente aus B sind die Schritte analog.

Sei dazu x ein markiertes Element aus A. Die Position p_A ist dann trivialerweise die Position an der x in A steht. Dies haben wir an der ersten Stelle im Tupel von x gespeichert. Um die Position p_B bestimmen zu können, müssen wir in B nach der Position suchen, an der sich x in B einordnen lässt. Damit wir uns eine Suche über das ganze Array B ersparen, nutzen wir die Ergebnisse aus Schritt zwei, um das Segment zu bestimmen, in dem wir weiter nach x suchen. Dazu betrachten wir die sortierte Reihenfolge der markierten Elemente aus Schritt zwei. Weiterhin betrachten wir ein markiertes Element aus B der Form $x = (p_a, 1, c)$ und das darauf folgende markierte Element $x' = (p'_a, 1, c')$ aus B. Wir beobachten, dass alle markierten Elemente $(q_a, 0, e)$ aus A, die in der sortierten Reihenfolge zwischen x und x' stehen, im Segment zwischen den markierten Elementen p_a und p'_a in B eingeordnet werden müssen. Mit dieser Beobachtung können wir nun parallel die Segmente aus B bestimmen, in denen die markierten Elemente aus A vorkommen und umgekehrt.

Dazu erstellen wir zwei Hilfs-Arrays pos_A und pos_B. In pos_A speichern wir die Positionen der markierten Elemente in der sortierten Reihenfolge aus Schritt zwei und analog für pos_B. Da wir zu jedem markierten Element die Nummer in A bzw. B speichern, können wir pos_A und pos_B direkt bestimmen. Anschließend betrachten wir zwei direkt aufeinanderfolgende Elemente $x = (p_a, 1, c)$ und $x' = (p'_a, 1, c')$ in pos_B. Aus unserer Beobachtung können wir schließen, dass alle Elemente zwischen x und x' in der sortierten Reihenfolge aus Schritt zwei im Segment zwischen p_a und p'_a eingeordnet werden müssen. Daher können wir parallel für jedes dieser Elemente das Segment $[p_a + 1, p'_a)$ speichern. Analog lässt sich dieser Schritt für pos_A durchführen.

In unserem Beispiel kommen die markierten Elemente von A in der sortierten Reihenfolge an den Positionen 0 und 2 vor und das markierte Element von B an der Position 1. Also ist $pos_A = [0, 2]$ und $pos_B = [1]$. Wir betrachten jetzt je zwei aufeinanderfolgende Elemente in pos_A, hier also 0 und 2. Dies bedeutet, dass in der sortierten Reihenfolge alle Elemente zwischen den Positionen 0 und 2 zum einen aus B kommen und dass diese im Segment zwischen den dazugehörigen markierten Elementen liegen. Also können wir schließen, dass das markierte Element 16 im Segment $[3, 5)$ in A eingeordnet werden müsste. In pos_B kommt nur das Element 1 vor. Für alle Elemente, die davor oder dahinter in der sortierten Reihenfolge vorkommen, können wir dennoch das Segment bestimmen. Dazu ordnen wir dem ersten Element der sortierten Reihenfolge das Segment $[0, 2)$ und dem letzten das Segment $[3, 5)$ in B zu.

Im vierten Schritt können wir nun für jedes markierte Element in dem in Schritt drei bestimmten Segment eine binäre Suche durchführen um die Position p_B zu erhalten. Diese Position wird in einem Array `pos_in_other_A` gespeichert, da wir diese Position im nächsten Schritt benötigen. Mit p_A und p_B lässt sich die Position der markierten Elemente in C durch $p_A + p_B$ berechnen.

Im Beispiel kommt das markierte Element 5 in A an der Position 2 vor. Mittels einer binären Suche im Segment $[0, 2)$ in B können wir die Position 1 berechnen, an der 5 in B eingeordnet werden müsste. Also müssen wir 5 in C an die Position 3 schreiben. Das markierte Element 16 steht in B an der Position 2 und mit einer binären Suche bestimmen wir die Position 5 an der 16 in A eingeordnet werden müsste. In C steht die 16 also an der Position 7. Für das markierte Element lässt sich analog die Position 8 in C berechnen. Unser Ausgabe-Array C sieht nach diesem Schritt also wie folgt aus:

i	0	1	2	3	4	5	6	7	8	9	10	11
C$[i]$				5				16	21			

Nach Schritt vier haben wir alle markierten Elemente an die korrekte Position in C geschrieben. Nun müssen wir noch alle übrigen Elemente mergen. Wir haben nun in C mehrere Segmente zwischen den markierten Elementen, die noch nicht beschrieben wurden. Wir beobachten, dass in einem freien Segment, welches durch x und x' induziert wird, nur diejenigen Elemente in A und B vorkommen können, die in A und B ebenfalls zwischen x und x' liegen. Durch die markierten Elemente und die in Schritt vier bestimmten Positionen der markierten Elemente im jeweils anderen Array ergibt sich somit eine Aufteilung in Subsegmente, die in ein freies Segment in C vereinigt werden sollen.

Im fünften Schritt werden wir also zunächst A und B in Subsegmente aufteilen. Dazu bestimmen wir zu Array A die Positionen der markierten Elemente `pos_marked_A` und mergen dies mit `pos_in_other_B` mit dem Mergealgorithmus aus für eine große Anzahl an Prozessoren, um die Grenzen der Subsegmente zu erhalten. Daraus können wir die Subsegmente direkt bestimmen, da die Segmentgrenzen durch je zwei aufeinanderfolgende Positionen im vereinigten Array gegeben sind. Hier ist zu beachten, dass die markierten Elemente nicht in einem Subsegment enthalten sein dürfen.

Im Beispiel sind die markierten Elemente von A an den Positionen $[2, 5]$ und das markierte Element 16 von B wurde in A an die Position [5] eingeordnet. Wenn wir beide Arrays mergen, ergibt sich das Array $[2, 5, 5]$, wobei die ersten beiden Elemente die markierten Elemente von A sind. Nun bestimmen wir die Subsegmente von A. Dabei dürfen die markierten Elemente von A nicht in den Subsegmenten enthalten sein. Die Subsegmente sind also $[[0, 2), [3, 5), [6, 7)]$. Analog ergeben sich die Subsegmente von B als $[[0, 1), [1, 2), [3, 5)]$.

Nun wollen wir die einzelnen Subsegmente aus A und B parallel mit dem in Abschnitt 6.4.4 beschriebenen Algorithmus in die richtigen Segmente in C mergen. Unser Ziel ist es, dass jeder Thread möglichst gleich viele Schritte beim Mergen benötigt. Dazu übernimmt jeder Thread ein Segment der Größe höchstens $s \leq \lceil \frac{m+n}{p} \rceil$ in A und B und ist dafür zuständig s Elemente zu mergen. Falls

es in A und B Segmente gibt, die nicht durch ein markiertes Element beendet werden, werden beide von einem Thread übernommen.

Wir berechnen zunächst für jeden Thread ein *Schedule*. Ein Schedule ist eine Liste von *Merge-Aktionen*, die festlegen, welche Subsegmente vereinigt werden und ob diese von links oder rechts vereinigt werden sollen. Genauer ist eine Merge-Aktion ein Tupel $(\mathsf{I}_A, \mathsf{I}_B, \mathsf{I}_C, d, l)$, wobei I_A ein Subsegment aus A, I_B ein Subsegment aus B und I_C ein freies Segment in C ist, in das I_A und I_B vereinigt werden sollen. d ist ein Indikator, wobei $d = 0$, wenn die Merge-Schritte von links, und $d = 1$, wenn die Merge-Schritte von rechts ausgeführt werden sollen. l bezeichnet die Anzahl der Merge-Schritte, die ausgeführt werden sollen.

Nun beschreiben wir, wie ein solches Schedule berechnet werden kann. Wenn wir das i-te Subsegment A_i und B_i aus A und B betrachten, lässt sich beobachten, dass beide in das gleiche freie Segment aus C vereinigt werden. Wir müssen also die Thread-Nummern j_A und j_B bestimmen, die für A_i und B_i zuständig sind. Dann ist der Thread j_A dafür zuständig $|\mathsf{A}_i|$ Schritte von links zu mergen und der Thread j_B ist dafür zuständig $|\mathsf{B}_i|$ Schritte von rechts zu mergen. Wir setzen $d' = \lceil \frac{m+n}{p} \rceil$ als Abstand zwischen den markierten Elementen und r_A ist die rechte Grenze von A_i und r_B ist die rechte Grenze von B_i. Dann lässt sich j_A berechnen durch $j_A = \lfloor \frac{r_A}{d'} \rfloor$ und j_B durch $j_B = \lfloor \frac{r_B}{d'} + m_A \rfloor$, wobei m_A die Anzahl der markierten Elemente in A ist. Falls $r_A = |\mathsf{A}| - 1$, ist A_i ein Segment, welches nicht durch ein markiertes Element abgeschlossen wird, und setzen $j_A = p - 1$. Analog für B.

Nun können wir für Thread j_A die Merge-Aktion $(\mathsf{A}_i, \mathsf{B}_i, \mathsf{C}_i, 0, |\mathsf{A}_i|)$ und für Thread j_B die Merge-Aktion $(\mathsf{A}_i, \mathsf{B}_i, \mathsf{C}_i, 1, |\mathsf{B}_i|)$ zum Schedule hinzufügen. C_i ist das Segment in C, in das A_i und B_i vereinigt werden. Dieses lässt sich direkt aus den Segmentgrenzen von A_i und B_i bestimmen.

Im Beispiel betrachten wir die Subsegmente $[0, 2)$ von A und $[0, 1)$ von B. Diese müssen in das freie Segment $[0, 3)$ von C vereinigt werden. Es ergeben sich mit $d' = 3$ und $m_A = 2$ die Thread-Nummern $j_A = \lfloor \frac{2}{3} \rfloor = 0$ und $j_B = \lfloor \frac{1}{3} \rfloor + 2 = 2$. Also fügen wir zu Thread 0 die Merge-Aktion $([0, 2), [0, 1), [0, 3), 0, 2)$ und zu Thread 2 die Merge-Aktion $([0, 2), [0, 1), [0, 3), 1, 1)$ hinzu. Am Ende ergibt sich für jeden Thread das folgende Schedule:

```
Thread 0:  [([0,2), [0,1), [0,3), 0, 2)]
Thread 1:  [([3,5), [1,2), [4,7), 0, 2)]
Thread 2:  [([0,2), [0,1), [0,3), 1, 1), ([3,5), [1,2), [4,7), 1, 1)]
Thread 3:  [([6,7), [3,5), [9,12), 0, 1), ([6,7), [3,5), [9,12), 1, 2)]
```

Im letzten Schritt müssen wir nun noch das berechnete Schedule aus Schritt sechs ausführen. Dazu iteriert jeder Thread über seine Liste der Merge-Aktionen. Wenn er aktuell die Merge-Aktion $(\mathsf{I}_A, \mathsf{I}_B, \mathsf{I}_C, 0, l)$ betrachtet, führt er die ersten l Merge-Schritte, in der I_A und I_B in I_C vereinigt werden, von links aus, und wenn er die Merge-Aktion $(\mathsf{I}_A, \mathsf{I}_B, \mathsf{I}_C, 1, l)$ betrachtet, führt er die ersten l Merge-Schritte von rechts aus.

Im Beispiel enthält das Ausgabe-Array C am Ende das korrekt vereinigte Array:

i	0	1	2	3	4	5	6	7	8	9	10	11
$\mathsf{C}[i]$	2	3	4	5	8	9	13	16	21	25	34	36

6.4.5 Optimierung und Evaluation

In diesem Kapitel wird auf verschiedene Optimierungen und deren Wirkungsweisen eingegangen. Außerdem werden generelle Vergleiche der Messergebnisse zwischen unserer Implementierungen des *DC3* – beziehungsweise *DC7* – Algorithmus und der Referenzimplementierung des *DC3* - Algorithmus, die der wissenschaftlichen Arbeit von Juha Kärkkäinen und Peter Sanders entnommen worden ist, gezogen [29, p. 954,955]. Dabei nehmen wir an, dass die Referenzimplementierung die Basisversion ist.

`Speicherverbrauch` Die Referenzimplementierung benötigt folgende Arrays:

1. T_{12} : lexikographische Namen der Suffixe in i modulo $3 \neq 0$
2. SA_{12} : sortierte Positionen der Suffixe i modulo $3 \neq 0$
3. T_0 : Positionen der Suffixe i modulo $3 = 0$
4. SA_0 : sortierte Positionen der Suffixe i modulo $3 = 0$

In unserer Implementierung benötigen wir jedoch nur drei Arrays. Und zwar können wir uns das Array T_0 sparen, da dieses nicht erst berechnet werden muss, sondern dem Sortieralgorithmus in Form einer *Compare*-Funktion übermittelt werden kann, ohne ein neues Array anlegen zu müssen. Ein Array der Länge $\frac{1}{3}n$ wird somit weniger verbraucht. Daher hat unsere Implementierung einen etwas besseren Speicherverbrauch als die Referenzimplementierung.

`Sortieralgorithmus` Die Referenzimplementierung von Juha Kärkkäinen und Peter Sanders verwendet als Sortieralgorithmus, die in den beiden ersten Phasen des *DC3* - Algorithmus benötigt werden, den *Radix-Sort*. Eine ähnliche Variante haben wir auch entwickelt, jedoch ist der Standard-Sortieralgorithmus der *C++*-Bibliothek schneller. Wir haben ebenfalls die sequentielle Variante des Algorithmus *In-place Parallel Super Scalar Samplesort (IPs^{4}o)* ausprobiert, jedoch keinen Unterschied weder bezüglich der Laufzeit noch des Speicherverbrauches feststellen können, sodass wir uns für den Standard-Sortieralgorithmus entschieden haben, damit wir dafür keine externe Bibliothek verwenden müssen. Aufgrund des besseren Sortieralgorithmus weist unsere Implementierung ebenfalls eine etwas bessere Laufzeit auf als die der Referenzimplementierung.

`DC7` Die Implementierung des *DC7* - Algorithmus ist sehr ähnlich zu der des *DC3* - Algorithmus. Es wird jedoch zusätzlich ein Array benötigt, um zuerst die Ränge der Suffixe beginnend in Position i modulo $7 = 5$ und anschließend die Ränge in i modulo $7 = 6$ zu speichern. Dieses Array kann jedoch anschließend gelöscht werden. Trotzdem ist der Speicherverbrauch niedriger als der unseres *DC3* - Algorithmus, da die Rekursion mit einem kleineren String aufgerufen wird. Dies wurde bereits bei den theoretischen Überlegungen angenommen.

Ein etwas größerer Unterschied der Implementierungen der beiden Algorithmen ist die dritte Phase – das Mergen. Hierbei haben wir beide Ansätze, die wir bereits im Kapitel 6.4.4 besprochen haben, implementiert und miteinander verglichen. Als Ergebnis erhielten wir, dass der naive erste Ansatz schneller ist als der zweite Ansatz. Dies liegt vermutlich daran, dass zuerst alle sortierten Mengen SA_0, SA_{124}, SA_3, SA_5 und SA_6 zuerst einmal zu einer Menge konkateniert werden müssen und anschließend eine Sortierung nach den ersten sieben Zeichen erfolgen muss, um dann erst mergen zu können. Dieser Ansatz hat zwar in der Theorie eine bessere Laufzeit, jedoch sind dies zu viele Schritte in der Praxis. Aus diesem Grund liegt der zweite Ansatz zwar in unserem SACABench- Framework vor, jedoch wird stattdessen nur der naive Ansatz bei den Auswertungen verwendet. Trotz der geringeren Rekursionstiefe weist der *DC7* - Algorithmus mit dem naiven Ansatz eine langsamere Laufzeit gegenüber dem *DC3* - Algorithmus auf. Diese lässt sich vermutlich auf das Mergen zurückführen, denn dies ist die Schwachstelle in dem Algorithmus.

6.5 Deep-Shallow

Mit dem SACA Deep-Shallow von Manzini und Ferragina aus ihrem Paper von 2004 [38] begann die Limitierung des Speicherverbrauches von SACAs. Da das Suffixarray besonders bei großen Texten interessant ist, ist der Speicherbedarf möglichst stark zu begrenzen. Das bedeutet, dass zusätzlich zu den n Byte, welche durch den Eingabetext belegt werden und den 4 bis 8 n Byte, welche das fertige Suffixarray speichern [1] nur wenig Hilfs-Speicherplatz belegt wird. Insgesamt hat der Speicherbedarf der Suffixarray-Konstruktion also eine untere Schranke von $5n$ Byte[2]. qsufsort (siehe Abschnitt 6.1) benutzt beispielsweise zusätzlich $4n$ Byte an Speicher [38]. Allerdings gibt es seit der Veröffentlichung des Papers auch viele Algorithmen mit konstantem oder keinem zusätzlichen Speicherverbrauch, wie beispielsweise SACA-K [42] (siehe Abschnitt 6.12).

Im hier vorliegenden Algorithmus kann durch einen Parameter bestimmt werden, wie viel zusätzlicher Speicherplatz gegen ersparte Laufzeit eingetauscht wird. Der im Paper verwendete Parameterwert kommt auf einen gesamten Speicherplatzverbrauch von $5.03n$ Byte[3]: je 100 MiB Eingabetext verwendet er lediglich 3 MiB Hilfsspeicher.

Ein Nachteil des hier vorgestellten Algorithmus besteht in seiner schlechten Worst-Case-Komplexität: Er braucht theoretisch maximal $\mathcal{O}(n^2 \log n)$ Rechenschritte, um das Suffixarray zu konstruieren. Allerdings konnten Manzini und Ferragina zeigen, dass das Verfahren in der Praxis dennoch gute Ergebnisse liefert [38].

Wir stellen hier zwei verschiedene Varianten vor: Die erste Variante (Deep-Shallow mit Big Buckets) ist aus dem Paper entnommen und wird im nächsten Teil vorrangig beschrieben. Die zweite Variante (Deep-Shallow mit Small Buckets, Abschnitt 6.5.4) wurde von uns selbst erdacht und ist besser für eine Parallelisierung geeignet. Dafür verzichtet sie jedoch auf jegliches Induzieren.

6.5.1 Überblick

Der Deep-Shallow SA Sortieralgorithmus besteht aus drei Teilen: Bucket Sort, Shallow Sort und Deep Sort. Außerdem findet nach jedem Shallow Sort ein Induzierschritt statt. Diese Teile werden nun erläutert und zuletzt mit einem Beispiel erklärt.

Bucket Sort (siehe Abschnitt 4.1.4) wird einmalig zu Beginn verwendet, dann wird jeder Bucket mit Shallow Sort sortiert und sobald erkannt wird, dass die zu sortierenden Suffixe einen vorher festgelegte LCP haben, wird Deep Sort verwendet.

[1] Unter den Annahmen, dass in der Eingabe nur maximal 256 verschiedene Zeichen vorkommen.

[2] Wenn man das Suffixarray mit 32-Bit Zahlen kodiert. Bei 40, 48 und 64 Bit Zahlen ist die untere Grenze jeweils höher.

[3] Bei Verwendung von 32-Bit Wörtern für das Suffix-Array.

```
def deep_shallow_sort(T: Text):
  Create a bucket for every character pair α, β ∈ Σ.
  Call them b_αβ.

  # Sort T into the buckets by comparing only the first two
  # characters of every string. Represent the suffixes by their
  # starting indices in T.
  bucket_sort(T)

  while (not all buckets are sorted) do:
    find the smallest unsorted bucket b_α.

    # Skip already sorted subbuckets b_ασ for every σ ∈ Σ.
    shallow_sort(b_α, 0)
    Mark A as sorted.

    # Use the copy technique to induce the order
    # of b_σα for every σ ∈ Σ.
    for σ ∈ Σ do:
      copy_technique(σ, α)
```

Algorithmus 6.6: Einstiegspunkt des Algorithmus

Teil 1: Bucket Sort

Die erste Stufe des Algorithmus erstellt $|\Sigma|^2$ Buckets. Diese enthalten jeweils alle Suffixe, die mit den gleichen zwei Buchstaben anfangen. Es gibt also für jede Buchstabenkombination $\alpha\beta$ einen Bucket $\mathsf{b}_{\alpha\beta}$. Allerdings muss nicht jeder Bucket Elemente enthalten. Es wird jeder Bucket als unsortiert markiert, wenn er mehr als ein Element enthält. Es gibt weiterhin Superbuckets b_α, welche alle Subbuckets $\mathsf{b}_{\alpha\sigma}$ für alle $\sigma \in \Sigma$ enthalten.

Die Buckets liegen sequentiell im Speicher des Suffixarrays, von links nach rechts sortiert. Daher erhält jeder Bucket einen festen Speicherbereich, in dem seine Elemente sortiert werden müssen. Kein Suffix muss nach diesem Schritt seinen Bucket wechseln. In Algorithmus 6.6 wird anhand eines Pseudocode der Einstiegspunkt des Algorithmus erklärt.

Teil 2: Shallow Sort

Der Hauptteil des Algorithmus wählt immer den kleinsten, nicht-sortierten Bucket aus und sortiert ihn mittels Shallow Sort. Dabei wird Multikey-Quicksort (siehe Abschnitt 4.1.3) verwendet, welches gut auf Strings funktioniert, die keine langen gemeinsamen Präfixe haben [38]. Der normale MKQS-Algorithmus wird so abgewandelt, dass bei einer Vergleichstiefe größer einem Schwellwert automatisch abgebrochen wird und die verbleibenden Partitionen jeweils mit Deep Sort sortiert werden.

Teil 3: Deep Sort

Wird die Rekursion im Shallow Sort zu tief, bedeutet das, dass im zu sortierenden Bucket viele Strings mit einem gemeinsamen Präfix liegen. Daher wird ein Verfahren verwendet, welches diese Strings schnell sortieren kann. Genauer gesagt werden drei Verfahren verwendet, je nachdem welches anwendbar ist. Diese werden insgesamt Deep Sort genannt.

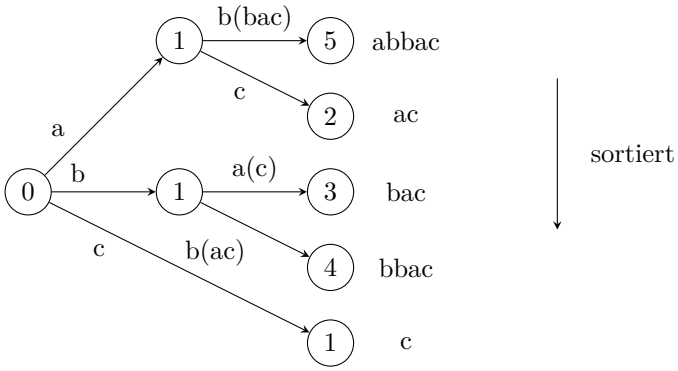


Abbildung 6.10: Ein Beispiel für einen Blind Trie, welcher fünf Suffixe enthält. Die Kanten sind jeweils nur mit dem ersten Zeichen beschriftet. Die Zeichen in Klammern sind nur implizit gespeichert. Diese Information ist in der Zahl im nachfolgenden Knoten enthalten.

Das so genannte *Blind Sort* verwendet einen Blind Trie (siehe Abbildung 6.10), in welchen die Suffixe eingefügt werden. Danach ist es einfach, den Trie in lexikografisch korrekter Reihenfolge zu durchlaufen. Da der Trie allerdings einen relativ großen Speicherbereich belegt (36 Byte je String [38]), wird er nur für bis zu M zu sortierende Strings verwendet.

Wenn es mehr als M Strings gibt, die es zu sortieren gilt, wird erneut Quicksort [8] verwendet, allerdings werden diesmal ganze Suffixe verglichen. Die Rekursion verwendet Blind Sort, wenn die zu sortierende Menge klein genug ist.

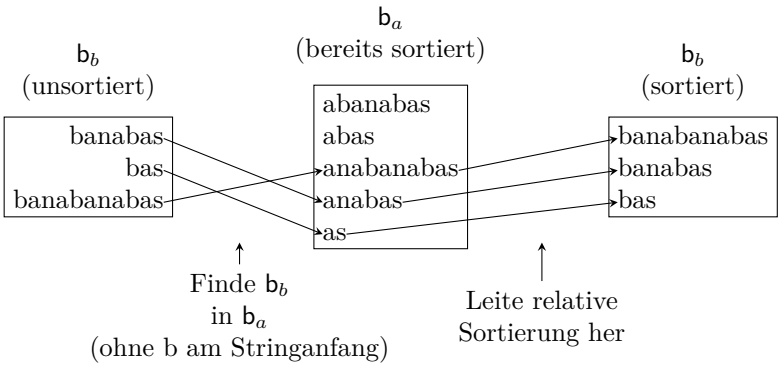


Abbildung 6.11: Funktionsweise von Induced Sort am Beispiel von „banabanabas“. Der linke Bucket b_b (zur Vereinfachung gibt es nur Buckets für einzelne Buchstaben) soll mithilfe von b_a sortiert werden. Dazu wird das LCP von allen Einträgen in b_b berechnet („ba“). Da „a“ in „ba“ enthalten ist, kann b_a benutzt werden, um b_b zu sortieren. Dazu wird b_a durchlaufen um die Reihenfolge aller Elemente von b_b zu bestimmen.

Induced Sort

Diese beiden Verfahren nutzen die Struktur der zu sortierenden Strings (sie sind Suffixe eines gemeinsamen Textes) nicht aus. Allerdings wird außerdem ein drittes Verfahren verwendet, welches *Induced Sort* heißt. Dieses verwendet bereits sortierte Buckets, um eine Menge von Strings zu sortieren. Dazu müssen die Strings ein gemeinsames Präfix haben. Da Induced Sort nur innerhalb von Deep Sort verwendet wird, ist diese Bedingung gegeben. Siehe zur Erläuterung Abbildung 6.11.

Dieses Sortierverfahren ist auch ohne zusätzlichen Speicherplatz möglich. Allerdings ist es dann schwierig, die Positionen der unsortieren Strings im sortierten Bucket zu finden. Daher wird der Text in (etwa) gleich große Segmente zerlegt und eine zusätzliche Datenstruktur gespeichert, die zu jedem Segment des Textes für das linkeste bereits sortierte Suffix speichert, in welchem Bucket es liegt und wo es in diesem Bucket liegt. Dadurch kann effizient überprüft werden, ob Induced Sort verwendet werden kann.

Die Anzahl der Segmente für Induced Sort kann durch einen Parameter d eingestellt werden, da sie einen direkten Einfluss auf den Speicherplatzverbrauch des Algorithmus haben. Im Paper verwendet wurde $d = 500$, welches zum oben genannten Speicherplatzverbrauch von $5.03n$ Byte führt. Je nach Benchmark führen kleinere d Werte (= mehr Speicherverbrauch) zu Verbesserungen der Laufzeit. Es liegt am Benutzer, die richtige Balance zwischen Laufzeit und Speicherverbrauch zu finden.

Falls Induced Sort nicht möglich ist, da kein passender Bucket gefunden wurde, können die anderen beiden Sortierverfahren weiterhin benutzt werden.

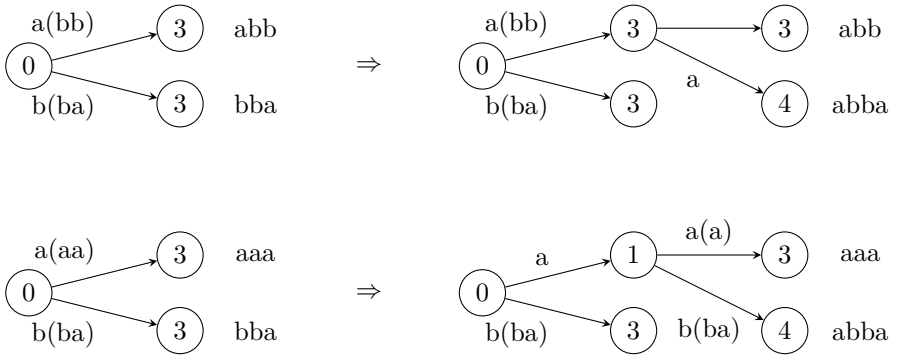


Abbildung 6.12: Zwei Fälle beim Einfügen in Blind Tries. Der einzufügende String ist „abba“. Die eingeklammerten Buchstaben sind nur implizit gespeichert.

6.5.2 Pseudocode

Nachdem die verwendeten Verfahren grob erklärt wurden, wird nun am folgenden Pseudocode der Algorithmus im Detail erläutert.

Zuerst werden Buckets für jede mögliche zweistellige Buchstabenkombination erstellt, wie bereits oben beschrieben. Diese werden nacheinander mit `shallow_sort` sortiert.

Blind Sort

Blind Sort basiert auf Blind Tries, in welche die zu sortierenden Strings eingefügt werden. Daraufhin kann der Baum In-Order durchlaufen werden, wodurch die Suffixe in lexikographischer Reihenfolge erhalten werden. An Abbildung 6.10 wurde ein solcher Blind Trie bereits beispielhaft erklärt.

Ein Blind Trie besteht aus Knoten und Kanten in einer Baumstruktur, wobei jeder Knoten mindestens zwei Kinder hat. Jede Kante ist mit einem Buchstaben beschriftet und jeder Knoten mit einem Index. Ein Knoten entspricht dabei einer Menge von Strings, die einen gemeinsamen Präfix haben. Die Zahl im Knoten gibt an, wie lang dieser gemeinsame Präfix ist. Die ausgehenden Kanten zeigen dann an, welches das Zeichen ist, durch das sie sich unterscheiden. Jedes Blatt entspricht einem Suffix, und ist durch einen Suffixindex gekennzeichnet.

Beim Einfügen wird jeder Kante gefolgt, die zum neuen String passt. Hierbei kann es sein, dass keine passende Kante existiert. Dann kann eine neue Kante eingefügt werden, die den Rest des Strings enthält (siehe Abbildung 6.12, oberer Fall). Der andere Fall wäre, dass zwar eine passende Kante existiert, der Index im folgenden Knoten allerdings zu groß ist und auch einen Teil enthält, der nicht mit dem String übereinstimmt. Dann muss in die Kante ein Knoten eingefügt werden, der den gemeinsamen Teil vom unterschiedlichen Teil trennt (siehe Abbildung 6.12, unterer Fall). Für Details über die Datenstruktur siehe

das Paper von Ferragina und Grossi [22], sowieso das Paper von Manzini und Ferragina [38].

Da die ausgehenden Kanten jedes Knoten alphabetisch sortiert sind, sind auch die Blätter lexikographisch sortiert. Mit einem Durchlauf des Baumes kann man an den Blättern die korrekte Sortierung ablesen.

Die Verwendung von Blind Sort ist auf ein konstantes M beschränkt, da der Baum $36M$ Byte an Speicher verbraucht. Manzini und Ferragina wählen $M = n/2000$. Dadurch beschränkt sich der Speicher durch den Trie auf $9n/500$ Byte [38].

Ternäres Quicksort

Für größere Buckets wird ternäres Quicksort verwendet. Dieses sortiert den Bucket mit simplen, direkten Vergleichen über die gesamten Suffixe. Es ist zu beachten, dass die mittlere Partition („=“) nur ein Element enthält, welches p ist. Daher muss diese Partition nicht sortiert werden. Statt einem rekursiven Aufruf an sich selbst wird jedoch wieder `deep_sort` aufgerufen, um von Blind Sort zu profitieren, falls die Partition nun klein genug ist. Für Details über ternäres Quicksort siehe Abschnitt 4.1.1 oder auch das Paper von Bentley und McIlroy [8].

Induced Sort

Die bisher vorgestellten Algorithmen verwenden keine Information darüber, dass es sich bei den sortierten Strings um Suffixe desselben Textes handelt. In diesem Abschnitt wird das Verfahren um einen Algorithmus ergänzt, welcher eine Menge von Strings sortieren kann, die ein gemeinsames Präfix haben.

Induced Sort benutzt die bereits sortierten Buckets, um zu sortieren. In Abbildung 6.11 findet sich eine Visualisierung. Die Vorraussetzung dafür ist, dass die Strings in A einen gemeinsamen Präfix haben, und dass es einen dazu passenden Bucket gibt. Zuerst wird das eigentliche Prinzip erklärt und danach eine Verbesserung eingeführt, welche die Performance auf Kosten von zusätzlichem Speicher erhöht.

Um herauszufinden, ob Induced Sort verwendet werden kann, wird der LCP berechnet und jeder Bucket, der in Frage kommt, überprüft. Wenn im LCP der Strings eine zweistellige Zeichenkette $\alpha\beta$ vorkommt, für die der Bucket $\mathsf{b}_{\alpha\beta}$ bereits sortiert wurde, so kann die Sortierung der Strings leicht aus der Ordnung des Buckets hergeleitet werden. Alle Zeichen vor $\alpha\beta$ werden abgeschnitten. Die so erhaltenen Strings werden im Bucket gesucht und markiert. Dann wird der Bucket in richtiger Reihenfolge durchlaufen und jeder markierte String zurückgerechnet (um die Länge des gemeinsamen Präfix) und in dieser Reihenfolge gespeichert. Da der Bucket sortiert war, sind folglich auch die Strings danach sortiert.

Die zeitintensivste Aufgabe ist laut Profiling, einen passenden Bucket zu finden [38]. Daher wird durch eine Verbesserung zusätzlicher Speicherplatz verwendet, um zwei Arrays, OFFSET und ANCHOR, anzulegen. Der Eingabetext

```
def induced_sort(A: Array<SuffixStartPos>, t: Integer,
                      B: Bucket):
  n_marked := 0
  for s in B do: # Do a pass through all the suffixes in B.
    # We need to ignore the first t characters.
    if (A + t) contains s:
      Mark s in B.
      n_marked++

    if n_marked == |A|:
      Pass over B and store the marked elements to A.
      Unmark every element of B.
      return
```

Algorithmus 6.7: Induced Sort, Version 1 [38]

wird in etwa gleich große Teile segmentiert. Die Segmentgröße ist dabei durch den Parameter d einstellbar. Durch n/d ergibt sich die Anzahl der Segmente.

In $\mathsf{Offset}[i]$ wird der linkeste (also kleinste) Suffix-Index je Segment gespeichert, der zu einem Bucket gehört, der bereits als sortiert markiert ist. Der Wert ist dabei relativ zum jeweiligen Segmentbeginn. In $\mathsf{ANCHOR}[i]$ wird dann die Position vom Suffix $\mathsf{OFFSET}[i]$ in seinem Bucket gespeichert. In Abbildung 6.13 findet sich eine Veranschaulichung der beiden Arrays.

Da man die Startposition von Segment i nicht speichern muss (da jedes Segment i das Intervall $[i \cdot d, (i+1) \cdot d)$ umfasst), und jedes Segment höchstens d Elemente enthält, ist $\mathsf{OFFSET}[i] < d$. Unter der Annahme $d < 2^{16}$ kann man $\mathsf{OFFSET}[i]$ in 2 Byte speichern. Da $\mathsf{ANCHOR}[i]$ die Position im Bucket enthält und der Bucket im Worst-Case alle Suffixe enthält, sind dort die vollen 4 Byte

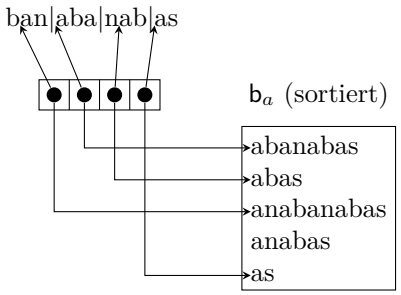


Abbildung 6.13: Verwendung der Arrays OFFSET und ANCHOR am Beispiel der Eingabe „banabanabas“. Die Eingabe wird in vier Segmente aufgeteilt und zu jedem Segment ein OFFSET und ein ANCHOR gespeichert. Das OFFSET entspricht jeweils dem Zeiger in den Eingabetext (oben). Der ANCHOR zeigt auf die Position dieses Suffix in seinem Bucket.

```
def induced_sort(A: Array<SuffixStartPos>, t: Integer,
                       B: Bucket):
  for a in A do:
    s := a mod d # Suffix a is from this segment number.
    if a < offset[s] <= a + L:
      bucket_pos = anchor[s]
      Induce Sort like above by scanning around bucket_pos.
      return

  # If no induced Sort is possible, use other methods.
  Use Deep Sort without Induced Sort.
```

Algorithmus 6.8: Induced Sort, Version 2 [38]

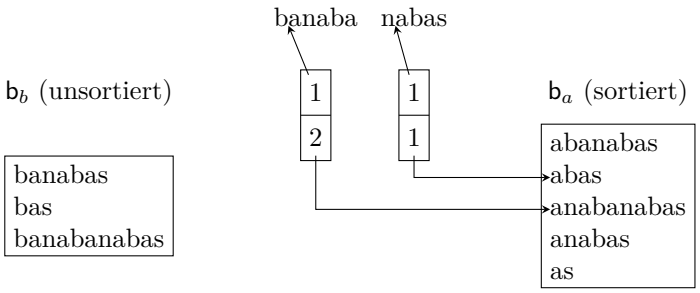


Abbildung 6.14: Ausgangs-Szenario im Beispiel. Betrachtet wird der String „banabanabas“, und der Bucket b_a ist bereits sortiert. b_b soll mittels Induced Sort sortiert werden.

nötig. Daher hat diese Erweiterung einen zusätzlichen Speicherbedarf von 6 Byte je Segment, also insgesamt $6n/d$ Byte.

Der Algorithmus verwendet OFFSET und ANCHOR (so genannte Anker), um einen passenden, bereits sortierten Bucket und die Position in jenem Bucket zu finden. In Algorithmus 6.8 findet sich Pseudocode des Verfahrens. Einer der zu sortierenden Strings wird ausgewählt und der Index des Segments berechnet. Wenn der Eintrag von $\mathsf{OFFSET}[s]$ zwischen a und $a + L$ liegt, ist klar, dass das Suffx an Position $\mathsf{OFFSET}[s]$ nach a beginnt und innerhalb der L ersten Zeichen liegt. Daher hat es mindestens ein übereinstimmendes Zeichen an vorderster Position mit den zu sortierenden Strings. Da bekannt ist, dass alle Strings in A die selben Zeichen an den ersten L Stellen haben, reicht es aus, einen der Strings aus A zu vergleichen. Daher kann der Bucket um $\mathsf{ANCHOR}[s]$ nach den zu sortierenden Strings gescannt werden. Da wir annehmen, dass die Strings im sortierten Bucket nah beieinander liegen, verbessert dies die Laufzeit [38].

Zur Veranschaulichung folgt ein kurzes Beispiel. Unser Text sei wieder „banabanabas“. Sagen wir wieder, wir wollen die Strings „banabas“, „bas“ und

„banabanabas“ sortieren. Diese haben jeweils die Suffix-Indizes 4, 8 und 0. Die Länge des LCP dieser Strings ist 2 („ba“). Daher sei für dieses Beispiel $L = 2$. Zum Sortieren benutzen wir den Bucket b_a, welcher bereits sortiert ist (siehe Abbildung 6.14). Sei für dieses Beispiel $d = 6$. Dann ist also jedes Segment 6 Suffixe groß und es gibt zwei Segmente. Die Segmente sind also „banaba“ und „nabas“. Dann ist $\mathsf{OFFSET}[0] = 1$ („anabanabas“) und $\mathsf{OFFSET}[1] = 1$ („abas“). Die jeweiligen Anker zeigen auf die Position dieser Suffixe in ihren Buckets.

Wir wählen einen der zu sortierenden Strings aus, z.B. den ersten. Dieser hat Suffix-Index 4 und gehört daher zu Segment 0. Da $\mathsf{OFFSET}[0] == 1$ und $1 \notin [4, 4 + L]$ liegt, ist dieser Anker nicht geeignet, um die Strings zu sortieren.

Eigentlich ginge das Sortieren mittels „banabas“ natürlich, da nach Löschen des ersten Zeichens „anabas“ in b_a liegt ($4 < 5 \leq 6$). Um jedoch die Laufzeit gering zu halten, da nicht bekannt ist, an welcher Position „anabas“ steht, werden stattdessen die anderen Kandidaten ausprobiert:

Das Segment vom zweiten String mit Suffix-Index 0 ist ebenfalls 0. Da $\mathsf{OFFSET}[0] == 1$ liegt 1 in diesem Fall im Intervall $[0, 0 + L] = [0, 2]$ und wir können $\mathsf{ANCHOR}[0]$ benutzen, um die Strings zu sortieren. Dazu verwenden wir den Zeiger $\mathsf{ANCHOR}[0]$, welcher auf die Position von „anabanabas“ in b_a zeigt. Wir schneiden die ersten $(1 - 0) = 1$ Zeichen aus unserer String-Menge ab und suchen die Suffixe im Bucket, wie oben beschrieben. Da wir bereits ein Element im Bucket kennen und annehmen, dass die anderen Suffixe nah daran liegen, fällt das Suchen des ersten Elements im Bucket weg. Im Ausgangspaper konnten die Autoren eine signifikante Performance-Erhöhung mittels dieses Ansatzes zeigen.

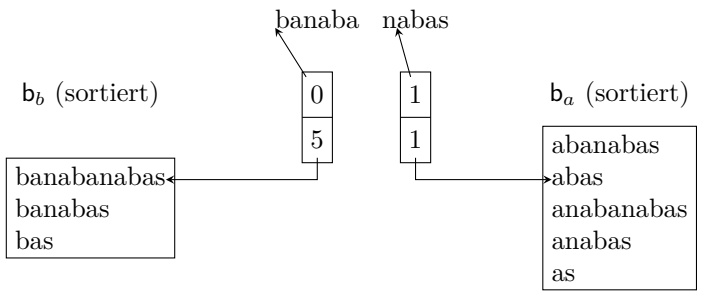


Abbildung 6.15: Szenario nach einem Aufruf von Induced Sort. Der Bucket b_b wurde sortiert und OFFSET und ANCHOR aktualisiert. Es ist zu beachten, dass $\mathsf{ANCHOR}[0]$ nun 5 ist, da dies die Position des Suffixes im Speicher ist und die Buckets nacheinander im Speicher liegen. Daher beginnt die Nummerierung der Suffixe in b_b dort, wo b_a endet.

Im vorherigen Abschnitt haben wir für das Beispiel angenommen, dass alle ANCHOR-Pointer auf denselben Bucket zeigen. Das ist in der Praxis unwahrscheinlich, wodurch die Auswahl möglicher Anker weiter eingeschränkt wird. Falls dennoch mehrere Anker zur Auswahl stehen, wird derjenige ausgewählt,

der $\mathsf{ANCHOR}[imodd] - i$ minimiert (also einen möglichst kurzen gemeinsamen Präfix mit dem Anker hat).

Manzini und Ferragina haben in ihren Benchmarks verschiedene Werte der Segmentgröße d zwischen 500 und 5000 getestet. Auf einigen Problemen hat eine Vergrößerung von d keinen merklichen Unterschied in der Laufzeit gebracht. Allerdings wurde die Laufzeit auf einigen Dateien deutlich schlechter. Die Autoren bewerben ihren Algorithmus allerdings mit $5.03n$ Byte Speicherverbrauch, was einem d von 500 entspricht.

Copy-Technik

Nachdem ein Bucket b_α fertig sortiert wurde, können alle Buckets $\mathsf{b}_{\sigma\alpha}$ in einem Schritt sortiert werden. Dies basiert auf dem copy-SACA von Seward [50]. Kern dieses Schrittes ist es, dass die relative Ordnung aller Suffixe, welche mit $\sigma\alpha$ beginnen, durch die Ordnung aller α-Suffixe sortiert werden können.

Um dies effizient durchzuführen, wird zuerst ein Array mit Pointern auf den Anfang jedes $\sigma\alpha$-Buckets erstellt. Es genügt nun jedes $i \in \mathsf{b}_\alpha$ zu durchlaufen und das Zeichen vor diesem Suffix zu lesen, $\mathsf{T}[i-1]$. Sei dieses Zeichen jeweils σ_i. Dieses kann man an die Stelle schreiben, auf die aktuell der Anfangspointer für den $\sigma_i\alpha$-Bucket zeigt. Der Pointer wird dann erhöht und das nächste Suffix gelesen. Nach diesem Schritt sind alle $\mathsf{b}_{\sigma\alpha}$ sortiert und können als solche markiert werden. So können sie beim Sortieren von b_σ übersprungen werden.

6.5.3 Beispiel

2	12	10	1	13	8	4	7	3	9	5	11	6	0
b_a								b_b		b_c			

Abbildung 6.16: SA nach Schritt 1: Bucket Sort

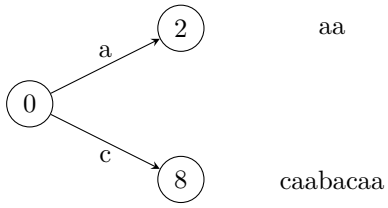


Abbildung 6.17: Blind Trie in der ersten Deep Sort Phase

Zur Veranschaulichung des Algorithmus wird im folgenden Abschnitt der Algorithmus beispielhaft[4] am Eingabetext „caabaccaabacaa“ ausgeführt.

[4]Insbesondere wird der Algorithmus mit Beispiel-Parameterwerten ausgeführt, die in einer echten Anwendung sehr unangebracht wären. Diese Parameter sind $L = 3, d = 7$. Außerdem wird der copy-Schritt weggelassen, um mehr interessante Fälle zeigen zu können.

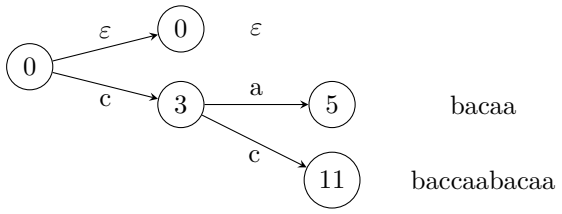


Abbildung 6.18: Blind Trie in der zweiten Deep Sort Phase. ε steht dabei für *keinen* Buchstaben.

Schritt 1 Sortiere alle Suffixe mittels Bucket Sort nach dem ersten Zeichen[5] in einen der Buckets. Der Status des Algorithmus nach diesem Schritt ist in Abbildung 6.16 abgebildet.

Schritt 2 Sortiere alle Buckets mit Shallow Sort in nach Größe aufsteigender Reihenfolge. Der kleinste Bucket ist b_b. Dieser enthält „bacaa“ und „bacaa“. Wir vergleichen die Strings in MKQS bis zum dritten Zeichen. Da wir bis dahin noch keinen Unterschied festgestellt haben, schalten wir auf Deep Sort um. Weil der Bucket klein genug ist (und damit das Beispiel interessant ist), wird Blind Sort benutzt, um den Bucket zu sortieren. Der Blind Trie ist in Abbildung 6.17 dargestellt. Beim Vergleich sind die ersten drei Zeichen ignoriert worden, da sie als gleich bekannt sind.

Schritt 3 Der Bucket b_b wird als sortiert markiert und der Nächstgrößere sortiert. b_c wird sortiert; dieser enthält „ccaabacaa“, „caa“, „caabacaa“ und „caabaaccaabacaa“. Beim Versuch, diese mit MKQS zu sortieren, kann nur die relative Ordnung von „ccaabacaa“ und den anderen drei Suffixen hergeleitet werden: „ccaabacaa“ ist größer als die anderen drei Suffixe. Die anderen drei Suffixe haben ein zu langes LCP, um komplett in Shallow Sort sortiert zu werden. Daher wird für diese wieder ein Blind Trie benutzt, welcher in Abbildung 6.18 dargestellt ist.

[5]Sowohl die Referenzimplementierung als auch unsere Implementierung benutzen Buckets, welche nach den ersten beiden Zeichen sortiert wurden. Aus Gründen der Veranschaulichung wird hier allerdings nach nur einem Buchstaben sortiert.

Schritt 4 Der letzte zu sortierende Bucket, b_c wird auch wieder mit Shallow Sort sortiert. Allerdings kann durch den Vergleich der ersten drei Zeichen nicht die Ordnung von „abaccaabacaa“, „abacaa“, „aabaccaabacaa“ und „aabacaa“ hergeleitet werden. Für jeweils Paare aus zwei Suffixen („aabacaa“ und „aabaccaabacaa“, und „abaccaabacaa“ und „abacaa“) können wir allerdings Induced Sort benutzen: Im bekannten LCP der ersten beiden Strings „aab“ befindet sich ein „b“. Daher können wir zur Sortierung den bereits sortierten b_b-Bucket verwenden. Darin suchen wir „bacaa“ und „baccaabacaa“ und können daraus die relative Sortierung herleiten. Das gleiche Verfahren wird für das zweite Paar verwendet.

Schlussendlich ist damit der gesamte Text sortiert, da es keine unsortierten Buckets mehr gibt.

6.5.4 Variante und Parallelität

```
def deep_shallow_sort_parallel(T: Text):
  Create a bucket for every character pair α, β ∈ Σ.
  Call them b_αβ.

  # Sort T into the buckets by comparing only the first two
  # characters of every string. Represent the suffixes by their
  # starting indices in T.
  bucket_sort(T)

  parallel for all buckets b_αβ do:
    shallow_sort(b_αβ, 0)
```

Algorithmus 6.9: parallele Variante des Algorithmus

Die ursprüngliche Variante heißt in SACABench „Deep-Shallow_bb“ (bb für *Big Buckets*), da sie die Superbuckets b_σ durchläuft und sortiert. Durch das Weglassen des copy-Verfahrens haben wir eine weitere Variante implementiert. Diese durchläuft nicht alle Buckets b_σ in aufsteigender Größe, sondern alle $\mathsf{b}_{\alpha\beta}$.

Durch zusätzliches Weglassen des Induced Sorts ist es möglich, den Algorithmus sehr simpel zu parallelisieren. In Listing 6.9 findet sich der Pseudocode dieser Variante. Dabei werden die $\alpha\beta$-Buckets alle zeitgleich sortiert und der copy-Schritt (siehe Abschnitt 6.5.2) weggelassen. Eine Synchronisation der Threads ist dann nicht nötig, wodurch sich eine hohe parallele Effizienz ergibt. Die Referenzimplementierung konnte von uns aufgrund ihrer unübersichtlichen C-Implementierung nicht parallelisiert werden, da nicht klar erkennbar war, an welchen Stellen auf die gemeinsamen Speicherbereiche zugegriffen wird. In unserer Auswertung zeigt sich, dass dieser Ansatz gut skaliert und keinen zusätzlichen Speicher verbraucht.

6.5.5 Vergleich

Manzini und Ferragina konnten zeigen, dass Deep-Shallow-Sort ein zuverlässiger Ansatz ist, der mit anderen Verfahren mithalten kann [38]. Im Laufe des ersten Semesters der Projektgruppe war es Deep-Shallow einfach möglich, die meisten Algorithmen in ihrem Speicherverbrauch zu unterbieten (abgesehen von den Algorithmen, welche keinen zusätzlichen Speicher verbrauchen). Allerdings zeigte sich unsere Implementierung schwerfällig, die anderen Algorithmen in ihrer Laufzeit zu schlagen.

6.6 BPR

Der im Jahr 2005 vorgestellte Algorithmus *Bucket-Pointer Refinement* [49] erreicht trotz nicht optimaler asymptotischer Laufzeitschranke sehr schnelle Konstruktionszeiten von Suffix-Arrays für eine Vielzahl verschiedenartiger Strings. Aufgrund der daraus resultierenden Relevanz im Bereich der Textindexierung wird dieser Algorithmus hier im Rahmen der Projektgruppe SACABench anhand einfacher Beispiele aufgearbeitet, wobei neben der Funktionsweise auch Laufzeit und Speicherbedarf im Vordergrund stehen.

Die Ziele des *Bucket-Pointer Refinement*-Algorithmus von Schürmann und Stoye sind eine weniger komplexe Verarbeitung im Vergleich zu anderen Algorithmen sowie gute praktische Laufzeiten. Dieser Algorithmus wird hier daher zunächst anhand eines geeigneten Beispiels erklärt und anschließend im Bezug auf Komplexität hinsichtlich Speicherbedarf und Laufzeit analysiert.

6.6.1 Vorüberlegungen

Damit einige der Schritte im späteren Verlauf des Algorithmus im Bezug auf Indizes außerhalb des zu verarbeitenden Strings wohldefiniert sind, betrachten wir häufig eine erweiterte Version eines Strings gemäß Definition 14. Abbildung 6.19 zeigt beispielhaft eine Erweiterung eines Strings.

c	o	v	f	e	f	e	f	e
$\mathsf{T}[0]$	$\mathsf{T}[1]$	$\mathsf{T}[2]$	$\mathsf{T}[3]$	$\mathsf{T}[4]$	$\mathsf{T}[5]$	$\mathsf{T}[6]$	$\mathsf{T}[7]$	$\mathsf{T}[8]$

c	o	v	f	e	f	e	f	e	$\cdots$
$\mathsf{T}^+[0]$	$\mathsf{T}^+[1]$	$\mathsf{T}^+[2]$	$\mathsf{T}^+[3]$	$\mathsf{T}^+[4]$	$\mathsf{T}^+[5]$	$\mathsf{T}^+[6]$	$\mathsf{T}^+[7]$	$\mathsf{T}^+[8]$	

$\cdots$	$	$	$	$	$	$	$	$	$
	$\mathsf{T}^+[9]$	$\mathsf{T}^+[10]$	$\mathsf{T}^+[11]$	$\mathsf{T}^+[12]$	$\mathsf{T}^+[13]$	$\mathsf{T}^+[14]$	$\mathsf{T}^+[15]$	$\mathsf{T}^+[16]$	$\mathsf{T}^+[17]$

Abbildung 6.19: Erweiterung von T zu T^+ durch Anhängen von $\$^n$

Definition 14 (T^+)**.** *Sei Σ ein total geordnetes endliches Alphabet und $\mathsf{T} = \mathsf{T}[0]\mathsf{T}[1]\dots\mathsf{T}[n-1] \in \Sigma$ ein String über Σ. Sei außerdem $\$ \notin \Sigma$ ein nicht in Σ enthaltenes Symbol mit $\forall c \in \Sigma : \$ < c$. Den um $\$^n$ erweiterten String $\mathsf{T}\$^n$ nennen wir T^+.*

Ist das zugrunde liegende Alphabet außerdem total geordnet, so können auch die Suffixe eines Strings anhand dieser Ordnung sortiert werden. Der Vergleich zweier Symbole des Alphabets bezüglich der Ordnung wird dadurch vereinfacht, dass jedem Symbol mit der Funktion *eff* (Definition 9) eine natürliche Zahl als Rang zugewiesen wird.

Während für einzelne Symbole die Verwendung des effektiven Alphabets gegenüber dem direkten Vergleich keinen wesentlichen Vorteil erzielt, ergibt sich bei dem lexikographischen Vergleich zweier Strings bereits eine andere Situation:

Auf Basis der Ränge der einzelnen Symbole kann auch ganzen Strings eine ganzzahlige Codierung zugewiesen werden, welche den Vergleich beschleunigt.

Definition 15 (Stringkodierung)**.** *Sei* $code_d : (\Sigma \cup \{\$\})^d \to \{0, \ldots, (|\Sigma|+1)^d - 1\}$ *eine bijektive Funktion*[1]*, sodass für zwei Strings* $u, v \in (\Sigma \cup \{\$\})^d$ *gilt* $u < v \Leftrightarrow code_d(u) < code_d(v)$*.*

Eine Stringkodierung wie in Definition 15 kann einfach unter Verwendung einer gegebenen *eff*-Funktion (Definition 9) realisiert werden, indem die Symbole eines Strings abhängig von ihrem Index mit absteigender Wertigkeit für höhere Indizes summiert werden. Auf diese Weise ergibt sich für einen String $\mathsf{T}[i, i+d) = \mathsf{T}[i] \ldots \mathsf{T}[i+d-1]$ die Kodierung

$$code_d(\mathsf{T}[i, i+d)) = \sum_{j=0}^{d-1} (|\Sigma|+1)^{d-1-j} \cdot eff(\mathsf{T}[i+j]).$$

Auch Kodierungen für Strings größerer Länge als d können auf diese Weise berechnet werden, indem alle Symbole an Indizes größer oder gleich d ignoriert werden. Aufgrund der hier gewählten Berechnungsvorschrift für $code_d$ lassen sich außerdem auf sehr einfache Weise Kodierungen für aufeinanderfolgende Suffixe berechnen.

$$code_d(\mathsf{S}_{i+1}) = (|\Sigma|+1) \cdot \left(code_d(\mathsf{S}_i) \bmod (|\Sigma|+1)^{d-1}\right) + eff(\mathsf{T}^+[i+d]) \quad (6.1)$$

Man beachte hier, dass für zwei aufeinanderfolgende Suffixe S_i und S_{i+1} lediglich das erste Symbol aus S_i aus der Kodierung entfernt werden muss (mittels mod $(|\Sigma|+1)^{d-1}$), woraufhin die Wertigkeit aller verbleibenden Symbole jeweils um eine Stelle ansteigt. Zuletzt muss nur noch der Rang des letzten Symbols aus S_{i+1} hinzu addiert werden.

Wie viele andere Algorithmen zur Konstruktion von Suffix-Arrays, arbeitet auch *Bucket-Pointer Refinement* mit einer Unterteilung des Suffix-Arrays in sogenannte Buckets (Definition 7 auf Seite 29). Die Unterteilung des Arrays in Buckets ist erforderlich, um das Suffix-Array schrittweise sortieren zu können, indem bestehende Buckets durch Aufteilung weiter verfeinert werden.

6.6.2 Algorithmus

Nachdem in Kapitel 6.6.1 bereits die formalen Grundlagen für die Beschreibung des *Bucket-Pointer Refinement*-Algorithmus geschaffen wurden, können wir nun präziser auf die Funktionsweise des Algorithmus eingehen. Die Basis für das Verfahren bildet die Nutzung von Abhängigkeiten zwischen mehreren Suffixen: Teilen sich zwei Suffixe S_i und S_j einen gemeinsamen Präfix der Länge *offset*, so ist zunächst festzustellen, dass $\mathsf{S}_{i+offset}$ bzw. $\mathsf{S}_{j+offset}$ nicht nur Suffixe von T, sondern auch Suffixe von S_i bzw. S_j sind. Die Reihenfolge von S_i und S_j im Suffix-Array kann also einfach durch die Reihenfolge der eventuell bereits zuvor sortierten Suffixe $\mathsf{S}_{i+offset}$ und $\mathsf{S}_{j+offset}$ bestimmt werden. Um allerdings auf eine

derartige Vorsortierung zurückgreifen zu können, muss zu Beginn zumindest eine grobe „initiale Sortierung“ der Suffixe existieren.

Um dies zu erreichen, verwendet der hier beschriebene Algorithmus zwei Phasen, von denen die erste dazu dient, die Suffixe initial nach gemeinsamen Präfixen der Länge d zu gruppieren. Auf Basis dieser Gruppierung werden die Suffixe Buckets zugewiesen, sodass sich zwei Suffixe genau dann im gleichen Bucket befinden, wenn sie einen gemeinsamen Präfix der Länge d besitzen.

In der zweiten Phase werden dann die Buckets anhand der oben genannten Vorschrift schrittweise rekursiv verfeinert. Auf diese Weise ergibt sich das fertig sortierte Suffix-Array, sobald jeder Bucket die minimale Länge von 1 erreicht hat.

Im Folgenden werden beide Phasen im Detail erklärt. Wir verwenden dafür als Beispiel das Fantasiewort `covfefefe`, dessen Suffixe wir mittels *Bucket-Pointer Refinement* sortieren.

Erste Phase

Die erste Phase führt eine Gruppierung der Suffixe nach gemeinsamen Präfixen der Länge d durch. Die Wahl eines geeigneten Parameters d wird nicht vom Algorithmus übernommen, es wird allerdings empfohlen, $d < \log n$ zu wählen [49].

Um nun alle Suffixe von T in Level-d-Buckets zu gruppieren, wird zunächst eine Tabelle *bkt* der Länge $(|\Sigma| + 1)^d$ mit Einträgen für alle möglichen Präfixe der Länge d erstellt, wobei jeder Präfix eindeutig durch den zugehörigen Funktionswert von $code_d$ identifiziert werden kann. In einem sequentiellen Durchlauf des Textes T^+ können dann mithilfe der Eigenschaft aus Gleichung 6.1 die Kodierungen der vorkommenden Präfixe effizient berechnet werden. In diesem Durchlauf kann für jeden Präfix p die zugehörige Anzahl der Vorkommen in T^+ (und damit die spätere Größe des Buckets b_p) bestimmt werden.

Da die Sortierung der Buckets durch den Schlüssel $code_d$ gegeben ist, können im Anschluss anhand der nun bekannten Bucketgrößen auch die Positionen aller Buckets im Suffix-Array festgelegt werden. Der entsprechende Stand der

prefix	⋯	c$	cc	ce	cf	co	cv	e$	ec	ee	ef	⋯	fe	⋯	ov	⋯	vf	vo	vv
$code_d$ (base $\|\Sigma\|+1$)	⋯	10	11	12	13	14	15	20	21	22	23	⋯	32	⋯	41	⋯	53	54	55
# in T	⋯	0	0	0	0	1	0	1	0	0	2	⋯	3	⋯	1	⋯	1	0	0
sum	⋯	0	0	0	0	1	1	2	2	2	4	⋯	7	⋯	8	⋯	9	9	9

Abbildung 6.20: Tabelle *bkt* zur Bestimmung der Bucket-Größen und -Positionen. Die Anzahl der Vorkommen eines Präfix im Text ist in Zeile 2 angegeben. Zeile 3 beinhaltet die kumulierte Anzahl und bestimmt damit die linke Grenze des nachfolgenden Buckets.

Verarbeitung für den Text $\mathsf{T} =$ `covfefefe` ist in Abbildung 6.20 zu sehen. Die kumulierte Größe aller Buckets bis einschließlich Bucket j legt gleichzeitig die linke Grenze des Buckets $j + 1$ fest. In einem weiteren Durchlauf können jetzt

alle Suffixe $\mathsf{S}_0, \ldots, \mathsf{S}_{n-1}$ im vorläufigen Suffix-Array SA einsortiert werden. Abbildung 6.21 zeigt die initiale Einteilung der Buckets für $d = 2$. Zur besseren Visualisierung sind hier in dem Array die Suffixe anstatt der bloßen Indizes angegeben.

S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
c	o	v	f	e	f	e	f	e
o	v	f	e	f	e	f	e	
v	f	e	f	e	f	e		
f	e	f	e	f	e			
e	f	e	f	e				
f	e	f	e					
e	f	e						
f	e							
e								

	co	e$		ef			fe		ov	vf
	S_0	S_8	S_4	S_6	S_3	S_5	S_7	S_1	S_2
	0	**1**	**2**	**3**	4	5	**6**	**7**	**8**
	↑	↑	↑	↑	↑	↑	↑	↑	↑
	c	e	e	e	f	f	f	o	v
	o		f	f	e	e	e	v	f
	v		e	e	f	f		f	e
	f		f		e	e		e	f
	e		e		f			f	e
	f				e			e	f
	e							f	e
	f							e	
	e								

Abbildung 6.21: Suffixe unsortiert (links) und nach der ersten Phase des Algorithmus (rechts)

Wie bereits zuvor erwähnt, wollen wir in der zweiten Phase des Algorithmus die Buckets verfeinern, indem eine weitere Gruppierung jedes bestehenden Buckets aufgebaut wird.

Wir wollen dazu auf die bereits bekannte Einteilung der Suffixe in Level-d-Buckets zugreifen und anhand dessen eine weiterführende Sortierung vollziehen. Eine Anfrage, in welchem Bucket sich ein Suffix befindet, ist jedoch mit der bisher eingeführten Datenstruktur nicht effizient zu beantworten.

Es wird daher mit dem Bucket Pointer BPTR ein zweites Array eingeführt, welches sinngemäß ein Reverse Mapping des Suffix-Arrays SA beinhaltet (s. Abbildung 6.22). In diesem Array symbolisiert jeder Index ein Suffix, wobei der Eintrag $\mathsf{BPTR}[i]$ den zugehörigen Bucket angibt, welcher jeweils über seine rechte Grenze identifiziert wird. Die Bestimmung der Grenzen erfolgt durch Ablesen der kumulierten Summe aus der Tabelle *bkt*.

	co	e$		ef			fe		ov	vf
	S_0	S_8	S_4	S_6	S_3	S_5	S_7	S_1	S_2
	0	1	2	3	4	5	6	7	8
	0	7	8	6	3	6	3	6	1
	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
	↑	↑	↑	↑	↑	↑	↑	↑	↑
	c	o	v	f	e	f	e	f	e
	o	v	f	e	f	e	f	e	
	v	f	e	f	e	f	e		
	f	e	f	e	f	e			
	e	f	e	f	e				
	f	e	f	e					
	e	f	e						
	f	e							
	e								

Abbildung 6.22: Vorsortierte Suffixe und zugehörige Bucket-Pointer

Zweite Phase

Zu Beginn dieser Phase des Algorithmus existiert bereits ein (noch nicht fertig sortiertes) Suffix-Array SA, welches nun rekursiv sortiert werden soll. Die aktuelle Einteilung der Suffixe in Buckets ist über den Bucket Pointer BPTR gegeben, aus dem abgelesen werden kann, ob sich zwei Suffixe im gleichen oder in verschiedenen Buckets befinden. Das Array *bkt*, welches in der ersten Phase des Algorithmus erstellt wurde, um die initiale Zuteilung festzulegen, wird im weiteren Verlauf nicht mehr verwendet.

Die Sortierung erfolgt rekursiv für jeden Bucket im Suffix-Array. Da die einzelnen Buckets untereinander zu jedem Zeitpunkt bereits sortiert sind, muss der Sortiervorgang in jedem Rekursionsschritt nur innerhalb des jeweiligen Buckets stattfinden. Da es für effizientes Sortieren aber erforderlich ist, zwei Suffixe mit konstantem Aufwand vergleichen zu können, werden wir zunächst sehen, anhand welcher Kriterien wir Suffixe effizient vergleichen können. Am bereits zuvor ein-

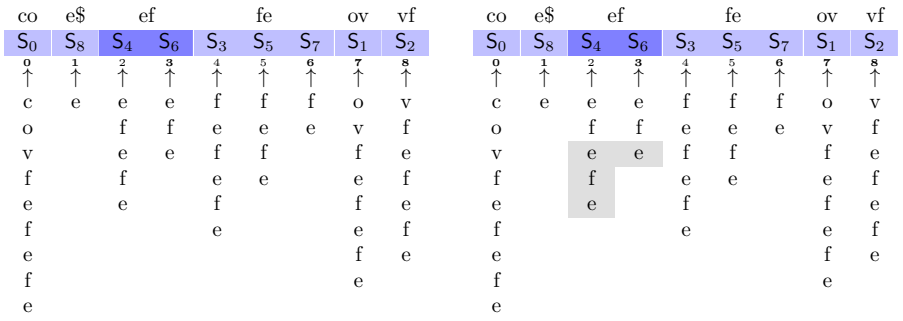


Abbildung 6.23: Vergleich zweier Suffixe innerhalb eines Buckets

geführten Beispiel beginnen wir, den Bucket b_{ef} zu sortieren (Abbildung 6.23). Der Bucket beinhaltet die Suffixe S_4 und S_6, welche nach dem Sortierschritt in Phase 1 einen gemeinsamen Präfix der Länge $d = 2$ haben. Da die Sortierung der Suffixe lexikographisch erfolgen soll, können gemeinsame Präfixe ignoriert werden, ohne die Korrektheit der Sortierung zu beeinflussen. Für Suffixe $\mathsf{S}_i, \mathsf{S}_j$ mit $i, j \in \mathsf{b}$ gibt der Level des Buckets b eine untere Schranke für *offset* an. Im Falle des Beispiels genügt es also, die Suffixe $\mathsf{S}_{4+2} = \mathsf{S}_6$ und $\mathsf{S}_{6+2} = \mathsf{S}_8$ zu vergleichen (Abbildung 6.24, oben links). Da selbstverständlich Suffixe der Suffixe von T auch selbst Suffixe von T sind, befinden sich auch diese in SA. Falls sich $\mathsf{S}_{i+offset}$ und $\mathsf{S}_{j+offset}$ bereits in verschiedenen Buckets befinden, ist das lexikographische Verhältnis dieser beiden Suffixe bereits bekannt. Um dies herauszufinden, werden die Bucket Pointer der entsprechenden Suffixe verglichen. Diese befinden sich im Array BPTR, welches bereits in Phase 1 initialisiert wurde und beinhalten einen Verweis auf die rechte Grenze des Buckets, in dem sich der jeweilige Suffix befindet.

Abbildung 6.24 (unten links) zeigt die Bucket Pointer der Suffixe S_6 (Bucket b_{ef} mit rechter Grenze an Index 3 in SA) und S_8 (Bucket $\mathsf{b}_{\mathsf{e\$}}$ mit rechter Grenze an Index 1 in SA). Aus einem einfachen Vergleich dieser Bucket Pointer geht

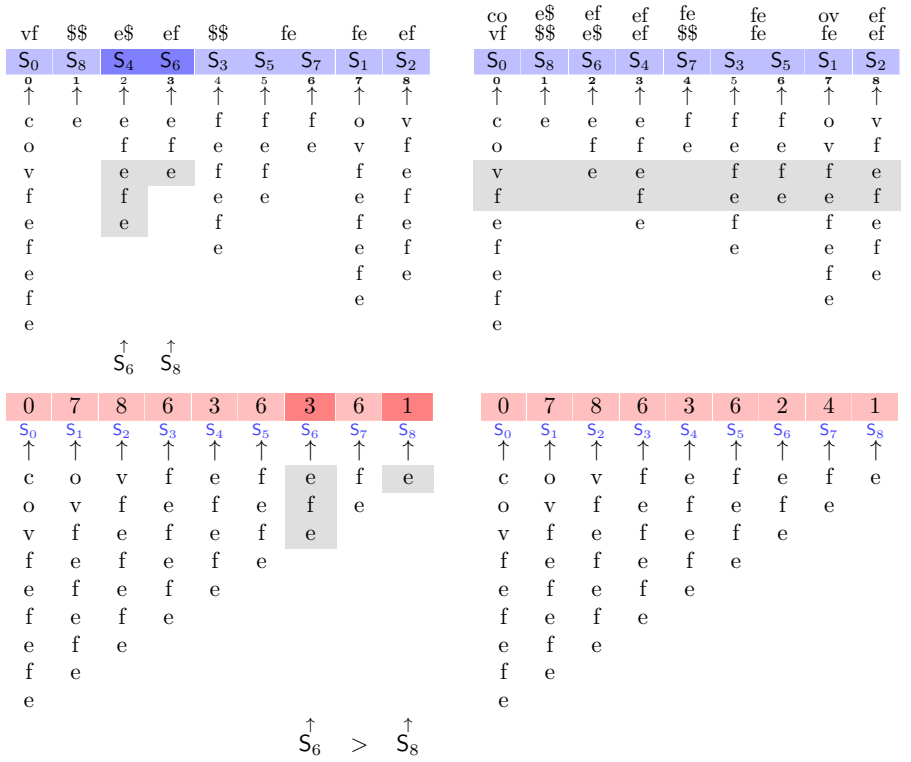


Abbildung 6.24: Buckets (oben) sowie Bucket-Pointer (unten) vor und nach der ersten Verfeinerung. Es ist gut zu erkennen, dass bereits bei einer Rekursionstiefe von 1 nur noch ein nicht eindeutig sortierter Bucket existiert.

hervor, dass S_8 lexikographisch kleiner ist als S_6. Mithilfe von Lemma 1 kann daraus geschlossen werden, dass auch S_6 lexikographisch kleiner ist als S_4. Ein solcher Vergleich kann durch zwei Zugriffe auf das Array BPTR in konstanter Zeit durchgeführt werden. Zur Veranschaulichung zeigt Abbildung 6.24 (rechts) den Zustand von SA und BPTR, wenn in SA nur noch Level-4-Buckets enthalten sind. Nachdem ein Verfeinerungsschritt in einem Bucket $\mathsf{b}_\mathrm{p} = [l, r]$ in kleinere Buckets durchgeführt wurde, müssen außerdem die zugehörigen Bucket-Pointer im Array BPTR angepasst werden. Dazu wird der Bucket b_p von rechts nach links durchlaufen, wobei für jeden darin enthaltenen Suffix $\mathsf{SA}[i], l \leq i \leq r$, der Sortiertschlüssel $\mathsf{BPTR}[\mathsf{SA}[i] + \mathit{offset}]$ abgefragt. Es wird dann zwischen zwei Fällen unterschieden:

Fall 1: $\mathsf{BPTR}[\mathsf{SA}[i-1] + \mathit{offset}] = \mathsf{BPTR}[\mathsf{SA}[i] + \mathit{offset}]$
Die Sortierschlüssel der Suffixe $\mathsf{SA}[i-1]$ und $\mathsf{SA}[i]$ sind identisch. Daher konnte in diesem Schritt keine echte Sortierung dieser beiden Suffixe vorgenommen werden. Deshalb befinden sie sich nach wie vor in einem gemeinsamen Bucket und $\mathsf{BPTR}[\mathsf{SA}[i-1]]$ wird auf $\mathsf{BPTR}[\mathsf{SA}[i]]$ gesetzt.

Fall 2: $\mathsf{BPTR}[\mathsf{SA}[i-1]+\mathit{offset}] \neq \mathsf{BPTR}[\mathsf{SA}[i]+\mathit{offset}]$
Die Sortierschlüssel der Suffixe $\mathsf{SA}[i-1]$ und $\mathsf{SA}[i]$ unterscheiden sich. Die Suffixe konnten also sortiert werden und befinden sich nach der Verfeinerung in unterschiedlichen Buckets. $\mathsf{SA}[i-1]$ ist das erste Element von rechts in dem neu beginnenden Bucket. Folglich wird $\mathsf{BPTR}[\mathsf{SA}[i-1]]$ auf $i-1$ gesetzt.

Nachdem die Bucket-Pointer aktualisiert wurden, ist der Sortiervorgang für b_p abgeschlossen und es kann rekursiv fortgefahren werden. Das fertig sortierte Suffix-Array (links), welches nach Abschluss der Sortierung nur noch Buckets der Größe 1 enthält, sowie die dazugehörigen Bucket-Pointer (rechts) sind in Abbildung 6.25 zu sehen.

S_0	S_8	S_6	S_4	S_7	S_5	S_3	S_1	S_2
0	1	2	3	4	5	6	7	8
c	e	e	e	f	f	f	o	v
o		f	f	e	e	e	v	f
v		e	e		f	f	f	e
f			f		e	e	e	f
e			e			f	f	e
f						e	e	f
e							f	e
f							e	
e								

0	7	8	6	3	5	2	4	1
S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
c	o	v	f	e	f	e	f	e
o	v	f	e	f	e	f	e	
v	f	e	f	e	f	e		
f	e	f	e	f	e			
e	f	e	f	e				
f	e	f	e					
e	f	e						
f	e							
e								

Abbildung 6.25: Sortiertes Suffix-Array (links) und zugehörige Bucket-Pointer (rechts)

Bei der Wahl des Sortierverfahrens unterscheidet *Bucket-Pointer Refinement* abhängig von der Größe eines Buckets zwischen *Insertionsort* und *Quicksort*. Für Buckets mit maximal 15 Elementen wird *Insertionsort* verwendet, während alle größeren Buckets mit einer Variante von *Quicksort* sortiert werden. Ergänzend zur Partitionierung des Buckets in zwei Hälften werden alle an das Pivotelement angrenzenden Elemente mit gleichem Sortierschlüssel von den rekursiv zu sortierenden Partitionen ausgenommen, solange bis in beide Richtungen das erste Element mit unterschiedlichem Schlüssel gefunden wird. Diese Heuristik sorgt dafür, dass je nach Struktur des Eingabestrings die zu sortierenden Partitionen deutlich verkleinert werden können, falls in einem Bucket viele Suffixe den gleichen Schlüssel haben.

Falls es aufgrund der Struktur des Eingabestrings vorkommt, dass innerhalb eines Buckets alle Suffixe einen gemeinsamen Präfix einer deutlich größeren Länge als *offset* haben, dann würde *Quicksort* (bzw. bei kleinen Buckets *Insertionsort*) für mehrere Schritte in der Rekursion versuchen, ausschließlich Elemente mit gleichem Schlüssel zu sortierten. Dies beeinflusst zwar nicht die Korrektheit des Verfahrens, allerdings entsteht durch jeden überflüssigen Sortiervorgang ein unerwünschter und vermeidbarer Aufwand. Um dem entgegen zu wirken, wird eine Heuristik verwendet, welche nach einem „erfolglosen“ Sortiervorgang die Länge des längsten gemeinsamen Präfixes aller Suffixe in diesem

Bucket bestimmt. Der darauf folgende Rekursionsschritt wird dann mit dieser Länge anstelle von *offset* aufgerufen.

Beispiel

Am Beispiel `caabaccaabacaa` wird noch einmal die Funktionsweise des gesamten Algorithmus im Detail demonstriert. Das Beispiel dient der Übersichtlichkeit und Vergleichbarkeit zu anderen Algorithmen. Der erste Schritt besteht ledig-

Schritt 1 — Phase 1

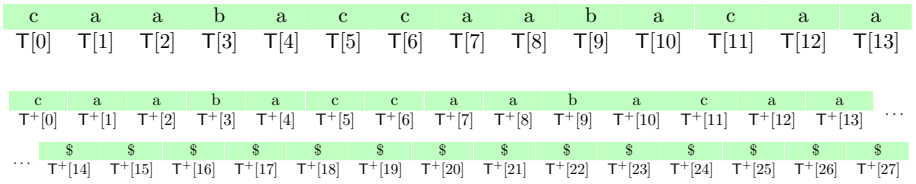


Abbildung 6.26: *Bucket-Pointer Refinement*: Phase 1, Schritt 1. Oben: Eingabetext T. Unten: Erweiterter Eingabetext T^+.

lich daraus, die Eingabe aufzubereiten, indem n symbolische Sentinels an das Eingabewort angehängt werden. Dieses Vorgehen wird im Paper beschrieben, im Algorithmus allerdings aufgrund des linearen Speicherbedarfs nicht direkt umgesetzt.

Schritt 2

	prefix	$	a	b	c	a$	aa	ab	ac	b$	ba	bb	bc	c$	ca	cb	cc
code_d (base $\lvert\Sigma\rvert+1$)		00	01	02	03	10	11	12	13	20	21	22	23	30	31	32	33
	# in T	0	0	0	0	1	3	2	2	0	2	0	0	0	3	0	1
	sum	0	0	0	0	1	4	6	8	8	10	10	10	10	13	13	14

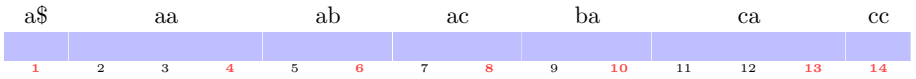


Abbildung 6.27: *Bucket-Pointer Refinement*: Phase 1, Schritt 2. Oben: Größen und Positionen der Buckets. Unten: Leere Buckets im Suffix-Array.

Im zweiten Schritt wird der Bucketsort Algorithmus (Abschnitt 4.1.4) durchgeführt, um eine Vorsortierung für das Suffix-Array zu erlangen. Die Tiefe, die für Bucketsort als Parameter verwendet wird, leitet sich in festgelegten Stufen aus der Größe des Alphabets ab und beträgt auch für große Alphabete mindestens 3. In diesem Beispiel wird aus Gründen der Übersichtlichkeit nur eine Tiefe von 2 verwendet.

Schritt 3

a$		aa		ab		ac		ba		ca		cc	
S_{14}	S_2	S_8	S_{13}	S_3	S_9	S_5	S_{11}	S_4	S_{10}	S_1	S_7	S_{12}	S_6
1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	a	a	a	a	a	a	a	b	b	c	c	c	c
	a	a	a	b	b	c	c	a	a	a	a	a	c
	b	b		a	a	c	a	c	c	a	a	a	a
	a	a		c	c	a	a	c	a	b	b		a
	c	c		c	a	a		a	a	a	a		b
	c	a		a	a	b		a		c	c		a
	a	a		a		a		b		c	a		c
	a			b		c		a		a	a		a
	b			a		a		c		a			a
	a			c		a		a		b			
	c			a				a		a			
	a			a						c			
	a									a			
										a			
13	4	6	10	8	14	13	4	6	10	8	13	4	1
S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}
c	a	a	b	a	c	c	a	a	b	a	c	a	a
a	a	b	a	c	c	a	a	b	a	c	a	a	
a	b	a	c	c	a	a	b	a	c	a	a		
b	a	c	c	a	a	b	a	c	a	a			
a	c	c	a	a	b	a	c	a	a				
c	c	a	a	b	a	c	a	a					
c	a	a	b	a	c	a	a						
a	a	b	a	c	a	a							
a	b	a	c	a	a								
b	a	c	a	a									
a	c	a	a										
c	a	a											
a	a												
a													

Abbildung 6.28: *Bucket-Pointer Refinement*: Phase 1, Schritt 3. Links: Befüllte Buckets im Suffix-Array. Rechts Initiales BPTR-Array.

Zu Beginn des dritten Schrittes sind die rechten Grenzen der Buckets bereits bekannt (siehe Abbildung 6.27, rot markiert und Abbildung 6.28, oben) und die Suffixe in die entsprechenden Buckets einsortiert. Daraus wird im Anschluss das Bucket-Pointer Array (Abbildung 6.28, unten) berechnet, in dem für jeden Suffix gespeichert ist, in welchem Bucket sich dieser in der aktuellen Sortierung befindet. Die Buckets werden dabei über ihre rechte inklusive Grenze identifiziert.

Sobald die Bucket-Pointer bestimmt sind, ist ein Zustand erreicht, von dem aus im Anschluss in der zweiten Phase alle Buckets Schritt für Schritt verfeinert werden können. Diese Verfeinerung erfolgt iterativ über die Buckets und

rekursiv innerhalb der Buckets. Die Reihenfolge, in der die nach Phase 1 entstandenen Buckets sortiert werden, ist im Grundalgorithmus für die Korrektheit irrelevant. Unter Verwendung der Copy-Technik (Abschnitt 6.6.3) lässt sich der Rechenaufwand aber durch eine geschickte Wahl der Reihenfolge verringern. Copy wird in diesem überschaubaren Beispiel nicht verwendet, da das Prinzip von Copy auf derart kurzen Texten nicht hinreichend veranschaulicht werden kann.

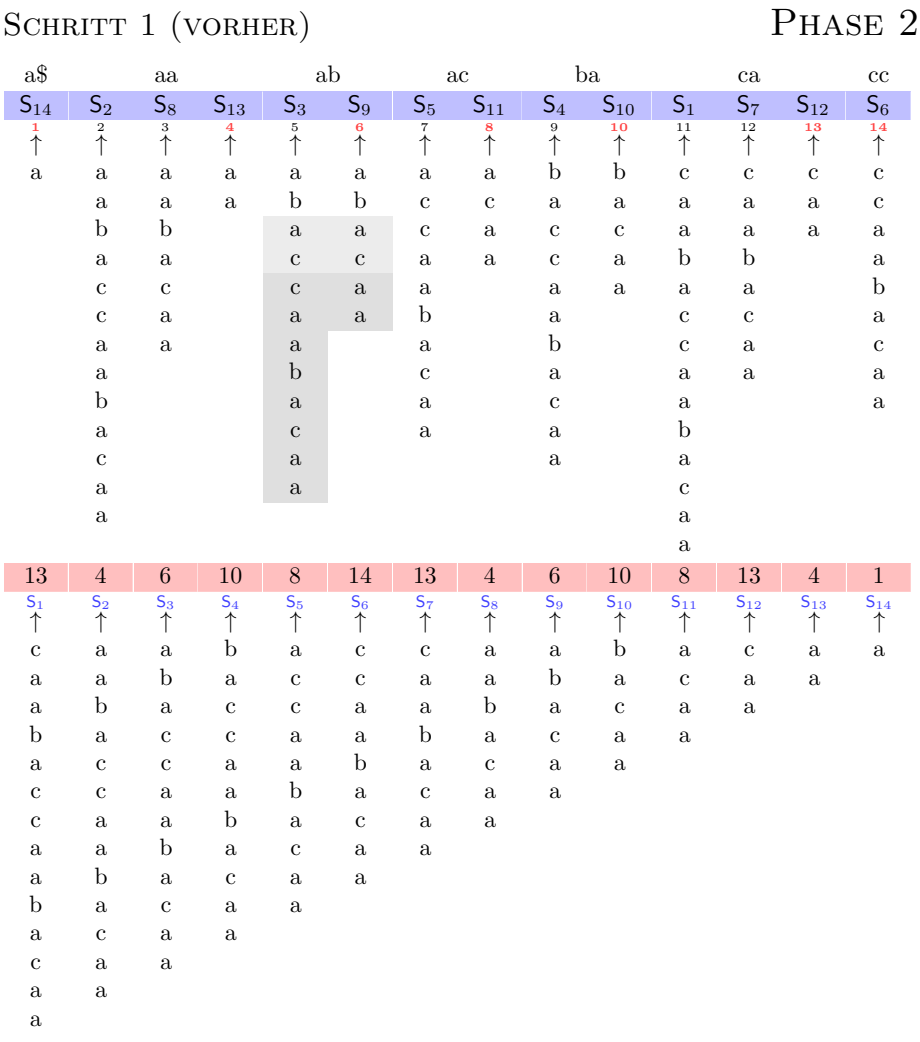


Abbildung 6.29: *Bucket-Pointer Refinement*: Phase 2, Schritt 1. SA und BPTR vor dem Sortierschritt.

SCHRITT 1 (NACHHER) PHASE 2

a$		aa		ab ac aa	ab ac ca	ac		ba		ca			cc
S_{14}	S_2	S_8	S_{13}	S_9	S_3	S_5	S_{11}	S_4	S_{10}	S_1	S_7	S_{12}	S_6
1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
a	a	a	a	a	a	a	a	b	b	c	c	c	c
	a	a	a	b	b	c	c	a	a	a	a	a	c
	b	b		a	a	c	a	c	c	a	a	a	a
	a	a		c	c	a	a	c	a	b	b		a
	c	c		a	c	a		a	a	a	a		b
	c	a		a	a	b		a		c	c		a
	a	a			a	a		b		c	a		c
	a				b	c		a		a	a		a
	b				a	a		c		a			a
	a				c	a		a		b			
	c				a			a		a			
	a				a					c			
	a									a			
										a			
13	4	6	10	8	14	13	4	5	10	8	13	4	1
S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
c	a	a	b	a	c	c	a	a	b	a	c	a	a
a	a	b	a	c	c	a	a	b	a	c	a	a	
a	b	a	c	c	a	a	b	a	c	a	a		
b	a	c	c	a	a	b	a	c	a	a			
a	c	c	a	a	b	a	c	a	a				
c	c	a	a	b	a	c	a	a					
c	a	a	b	a	c	a	a						
a	a	b	a	c	a	a							
a	b	a	c	a	a								
b	a	c	a	a									
a	c	a	a										
c	a	a											
a	a												
a													

Abbildung 6.30: *Bucket-Pointer Refinement*: Phase 2, Schritt 1. SA und BPTR nach dem Sortierschritt.

Die Verfeinerung erfolgt innerhalb eines Buckets Schrittweise mit einer Schrittgröße, die der zuvor gewählten Tiefe von Bucketsort entspricht. In diesem Beispiel wird zuerst der Bucket b_{ab} verfeinert. Aus der Bucket-Pointer Tabelle lässt sich nach einem Rekursionsschritt ablesen, dass die beiden darin enthaltenen Suffixe in ihrer Reihenfolge vertauscht werden müssen. Danach ist der Bucket fertig sortiert und die Bucket-Pointer werden an die neue Sortierung angepasst.

Schritt 2 (vorher)

a$		aa		ab ac aa	ab ac ca	ac		ba		ca			cc
S_{14}	S_2	S_8	S_{13}	S_9	S_3	S_5	S_{11}	S_4	S_{10}	S_1	S_7	S_{12}	S_6
1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	a	a	a	a	a	a	a	b	b	c	c	c	c
	a	a	a	b	b	c	c	a	a	a	a	a	c
	b	b		a	a	c	a	c	c	a	a	a	a
	a	a		c	c	a	a	c	a	b	b		a
	c	c		a	c	a		a	a	a	a		b
	c	a		a	a	b		a		c	c		a
	a	a			a	a		b		c	a		c
	a				b	c		a		a	a		a
	b				a	a		c		a			a
	a				c	a		a		b			
	c				a			a		a			
	a				a					c			
	a									a			
13	4	6	10	8	14	13	4	5	10	8	13	4	1
S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}
c	a	a	b	a	c	c	a	a	b	a	c	a	a
a	a	b	a	c	c	a	a	b	a	c	a	a	
a	b	a	c	c	a	a	b	a	c	a	a		
b	a	c	c	a	a	b	a	c	a	a			
a	c	c	a	a	b	a	c	a	a				
c	c	a	a	b	a	c	a	a					
c	a	a	b	a	c	a	a						
a	a	b	a	c	a	a							
a	b	a	c	a	a								
b	a	c	a	a									
a	c	a	a										
c	a	a											
a	a												
a													

Abbildung 6.31: *Bucket-Pointer Refinement*: Phase 2, Schritt 2. SA und BPTR vor dem Sortierschritt.

SCHRITT 2 (NACHHER)

a$		aa		ab ac aa	ab ac ca	ac aa	ac ca	ba		ca		cc
S_{14}	S_2	S_8	S_{13}	S_9	S_3	S_{11}	S_5	S_4	S_{10}	S_1	S_7	S_{12}	S_6
1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
a	a	a	a	a	a	a	a	b	b	c	c	c	c
	a	a	a	b	b	c	c	a	a	a	a	a	c
	b	b		a	a	a	c	c	c	a	a	a	a
	a	a		c	c	a	a	c	a	b	b		a
	c	c		a	c		a	a	a	a	a		b
	c	a		a	a		b	a		c	c		a
	a	a			a		a	b		c	a		c
	a				b		c	a		a	a		a
	b				a		a	c		a			a
	a				c		a	a		b			
	c				a			a		a			
	a				a					c			
	a									a			
13	4	6	10	8	14	13	4	5	10	7	13	4	1
S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
c	a	a	b	a	c	c	a	a	b	a	c	a	a
a	a	b	a	c	c	a	a	b	a	c	a	a	
a	b	a	c	c	a	a	b	a	c	a	a		
b	a	c	c	a	a	b	a	c	a	a			
a	c	c	a	a	b	a	c	a	a				
c	c	a	a	b	a	c	a	a					
c	a	a	b	a	c	a	a						
a	a	b	a	c	a	a							
a	b	a	c	a	a								
b	a	c	a	a									
a	c	a	a										
c	a	a											
a	a												
a													

Abbildung 6.32: *Bucket-Pointer Refinement*: Phase 2, Schritt 2. SA und BPTR nach dem Sortierschritt.

Anschließend wird der Bucket b_{ac} sortiert (Abbildungen 6.31 und 6.32), dessen Suffixe sich bereits nach einem gemeinsamen Präfix der Länge 2 unterscheiden. Aus der Bucket-Pointer Tabelle kann damit direkt abgelesen werden, in welchen Buckets sich die korrespondierenden Suffixe befinden, woraus dann die Reihenfolge abgeleitet wird. Das Vertauschen von S_5 und S_{11} führt zur richtigen Reihenfolge.

Die darauf folgende Sortierung des Buckets b_{ba} erfolgt analog (Abbildungen 6.33 und 6.34).

Schritt 3 (vorher)

a$		aa		ab ac aa	ab ac ca	ac aa	ac ca	ba		ca		cc	
S_{14}	S_2	S_8	S_{13}	S_9	S_3	S_{11}	S_5	S_4	S_{10}	S_1	S_7	S_{12}	S_6
1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	a	a	a	a	a	a	a	b	b	c	c	c	c
	a	a	a	b	b	c	c	a	a	a	a	a	c
	b	b		a	a	a	c	c	c	a	a	a	a
	a	a		c	c	a	a	c	a	b	b		a
	c	c		a	c		a	a	a	a	a		b
	c	a		a	a		b	a		c	c		a
	a	a			a		a	b		c	a		c
	a				b		c	a		a	a		a
	b				a		a	c		a			a
	a				c		a	a		b			
	c				a			a		a			
	a				a					c			
	a									a			
13	4	6	10	8	14	13	4	5	10	7	13	4	1
S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}
c	a	a	b	a	c	c	a	a	b	a	c	a	a
a	a	b	a	c	c	a	a	b	a	c	a	a	
a	b	a	c	c	a	a	b	a	c	a	a		
b	a	c	c	a	a	b	a	c	a	a			
a	c	c	a	a	b	a	c	a	a				
c	c	a	a	b	a	c	a	a					
c	a	a	b	a	c	a	a						
a	a	b	a	c	a	a							
a	b	a	c	a	a								
b	a	c	a	a									
a	c	a	a										
c	a	a											
a	a												
a													

Abbildung 6.33: *Bucket-Pointer Refinement*: Phase 2, Schritt 3. SA und BPTR vor dem Sortierschritt.

SCHRITT 3 (NACHHER)

a$		aa		ab ac aa	ab ac ca	ac aa	ac ca	ba ca	ba cc		ca		cc
S_{14}	S_2	S_8	S_{13}	S_9	S_3	S_{11}	S_5	S_{10}	S_4	S_1	S_7	S_{12}	S_6
1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
a	a	a	a	a	a	a	a	b	b	c	c	c	c
	a	a	a	b	b	c	c	a	a	a	a	a	c
	b	b		a	a	a	c	c	c	a	a	a	a
	a	a		c	c	a	a	a	c	b	b		a
	c	c		a	c		a	a	a	a	a		b
	c	a		a	a		b		a	c	c		a
	a	a			a		a		b	c	a		c
	a				b		c		a	a	a		a
	b				a		a		c	a			a
	a				c		a		a	b			
	c				a				a	a			
	a				a					c			
	a									a			
13	4	6	10	8	14	13	4	5	9	7	13	4	1
S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
c	a	a	b	a	c	c	a	a	b	a	c	a	a
a	a	b	a	c	c	a	a	b	a	c	a	a	
a	b	a	c	c	a	a	b	a	c	a	a		
b	a	c	c	a	a	b	a	c	a	a			
a	c	c	a	a	b	a	c	a	a				
c	c	a	a	b	a	c	a	a					
c	a	a	b	a	c	a	a						
a	a	b	a	c	a	a							
a	b	a	c	a	a								
b	a	c	a	a									
a	c	a	a										
c	a	a											
a	a												
a													

Abbildung 6.34: *Bucket-Pointer Refinement*: Phase 2, Schritt 3. SA und BPTR nach dem Sortierschritt.

Zuletzt werden in den Schritten 4 und 5 (Abbildungen 6.35 und 6.36 sowie Abbildungen 6.37 und 6.38) die Buckets b_{aa} und b_{ca} sortiert. Beide bestehen aus jeweils drei Suffixen, die sich alle bereits eindeutig anhand der Positionen in der nebenstehenden Tabelle in Buckets der Größe 1 einsortieren lassen. Nach Schritt 5 existieren schließlich nur noch Buckets der Größe 1 und das Suffix-Array ist damit fertig sortiert. An dieser Stelle verwirft der Algorithmus das Array BPTR und gibt SA als Lösung aus.

Schritt 4 (vorher)

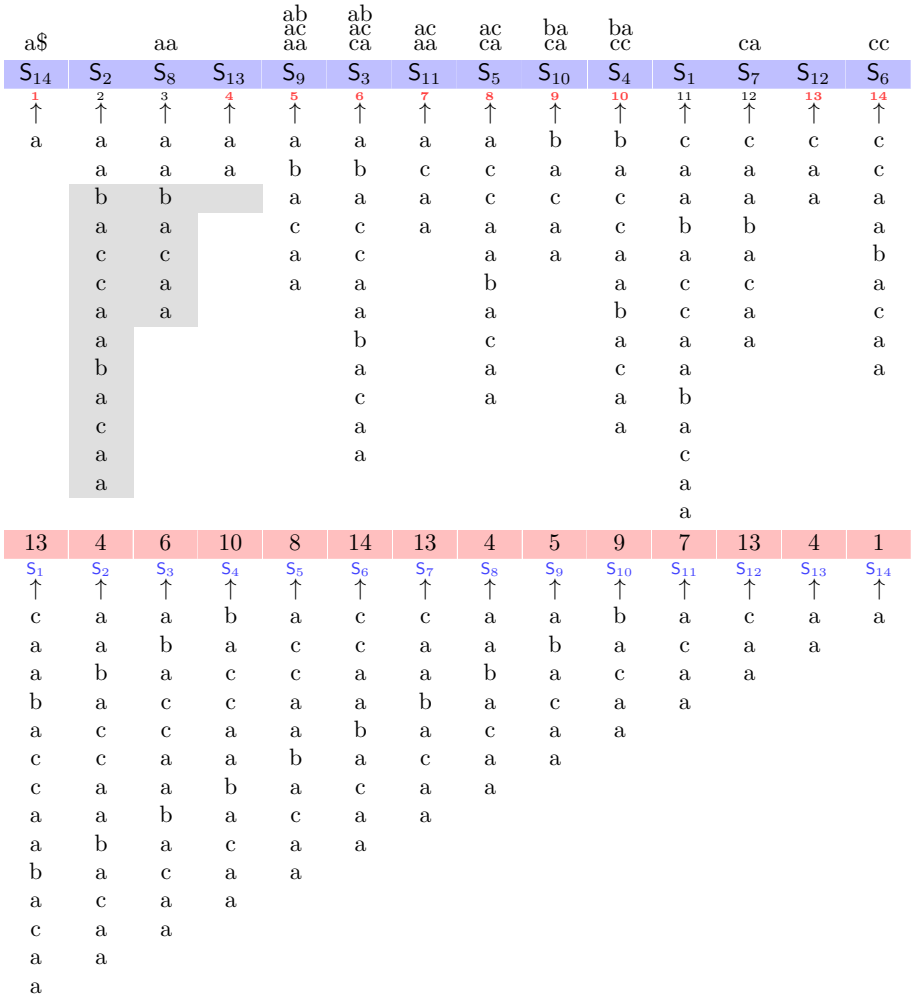


Abbildung 6.35: *Bucket-Pointer Refinement*: Phase 2, Schritt 4. SA und BPTR vor dem Sortierschritt.

Schritt 4 (nachher)

a$	aa $$	aa ba ca	aa ba cc	ab ac aa	ab ac ca	ac aa	ac ca	ba ca	ba cc		ca		cc
S_{14}	S_{13}	S_8	S_2	S_9	S_3	S_{11}	S_5	S_{10}	S_4	S_1	S_7	S_{12}	S_6
1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	a	a	a	a	a	a	a	b	b	c	c	c	c
	a	a	a	b	b	c	c	a	a	a	a	a	c
		b	b	a	a	a	c	c	c	a	a	a	a
		a	a	c	c	a	a	a	c	b	b		a
		c	c	a	c		a	a	a	a	a		b
		a	c	a	a		b		a	c	c		a
		a	a		a		a		b	c	a		c
			a		b		c		a	a	a		a
			b		a		a		c	a			a
			a		c		a		a	b			
			c		a					a			
			a		a					c			
			a							a			
										a			
13	4	6	10	8	14	13	3	5	9	7	13	2	1
S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}
c	a	a	b	a	c	c	a	a	b	a	c	a	a
a	a	b	a	c	c	a	a	b	a	c	a	a	
a	b	a	c	c	a	a	b	a	c	a	a		
b	a	c	c	a	a	b	a	c	a	a			
a	c	c	a	a	b	a	c	a	a				
c	c	a	a	b	a	c	a	a					
c	a	a	b	a	c	a	a						
a	a	b	a	c	a	a							
a	b	a	c	a	a								
b	a	c	a	a									
a	c	a	a										
c	a	a											
a	a												
a													

Abbildung 6.36: *Bucket-Pointer Refinement*: Phase 2, Schritt 4. SA und BPTR nach dem Sortierschritt.

Schritt 5 (vorher)

a$	aa $$	aa ba ca	aa ba cc	ab ac aa	ab ac ca	ac aa	ac ca	ba ca	ba cc		ca		cc
S_{14}	S_{13}	S_8	S_2	S_9	S_3	S_{11}	S_5	S_{10}	S_4	S_1	S_7	S_{12}	S_6
1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
a	a	a	a	a	a	a	a	b	b	c	c	c	c
	a	a	a	b	b	c	c	a	a	a	a	a	c
		b	b	a	a	a	c	c	c	a	a	a	a
		a	a	c	c	a	a	a	c	b	b		a
		c	c	a	c		a	a	a	a	a		b
		a	c	a	a		b		a	c	c		a
		a	a		a		a		b	c	a		c
			a		b		c		a	a	a		a
			b		a		a		c	a			a
			a		c		a		a	b			
			c		a				a	a			
			a		a					c			
			a							a			
										a			
13	4	6	10	8	14	13	3	5	9	7	13	2	1
s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
c	a	a	b	a	c	c	a	a	b	a	c	a	a
a	a	b	a	c	c	a	a	b	a	c	a	a	
a	b	a	c	c	a	a	b	a	c	a	a		
b	a	c	c	a	a	b	a	c	a	a			
a	c	c	a	a	b	a	c	a	a				
c	c	a	a	b	a	c	a	a					
c	a	a	b	a	c	a	a						
a	a	b	a	c	a	a							
a	b	a	c	a	a								
b	a	c	a	a									
a	c	a	a										
c	a	a											
a	a												
a													

Abbildung 6.37: *Bucket-Pointer Refinement*: Phase 2, Schritt 5. SA und BPTR vor dem Sortierschritt.

SCHRITT 5 (NACHHER)

a$	aa $$	aa ba ca	aa ba cc	ab ac aa	ab ac ca	ac aa	ac ca	ba ca	ba cc	ca a$	ca ab ac aa	ca ab ac ca	cc
S_{14}	S_{13}	S_{8}	S_{2}	S_{9}	S_{3}	S_{11}	S_{5}	S_{10}	S_{4}	S_{12}	S_{7}	S_{1}	S_{6}
1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	a	a	a	a	a	a	a	b	b	c	c	c	c
	a	a	a	b	b	c	c	a	a	a	a	a	c
		b	b	a	a	a	c	c	c	a	a	a	a
		a	a	c	c	a	a	a	c		b	b	a
		c	c	a	c		a	a	a		a	a	b
		a	c	a	a		b		a		c	c	a
		a	a		a		a		b		a	c	c
			a		b		c		a		a	a	a
			b		a		a		c			a	a
			a		c		a		a			b	
			c		a				a			a	
			a		a							c	
			a									a	
												a	
13	4	6	10	8	14	12	3	5	9	7	11	2	1
S_{1}	S_{2}	S_{3}	S_{4}	S_{5}	S_{6}	S_{7}	S_{8}	S_{9}	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}
c	a	a	b	a	c	c	a	a	b	a	c	a	a
a	a	b	a	c	c	a	a	b	a	c	a	a	
a	b	a	c	c	a	a	b	a	c	a	a		
b	a	c	c	a	a	b	a	c	a	a			
a	c	c	a	a	b	a	c	a	a				
c	c	a	a	b	a	c	a	a					
c	a	a	b	a	c	a	a						
a	a	b	a	c	a	a							
a	b	a	c	a	a								
b	a	c	a	a									
a	c	a	a										
c	a	a											
a	a												
a													

Abbildung 6.38: *Bucket-Pointer Refinement*: Phase 2, Schritt 5. SA und BPTR nach dem Sortierschritt.

6.6.3 Effizienz

Im Bezug auf die Effizienz des hier beschriebenen Algorithmus muss zwischen der asymptotischen Worst-Case Analyse und der Effizienz in realen Anwendungsszenarien unterschieden werden. Es existieren bereits andere Verfahren mit sehr guten theoretischen Laufzeitschranken von $\Theta(n)$ [29] (DC3, Abschnitt 6.4) sowie praxisnähere Algorithmen mit einer Laufzeitkomplexität von $\mathcal{O}(n \log n)$ [23] (DivSufSort, Abschnitt 6.9), wobei n die Länge des Eingabestrings ist. Da es aber in der Praxis durchaus vorstellbar ist, dass Algorithmen mit schlechterer asymptotischer Laufzeit für reale Anwendungszwecke schneller sind, werden wir uns in Kapitel 7 mit praktischen Laufzeitmessungen im Vergleich zu anderen Algorithmen auseinandersetzen.

Gerade in der Praxis ist allerdins nicht nur die Laufzeitkomplexität, sondern auch die Speicherkomplexität von hoher Bedeutung. Die nachfolgende Analyse beschäftigt sich daher insbesondere auch mit dem Speicherbedarf des *Bucket-Pointer Refinement*-Algorithmus.

Worst-Case Analyse

Die Worst-Case Analyse des Algorithmus kann für beide Phasen separat erfolgen, da die asymptotische Schranke nur durch die schlechtere Schranke der beiden Phasen festgelegt wird. Wir betrachten daher zunächst die Speicher- und Laufzeitkomplexität der ersten Phase, bevor wir die zweite Phase auf ähnliche Art und Weise untersuchen. Für die angegebenen Laufzeitschranken gibt die Größe n dabei immer die Länge des Eingabestrings an.

Phase 1 In der ersten Phase des Algorithmus wird die initiale Einteilung der Suffixe in Level-d-Buckets bestimmt. Gemäß der Beschreibung von Schürmann und Stoye [49, Abschnitt 3.3] wird dazu eine Tabelle *bkt* (s. Abbildung 6.20) angelegt. Bedingt durch die Größe des Alphabets und die Länge d der Präfixe fällt auf, dass die Tabelle Speicher in der Größe von $\mathcal{O}(|\Sigma|^d)$ benötigt. Unter der Annahme, dass die Allokation von Speicher der Größe n einen Laufzeitaufwand von $\Theta(n)$ verursacht, bildet die Speicherkomplexität zugleich eine untere Schranke für die Laufzeitkomplexität. Dennoch geben Schürmann und Stoye in ihrer Analyse eine Worst-Case Laufzeit von $\mathcal{O}(n)$ für die erste Phase an [49, Kapitel 3.2]. Um zumindest in der Theorie eine lineare Laufzeit in Abhängigkeit von n zu garantieren, kann angenommen werden, dass $|\Sigma|$ und d konstant oder zumindest unabhängig von n sind. Falls d allerdings von n abhängig gewählt wird, so sind einige andere Faktoren entscheidend, auf welche im Folgenden eingegangen wird.

Zunächst ist zu beachten, dass nicht der gesamte allokierte Speicher auch verwendet werden muss: Die Verarbeitung der Eingabe erfolgt laut Dokumentation des Algorithmus in dieser Phase in einer konstanten Anzahl von Iterationen über den Eingabestring[6] T bzw. T^+. Obwohl das Array *bkt* Platz für alle mög-

[6] Laut Schürmann und Stoye [49, Kapitel 3.3] besteht die erste Phase von *Bucket-Pointer Refinement* lediglich aus drei Iterationen über den Eingabestring, was eine lineare Laufzeit-

lichen Präfixe der Länge d bietet, müssen nur die Einträge verwendet werden, zu denen tatsächlich entsprechende Teilstrings in T^+ existieren. Die Anzahl solcher Präfixe ist allerdings durch $\mathcal{O}(n)$ beschränkt, woraus eine lineare Laufzeit resultiert, sofern es die Implementierung erlaubt, Speicher beliebiger Größe in konstanter Zeit zu allokieren.

Alternativ ist auch die Verwendung einer geeigneten Hashtabelle denkbar, um den exponentiellen Speicherbedarf zu vermeiden. Ein solcher Ansatz bietet – wenn auch im ursprünglichen Algorithmus nicht vorgesehen – für große Alphabete schon bei kleiner Konstante d die einzige Möglichkeit, den Algorithmus auf gängiger Hardware angemessen speicherschonend betreiben zu können.

Phase 2 Für die asymptotische Laufzeit der zweiten Phase wurde von Schürmann und Stoye nur eine auf groben Abschätzungen basierende Schranke von $\mathcal{O}(n^2)$ angegeben [49, Kapitel 3.2]. Die entsprechende Analyse beruht auf der Annahme, dass in jeder Ebene der Rekursion insgesamt höchstens n Suffixe sortiert werden müssen, was eine Laufzeit von $\mathcal{O}(n \log n)$ zur Folge habe. Da die Anzahl der Rekursionsebenen offensichtlich durch $\frac{n}{d}$ beschränkt ist (bedingt durch die Erhöhung von *offset* um d pro Ebene), sei die Laufzeit der gesamgen Phase durch $\mathcal{O}(\frac{n}{d} \cdot n \log n)$ beschränkt. Es wird außerdem angegeben, dass eine Schranke von $\mathcal{O}(n^2)$ erreicht werden kann, wenn $d = \log n$ gesetzt wird.

Wir wollen uns diese Laufzeitschranke genauer ansehen. Zunächst fällt auf, dass für den Sortiervorgang in Phase 2 die Algorithmen *Insertionsort* und *Quicksort* verwendet werden. *Insertionsort* hat bekanntermaßen sowohl im besten Fall, als auch im durchschnittlichen Fall eine Laufzeit von $\mathcal{O}(n^2)$. Da dieser Algorithmus allerdings nur für Buckets der Größe 15 oder kleiner verwendet wird, ist in diesem Fall n durch 15 nach oben beschränkt, woraus sogar eine konstante Laufzeit resultiert. Interessanter ist die Analyse bei *Quicksort*: Zwar beträgt die Laufzeit dieses Verfahrens im durchschnittlichen Fall $\mathcal{O}(n \log n)$, im schlechtesten Fall kann die Laufzeit allerdings auch bis auf $\mathcal{O}(n^2)$ ansteigen. In einer Worst-Case Analyse muss also streng genommen eine quadratische Laufzeitschranke für den Sortiervorgang angenommen werden. Die von Schürmann und Stoye angenommene asymptotische Schranke von $\mathcal{O}(n \log n)$ kann tatsächlich erreicht werden, wenn für den Sortiervorgang *Mergesort* oder *Heapsort* verwendet wird. Da es in der Praxis dennoch vorteilhaft sein kann, *Quicksort* zu verwenden [25, Tabelle 1], ist es hier vertretbar, die Sortiervorgänge mit diesem Algorithmus durchzuführen und anzumerken, dass eine asymptotische Laufzeitschranke von $\mathcal{O}(n \log n)$ zumindest mit alternativen Sortieralgorithmen *möglich* ist. Wir gehen daher auch in der weiteren Analyse davon aus, dass Sortieren in $\mathcal{O}(n \log n)$ möglich ist.

---

schranke zur Folge hat. Bei genauerer Betrachtung der im Rahmen der Publikation veröffentlichten Implementierung (abrufbar unter `http://bibiserv.techfak.uni-bielefeld.de/bpr/`) fällt allerdings auf, dass tatsächlich auch eine vollständige Iteration über das Array *bkt* stattfindet, welche die Laufzeitschranke auf $\mathcal{O}(|\Sigma|^d)$ anhebt. d wird aber im weiteren Verlauf des Papers als logarithmisch abhängig von $|\Sigma|$ gewählt. Diese Iteration wird jedoch weder in der Beschreibung des Algorithmus erwähnt, noch in der zugehörigen Analyse berücksichtigt.

Hervorzuheben ist auch die Wahl von $d = \log n$, um insgesamt eine Laufzeit von $\mathcal{O}(n^2)$ zu erhalten. Den Parameter d in Abhängigkeit von n zu wählen, ist zwar grundsätzlich möglich, steht aber im Gegensatz zur in Phase 1 aufgestellten Annahme, d sei konstant. Betrachten wir nur konstante Werte für d, so fällt der Faktor $\frac{1}{d}$ in $\mathcal{O}(\frac{n}{d} \cdot n \log n)$ weg und wir erhalten eine Worst-Case Laufzeit von $\mathcal{O}(n^2 \log n)$.

Obwohl es mit der oben angewandten Methode zur Abschätzung der Laufzeit, welche sich auf die Tiefe der Rekursion und den maximalen Aufwand pro Ebene bezieht, scheinbar nicht möglich ist, kleinere obere Schranken als $\mathcal{O}(n^2 \log n)$ zu finden, konnte noch keine einfache Beispieleingabe gefunden werden, die tatsächlich eine derart hohe Laufzeit verursacht. Die größte Schwierigkeit dabei ist, dass der Algorithmus vergleichsweise undurchsichtig die Abhängigkeiten zwischen den Suffixen verwendet.

Gesamter Algorithmus In den beiden vorangegangenen Abschnitten wurde die Laufzeit der beiden Phasen einzeln analysiert. Eine Laufzeitschranke für den Gesamten Algorithmus ergibt sich durch das Maximum beider Komponenten. Da es aber für eine präzise Angabe notwendig ist, die Wahl des Parameters d festzulegen, betrachten wir zuerst einige Optionen. Im Vorfeld wurde bereits diskutiert, welche Auswirkungen auf die Laufzeit die Wahl von $d = \log n$ im Gegensatz zu einem konstanten Wert für d hat. Diese Unterschiede zeigt Tabelle 6.14. Für einen konstanten Wert d haben wir bereits in den beiden teilweisen

Phase	d konstant	$d = \log n$	$d = \log_{\lvert\Sigma\rvert} n$
Phase 1	$\mathcal{O}(\lvert\Sigma\rvert^d)$	$\mathcal{O}(\lvert\Sigma\rvert^{\log n}) = \mathcal{O}(n^{\log \lvert\Sigma\rvert})$	$\mathcal{O}(\lvert\Sigma\rvert^{\log_{\lvert\Sigma\rvert} n}) = \mathcal{O}(n^{\log_{\lvert\Sigma\rvert} \lvert\Sigma\rvert}) = \mathcal{O}(n)$
Phase 2	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}\left(\frac{n^2 \log n}{\log_{\lvert\Sigma\rvert} n}\right) = \mathcal{O}(n^2 \log \lvert\Sigma\rvert)$
Gesamt	$\mathcal{O}(\lvert\Sigma\rvert^d + n^2 \log n)$	$\mathcal{O}(n^{\max\{2, \log \lvert\Sigma\rvert\}})$	$\mathcal{O}(n^2 \log \lvert\Sigma\rvert)$

Tabelle 6.14: Laufzeitschranken für *Bucket-Pointer Refinement* in Abhängigkeit von d

Analysen gesehen, dass die Laufzeit polynomiell in $|\Sigma|$ und n ist. Allerdings ist festzustellen, dass sich bereits für kleine d eine sehr hohe reale Laufzeit ergeben kann. Für den Fall $d = \log n$, welcher von Schürmann und Stoye gewählt wurde, ist die Laufzeit zwar für feste Alphabetgrößen $|\Sigma|$ polynomiell in n. Für allgemeine Alphabete jedoch ist die Laufzeit nicht polynomiell.

Ein möglicher Ansatz um eine in n und $|\Sigma|$ polynomielle Laufzeitschranke zu erreichen kann gefunden werden, wenn d in Abhängigkeit der Alphabetgröße $|\Sigma|$ und der Eingabelänge n als $d = \log_{|\Sigma|} n$ gewählt wird. Da auf diese Option bisher noch nicht eingegangen wurde, sind in Tabelle 6.14 auch die separaten Schranken für die erste und zweite Phase angegeben. Der Parameter d ist hier so gewählt, dass das exponentielle Wachstum in Phase 1, wie es für $d = \log n$ auftritt, vermieden werden kann, wodurch dort eine asymptotische Laufzeit von $\mathcal{O}(n)$ erreicht wird. Die Laufzeit für die zweite Phase ist zwar etwas schlechter

als für $d = \log n$, da aber $|\Sigma| \leq n$ gilt, wird hier zumindest eine bessere Laufzeit erzielt, als für ein konstantes d.

In den praktischen Tests (Kapitel 7) wurde d dennoch nur in Abhängigkeit von $|\Sigma|$ und unabhängig von n gewählt, da die Größe der von unserem Testsystem verarbeitbaren Dateien nicht ausgereicht hat, um vom asymptotisch besseren Fall profitieren zu können.

Unterschiede zur Referenzimplementierung

Die Implementierung des *Bucket-Pointer Refinement* Algorithmus für das SACABench Framework erfolgte in erster Linie entlang der im Paper [49] beschriebenen Variante des Algorithmus. Daraus ergaben sich zunächst einige strukturelle Unterschiede des Quellcodes im Vergleich zur Referenzimplementierung des Autors. Erst nach der Fertigstellung einer lauffähigen und stabilen Implementierung wurden dann die von Schürmann beschriebenen Modifikationen [49, Kapitel 3] implementiert, die zur Verbesserung der praktischen Laufzeit beitragen sollen, ohne die asymptotische Laufzeit zu beeinflussen.

Erst im Anschluss daran fand ein direkter Vergleich der beiden Implementierungen statt, der einige algorithmische Unterschiede in der Umsetzung von Teilfunktionen offenbarte. Teilweise konnte die Laufzeit durch Anpassung derartiger Codestellen an das Original verringert werden, während sich in anderen Funktionen die neu implementierte Variante als effizienter herausstellte. In allen Fällen wurde dann jeweils die schnellere Methode beibehalten. Die wichtigsten dadurch entstandenen Unterschiede zwischen der Referenzimplementierung und derer der Projektgruppe werden in diesem Kapitel aufgeführt.

Seward's copy Beide Varianten des Algorithmus verwenden die von Seward [50] beschriebene Copy-Technik, um aus bereits sortierten Einträgen die Reihenfolge von bis dahin unsortierten Suffixen abzuleiten. Der Unterschied besteht

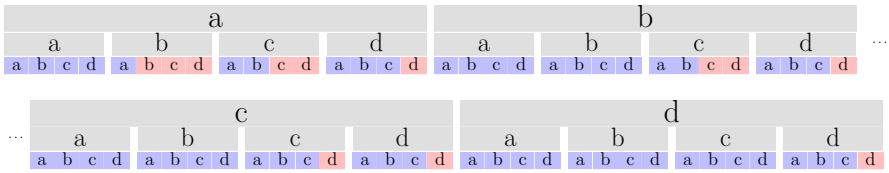


Abbildung 6.39: Level-1 bis Level-3 Buckets über dem Alphabet $\{a, b, c, d\}$. Rot markierte Buckets müssen mit Quicksort sortiert werden. Blau markierte Buckets können in Linearzeit induziert werden.

lediglich in der Reihenfolge, in der die Sortieroperationen und das Induzieren angewendet werden: Während Schürmann nach jedem Durchlauf durch einen Top-Level Bucket direkt den Rest dieses Buckets induziert und danach erneut die Bucket-Pointer aktualisiert, werden bei der SACABench-Implementierung zuerst alle Top-Level Buckets vorsortiert, bevor im Anschluss in einem Durchlauf alle verbleibenden Indizes zusammen induziert werden. Letzteres erspart im

```
# m is assumed to be |Σ|
for x in 0..m-1
  for y in x+1..m-1
    for z in y..m-1
      quicksort on bucket xyz
      update bucket pointers
for x in 0..m
  seward-copy rest of bucket x
```

```
# m is assumed to be |Σ|
for x in 0..m-1
  for y in x+1..m-1
    for z in y..m-1
      quicksort on bucket xyz
      update bucket pointers
  seward-copy rest of bucket x
  update bucket pointers
```

Algorithmus 6.10: Verwendung der Copy Technik der SACABench-Version (links) und bei Schürmann (rechts)

Vergleich zur Referenzimplementierung die Aktualisierung der Bucket-Pointer nach dem Induzieren, welche als Random Access auf das Array einen messbaren Teil der Laufzeit verursachen. Die Aktualisierung kann weg fallen, da die Bucket-Pointer lediglich als Sortierschlüssel für Quicksort verwendet werden, was aber in dieser Variante zum Zeitpunkt der Induzierung schon abgeschlossen werden.

Zu erwarten ist, dass bedingt durch die zum Zeitpunkt der späteren Quicksort-Aufrufe im Vergleich zur Referenzimplentierung noch ungenaueren Sortierschlüssel mehr Sortieraufrufe durchgeführt werden müssen, was eine längere Laufzeit zur Folge hat. Ein solcher Effekt war jedoch für Eingabegrößen bis zu mehreren hundert Megabyte nicht messbar.

Adressierung der Arrays Die Referenzimplementierung von Schürmann verwendet für die Einträge im Suffix-Array 64 Bit lange Zeiger auf die zugehörige Speicherstelle im Eingabetext, an der das jeweilige Suffix beginnt. Analog dazu werden im Bucket-Pointer Array BPTR 64 Bit lange Zeiger auf die Einträge im Suffix-Array gespeichert. Alle Zugriffe auf diese Arrays finden über direkte Pointer-Arithmetik statt.

In der SACABench-Implementierung des Algorithmus werden für alle Zugriffe auf Arrays durch Indexzugriffe durchgeführt. Das ermöglicht es, je nach Größe der Eingabe die Bitbreite für die Adressierung dynamisch festzulegen. Da für die meisten in annehmbarer Zeit zu verarbeitenden Eingaben eine Größe von 32 Bit ausreicht, kann in den meisten Fällen der Speicherbedarf sowohl für das Suffix-Array als auch für das Bucket-Pointer Array halbiert werden. Dadurch halbiert sich (abgesehen von konstantem Zusatzspeicher) der gesamte Speicherbedarf des Algorithmus auf die Hälfte. Ein messbarer Unterschied in der Laufzeit hat sich dadurch jedoch nicht ergeben.

Sortieralgorithmus für Schlüssel In der Referenzimplementierung verwendet Schürmann eine Version von Quicksort, um die Suffixe innerhalb der Buckets anhand ihrer Schlüssel zu sortieren. Für kleine Buckets mit einer Größe von maximal 15 Elementen wird dort Insertionsort verwendet. Einen deutlichen Vorteil gegenüber den von uns getesteten Varianten von Quicksort bietet in den meisten Fällen der Algorithmus In-Place Parallel Super Scalar Samplesort (IPS^{4}o [4]). Die Verwendung dieses Algorithmus hat die tatsächliche Laufzeit auf natürlichsprachlichen Eingaben von über 50 MB ausreichend beschleunigt, um eine schnellere Verarbeitung zu ermöglichen als die Referenzimplemtierung. Bei repetitiven Texten hingegen fällt der Vorteil von Quicksort und IPS^{4}o deutlich geringer aus. Ein Unterschied ist hier nicht mehr messbar.

6.6.4 Maßnahmen zur Parallelisierung

Die *Bucket-Pointer Refinement* Variante von Schürmann und Stoye [49] beschreibt einen komplett sequentiellen Algorithmus zur Konstruktion von Suffix-Arrays. Die Autoren haben zudem nicht die Intention geäußert, den Algorithmus durch die Verwendung entsprechender Teilalgorithmen grundlegend parallelisierbar zu konzipieren. Deshalb ist es Teil der Zielsetzung der Projektgruppe, *Bucket-Pointer Refinement* auf Parallelisierbarkeit zu untersuchen und mit bekannten anderen Verfahren zu vergleichen.

Bei genauerer Betrachtung des Algorithmus fällt auf, dass sich einige der eingesetzten Komponenten durch parallele Algorithmen ersetzen lassen, was die Laufzeit auf Multicore-Systemen positiv beeinflusst. Andere Teile des Algorithmus hingegen sind nur für den sequentiellen Einsatz konzipiert und konnten nicht durch vergleichbare parallele Methoden ersetzt werden.

Bucketsort Die initiale Sortierung des *Bucket-Pointer Refinement* Algorithmus geschieht in der sequentiellen Version mittels Bucketsort (Abschnitt 4.1.4). Bucketsort selbst ist in mehrere Phasen gegliedert, von denen sich die erste und die letzte Phase parallelisieren lassen. In der ersten Phase bestimmt Bucketsort die absoluten Häufigkeiten $h(p)$ für alle Präfixe $p \in \{\Sigma \cup \$\}^k$ der Länge k. Dies geschieht in der parallelen Version, indem jeder Thread $i \in \{0, \ldots, t-1\}$ einen gleich großen Abschnitt des Eingabetextes T zugewiesen bekommt und für diesen lokal die Häufigkeiten $h_i(p)$ bestimmt. Zentral werden daraus im Anschluss zunächst die globalen Häufigkeiten $h(p) = \sum_{i=mid}^{t-1} h_i(p)$ und darauf basierend die Startpositionen $H(p) = \sum_{p'<p} h(p')$ der Buckets im SA mittels einer Präfixsumme bestimmt. Zusätzlich zur Berechnung der Startpositionen bestimmt die parallele Variante von Bucketsort für jeden Bucket eine Thread-lokale Startposition $H_i(p) = H(p) + \sum_{j=mid}^{i-1} h_j(p)$ basierend auf den lokal gezählten Häufigkeiten des jeweiligen Elements. In einem erneuten parallelen Scan befüllt dann jeder Thread den ihm zugewiesenen Teil jedes Buckets mit den eingelesenen Substrings der Eingabe.

Vergleichsbasierte Verfeinerung der Buckets Die vergleichsbasierte Verfeinerung zweier Buckets b_p und b_q geschieht intuitiv unabhängig voneinander, was eine hohe Parallelisierbarkeit dieser Aufgabe vermuten lässt. Schließlich sind zu jedem Zeitpunkt alle Buckets im Suffix-Array disjunkt, was ein konfliktfreies paralleles Sortieren auf Bucketebene zulassen würde. Tatsächlich werden aber als Sortierschlüssel die Einträge des Bucket-Pointer Arrays BPTR verwendet, welche nach Abschluss jedes Sortiervorgangs aktualisiert werden.

Während des Sortiervorgangs auf einem Bucket b_p sind daher Lesezugriffe auf beliebige Elemente in BPTR erforderlich und nicht nur auf solche, die mit Elementen aus b_p korrespondieren. Insbesondere kann es dabei vorkommen, dass Einträge aus BPTR gelesen werden, die zu Elementen aus b_q gehören. Kommt es nun vor, das während eines laufenden Sortiervorgangs auf b_p der Sortiervorgang auf b_q von einem anderen Thread beendet wird, so aktualisiert dieser Thread die zu b_q gehörigen Einträge in BPTR. Dies führt im Allgemeinen zu einer fehlerhaften Sortierung von b_p, da sich während des laufenden Vorgangs die Sortierschlüssel verändern.

Um diesem unerwünschten Effekt vorzubeugen, kann jeder Thread auf einer lokalen Kopie von BPTR arbeiten. Die Verfeinerung de Buckets ist damit konfliktfrei und folglich korrekt. Dieser Ansatz hat allerdings neben einem deutlich erhóhten Speicherbedarf den Nachteil, dass die Verfeinerung der Sortierschlüssel nur noch dann zur Beschleunigung eines Sortiervorgangs beitragen kann, wenn dieser vom selben Thread bearbeitet wird. Durch eine regelmäßige Synchronisation der BPTR Arrays verschiedener Threads kann letzteres Problem zwar umgangen werden, der erhöhte Synchronisationsaufwand führt allerdings durch häufiges Kopieren von Speicherinhalten zu keiner Verbesserung der Performance. Beide Methoden zur Parallelisierung führen unter den Testbedingungen dazu, dass die parallele Variante des Algorithmus unter Verwendung dieses Verfahrens weitaus ineffizienter wird als die sequentielle Variante.

Die naive Methode zur Parallelisierung ist die Verwendung eines parallelen Sortierverfahrens anstelle eines sequentiellen Verfahrens. Die einzelnen Buckets werden auf diese Weise weiterhin sequentiell verarbeitet, der Sortiervorgang innerhalb eines Buckets profitiert aber möglicherweise von der Verwendung mehrerer Threads. Dazu wird der sequentielle IPS^{4}o [4] durch die entsprechende parallele Version ersetzt. Um überflüssige Synchronisation zwischen Threads zu vermeiden, werden nur Buckets mit mehr als 5000 Elementen parallel sortiert.

Copy-Technik Die Copy-Technik [50] (Abschnitt 6.6.3) dient in der sequentiellen Variante von *Bucket-Pointer Refinement* dazu, den hohen Aufwand für vergleichsbasiertes Sortieren auf großen Teilen des Suffix-Arrays zu vermeiden, indem die Reihenfolge bislang unsortierter Buckets anhand der Reihenfolge der Suffixe bereits vollständig sortierter Buckets induziert wird. Da dieser Algorithmus in der Variante der Tiefe 2 (also mit Induzierung Vor-Vorgänger-Substrings) zwischen den beiden Schritten konfligierende Zugriffe auf das Suffix-Array beinhaltet (Abbildung 6.40), kann keine triviale Parallelisierung auf Basis der Schritte (Left-to-Right Scan und Right-to-Left Scan jedes Level-1-Buckets) vorgenommen werden. Die grundsätzlichen Abhängigkeiten erlauben es nicht, dass der Right-to-Left Scan $rtl(\mathsf{b}_p)$ auf einem Bucket b_p vor dem Right-to-Left Scan aller Buckets $\mathsf{b}_{p'}$ mit $p' < p$ durchgeführt wird. Jeder Right-to-Left Scan basiert somit auf allen Right-to-Left Scans lexikographisch kleinerer Level-1-Buckets:

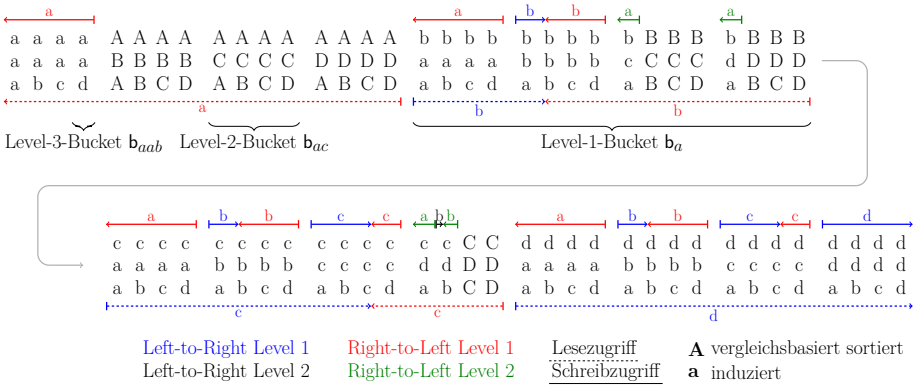


Abbildung 6.40: Level-1 bis Level-3 Buckets über dem Alphabet $\{a, b, c, d\}$. Buckets in Großbuchstaben sind zu Beginn des Copy-Schrittes bereits vergleichsbasiert sortiert worden. Die Grafik zeigt mögliche Konflikte zwischen Lese- und Schreibzugriffen.

$$\forall p : \ \forall p' < p : \ rtl(\mathsf{b}_{p'}) \prec rtl(\mathsf{b}_p)$$

Eine ähnliche Abhängigkeit ist für Left-to-Right Scans zu beobachten: Ein Left-To-Right Scan $ltr(\mathsf{b}_p)$ auf einem Bucket b_p kann nur dann konfliktfrei durchgeführt werden, wenn zuvor der zugehörige Right-To-Left Scan auf b_p sowie alle Left-to-Right Scans auf $ltr(\mathsf{b}_{p'})$ mit $p' < p$ beendet sind:

$$\forall p : \ \forall p' < p : \ ltr(\mathsf{b}_{p'}) \prec ltr(\mathsf{b}_p) \wedge ltr(\mathsf{b}_{p'}) \prec ltr(\mathsf{b}_p)$$

Aus beiden Abhängigkeiten folgt eine nicht parallelisierbare Sequenz von Schritten. Eine parallele Bearbeitung eines einzelnen Scans ist darüber hinaus auch nicht möglich, da jede Schreibposition von den zuvor gelesenen Elementen abhängt.

Eine Parallelisierung der Copy-Phase kann erfolgen, wenn diese statt mit der Tiefe 2 nur mit der Tiefe 1 durchgeführt wird. Die dadurch entstehende

Laufzeit für zusätzlichen Sortieraufwand ist jedoch höher als die durch parallele Verarbeitung eingesparte Laufzeit der Copy-Phase.

6.7 mSufSort

6.7.1 Einleitung

In diesem Abschnitt wird ein Algorithmus, oder vielmehr eine Sammlung aus Algorithmen, vorgestellt, die in „An Efficient, Versatile Approach to Suffix Sorting“von Maniscalco et al. [37] beschrieben sind. Dabei liegt der Fokus nicht auf der exakten Wiedergabe des Papers, sondern auf weiterführenden Beispielen und Erläuterungen, sowie einer Erweiterung des dort bereits angegebenen Pseudocodes. Zudem wird auf die Implementierung im Rahmen der Projektgruppe eingegangen und ein Beispiel angehängt, das zum besseren Verständnis beitragen sollte. In einem weiteren Unterkapitel wird auf Möglichkeiten der naiven Parallelisierung eingegangen.

Bei der Entwicklung dieses Algorithmus wurden von Maniscalco et al. drei Ziele verfolgt:
1. **Kurze Laufzeit** - der Algorithmus soll mit anderen SACAs mithalten können. Er soll außerdem besonders schnell die Burrows-Wheeler Transformation [12] berechnen können, die zur verlustfreien Kompression von Texten verwendet werden kann.
2. **Wenig Platzverbrauch** im Arbeitsspeicher - Für Suffix Arrays existiert der Begriff *lightweight* (leichtgewichtig), eingeführt von Manzini und Ferragina [38]. Er wird für SACAs verwendet, die nur einen Platzverbrauch von unter $6n$ Bytes bei einer Länge des Eingabe-Strings von n Zeichen haben. Dieser Platz wird für das Suffix Array ($4n$ Byte für $n < 2^{32}$) und zusätzlichen Arbeitsplatz verwendet.
3. **Sensitivität bezüglich dem Alphabet** - der Algorithmus sollte auch für große Alphabete Σ funktionieren. Viele SACAs beschränken sich auf $|\Sigma| \leq 256$, damit auch $|\Sigma|^2$ noch eine händelbare Größe darstellt. In einigen Situationen ist das Alphabet jedoch größer, zB. für wortbasierte Burrows-Wheeler Transformation ist das Alphabet zwischen 20.000 bis 100.000 Zeichen groß, und asiatische Zeitungen oder Bücher enthalten üblicherweise über 10.000 verschiedene Zeichen [56]. Im Rahmen der Projektgruppe werden jedoch nur Byte-Alphabete behandelt, dieser Aspekt wurde daher vernachlässigt.
Es werden (in Kapitel 6.7.2) vier verschiedene Techniken eingeführt, deren Kombination schlussendlich den Kern des Algorithmus bildet:

1. u-Chain Bucket-Sort
2. Induziertes Sortieren
3. Erweitertes Induziertes Sortieren
4. Erkennen und Behandeln von Wiederholungen

Da diese Techniken selbst alle noch kein Suffix Array konstruieren, muss anschließend noch eine der Varianten zur Umwandlung aus 4.2.3 folgen. Ohne diesen Teil wäre der gesamte Algorithmus kein SACA im eigentlichen Sinne: Er konstruiert das Inverse Suffix Array ISA. Dieses enthält für jedes Suffix einen

Rang, der die Position darstellt, an die es im Suffix Array sortiert werden soll. Während der Konstruktion des ISA werden in dem Speicherbereich des ISA zusätzlich weitere Komponenten des Algorithmus gespeichert, um Platz zu sparen.

6.7.2 Bestandteile des Algorithmus und Pseudocode

In diesem Kapitel wird der Algorithmus vorgestellt. Dazu wird er, wie von Maniscalco und Puglisi, in vier Abschnitte eingeteilt, die jeweils unterschiedliche Konzepte realisieren und alle dazu beitragen, dass der Algorithmus schneller wird. Obwohl die Abschnitte aufeinander aufbauen, kann schon ein Algorithmus der nur den ersten Abschnitt implementiert das ISA konstruieren. Im Originalpaper von Maniscalco und Puglisi wurden eine Version mit Abschnitt 6.7.2 und eine ohne diesen implementiert und im Anschluss experimentell mit anderen SACAs verglichen. Dabei stellte sich die Version mit erweitertem induzierten Sortieren als überlegen heraus.

u-Chain Bucket-Sort

Das erste Konzept, das in diesem Algorithmus Anwendung findet, ist eine Methode, bei der Suffixe sortiert werden. Dazu werden sogenannte u-Chains erstellt, verfeinert und abgearbeitet. Eine u-Chain ist eine Kette aus Suffixen des Input-Strings T, die den selben Präfix u haben. Die u-Chains sind daher eine Variante von Buckets, wie sie in 7 definiert werden. Wenn $\mathsf{T}[i, i+|u|) = \mathsf{T}[j, j+|u|)$, so sind alle derartigen i und j in der selben u-Chain verlinkt. Ein Beispiel: Der Input-String sei

i	0	1	2	3	4	5	6	7
$\mathsf{T}[i]$	a	b	c	a	c	a	b	\$

Für $|u| = l = 1$ gibt es folgende u-Chains:

u	\$	a	b	c
u-Chain	(7)	(5, 3, 0)	(6, 1)	(4, 2)

Falls eine u-Chain nur einen Eintrag hat, wie hier die \$-Chain, so spricht man von einem *Singleton*. u-Chains werden im ISA gespeichert:

i	0	1	2	3	4	5	6	7
$\mathsf{T}[i]$	a	b	c	a	c	a	b	\$
$\mathsf{ISA}[i]$	$\perp$	$\perp$	$\perp$	0	2	3	1	$\perp$

Sie sind nur von rechts nach links verlinkt und lassen sich auch nur in dieser Richtung durchlaufen. Das Zeichen $\perp$ markiert jeweils das Ende einer u-Chain. Um die Einträge, die u-Chains verlinken, im ISA von denjenigen Einträgen, die Ränge enthalten, zu unterscheiden, werden Ränge als negative Zahlen gespeichert. Wenn ISA[i] < 0 (bzw. das linkeste Bit ==1), dann gehört zu Suffix i der Rang -ISA[i].

Die u-Chains, die bearbeitet werden sollen, werden als Tupel (h, l) aus dem Index des Kopfes (Head) der Chain h und der Länge von u, l, auf einem Stack gespeichert. Zu bearbeitende u-Chains liegen stets lexikographisch sortiert auf dem Stack, dh. diejenige u-Chain mit dem kleinsten u liegt zuoberst.

```
1  formInitialChains()
2  repeat:
3    (h,l) ← chainStack.pop()
4    if ISA[h]  = ⊥:
5      ISA[h]  ← nextRank()
6    else:
7      while h≠⊥:
8        sym ← getSymbol(h+l)
9        updateSubchain(sym, h)
10       h ← ISA[h]
11
12     sortAndPushSubChains()
13 until chainStack = ∅
```

Algorithmus 6.11: u-Chain Bucket-Sort.

In Algorithmus 6.11 (aus [37]) wird der Pseudocode des u-Chain Bucket-Sort mSufSort dargestellt. Zu Beginn werden die initialen u-Chains der Länge 1 gebildet und sortiert auf den Stack gelegt. Danach werden solange u-Chains auf dem Stack abgearbeitet, bis dieser leer ist, und damit alle Suffixe im ISA einen Rang zugewiesen bekommen haben. Es wird in jedem Schleifendurchlauf ein u-Chain-Tupel vom Stack genommen. Handelt es sich dabei um ein Singleton, so wird direkt ein Rang zugewiesen. Wenn die u-Chain mehrere Einträge hat, so wird die u-Chain *verfeinert*. Das bedeutet, es werden Sub-Chains erstellt die jeweils ein Zeichen mehr berücksichtigen. Die Länge von u erhöht sich also um 1. Diese Sub-Chains werden dann, sobald die u-Chain durchlaufen wurde, sortiert und auf den Stack gelegt. Daher werden sie im nächsten Durchlauf als erstes abgearbeitet, und sollten durch die Verfeinerung Singletons entstanden sein, werden diese nun entsprechend Ränge zugewiesen bekommen.

Ein auf direkten Vergleichen basierendes Bucket-Sorting reicht allein für einen konkurrenzfähigen SACA noch nicht aus (vgl. [50]).

Induziertes Sortieren

In diesem Abschnitt wird die Grundidee des Induzierten Sortieren erläutert, wie sie hier verwendet wird. In Abschnitt 5.2.3 wird dieses Konzept noch allgemeiner erklärt. Grob gesagt gibt es viele Suffixe, deren Ränge abhängig von den Rängen anderer Suffixe sind, und diese Abhängigkeit kann zum schnelleren Sortieren genutzt werden. Man sagt, Suffix i aus einer u-Chain der Länge ℓ lässt sich *induziert* sortieren, wenn der Rang für Suffix $i + l$ bereits bekannt ist. Es kann beobachtet werden, dass solche induziert sortierbaren Suffixe lexikographisch vor den anderen Suffixen kommen: sie haben offensichtlich selbst lexikographisch kleinere Sub-Suffixe, als andere Suffixe mit dem selben Präfix u. Bei der Verfeinerung der u-Chains werden nun nur noch nicht-induziert sortierbare Suffixe in Sub-Chains platziert, während die induziert sortierbaren Suffixe anders sortiert werden. Die Reihenfolge dieser induziert sortierbaren Suffixe lässt sich mittels vergleichsbasiertem Sortieren bestimmen, indem der Rang von Suffix $i + l$ als Sortierschlüssel genutzt wird. Sobald die Menge M ($|M| = m$) der induziert sortierbaren Suffixe nach den Sortierschlüsseln sortiert wurde, werden diesen Suffixen die nächsten m Ränge zugewiesen. Diese Art des Sortierens für Suffixe nennen wir (hier) *Induziertes Sortieren.*

Im Pseudocode (Algorithmus 6.12, aus [37]) wird jetzt der Ansatz des Bucket-Sort um Induziertes Sortieren erweitert. Es wird nun direkt nach dem Check auf ein u-Chain-Singleton geprüft, ob sich der Eintrag der u-Chain per Induktion sortieren lässt (Zeile 8). Falls dies der Fall ist, wird das Suffix mit dem Sortierschlüssel annotiert. Am Ende einer u-Chain wird dann nach der Verfeinerung der Chains die Menge der annotierten Suffixe sortiert und gerankt.

Erweitertes Induziertes Sortieren

In diesem Abschnitt wird die Idee des Induzierten Sortierens weitergeführt, um die Laufzeit des mSufSort weiter zu verbessern. Dazu wurde überlegt, dass sich alle Suffixe direkt zu Beginn in zwei Gruppen unterteilen lassen: Typ S (smaller) und Typ L (larger).
Suffix i ist von Typ L, wenn es größer als das nächste Suffix $i + 1$ ist: $\mathrm{T}[i] > \mathrm{T}[i + 1]$, oder im Falle von Gleichheit wird das nächste Zeichen inspiziert und mit $T[i]$ verglichen, bis schließlich $\mathrm{T}[i] > \mathrm{T}[i + j]$. Andernfalls (falls entweder $\mathrm{T}[i] < \mathrm{T}[i + 1]$ oder bei Gleichheit dann $\mathrm{T}[i] < \mathrm{T}[i + j]$) ist es von Typ S. Alle Suffixe i von Typ L kommen vor Suffixen j mit Typ S, die den gleichen Anfangsbuchstaben $\mathrm{T}[i] = \mathrm{T}[j]$ haben: $\mathrm{T}[i, n] < \mathrm{T}[j, n]$.
Das bedeutet für Suffix i, dass sich sein Rang später von dem Rang des Suffix j ableiten lassen wird. Daher wird die gesamte Prozedur des Bucket-Sort und Induzierten Sortierens jetzt nur noch auf Typ S Suffixe angewandt, während Typ L Suffixe mit einer neuen Methode sortiert werden. Und zwar fügen wir immer, wenn Suffix i ein Rang zugewiesen wird, und Suffix $i - 1$ von Typ L ist, Suffix $i - l$ am Ende einer Liste $M_{\alpha\beta}$ ein, ganz konkret der Liste $M_{T[i-1]T[i]}$ (aufgrund von Typ L-Eigenschaft gilt für alle $M_{\alpha\beta}$: $\alpha \leq \beta$). Alle Listen $M_{\alpha\beta}$

```
1  formInitialChains()
2  repeat:
3    (h, l) ← chainStack.pop()
4    if ISA[h]  = ⊥:
5      ISA[h]  ← nextRank()
6    else:
7      while h ≠⊥:
8        if isRanked(h + l)
9          noteSuffix(h, ISA[h + l] )
10       else:
11       sym ← getSymbol(h + l)
12       updateSubchain(sym, h)
13
14       h ← ISA[h]
15
16     pushSubChains()
17     rankNotedSuffixes()
18
19   until chainStack = ∅
```

Algorithmus 6.12: Induziertes Sortieren.

zu einem gegebenen α und beliebigen β werden genau dann gerankt, wenn auf dem Stack die u-Chain mit Tupel (h, l) erreicht wird, deren $T[h] = \alpha$.

Im Pseudocode (Algorithmus 6.13) wird nicht näher auf dieses Einfügen in die Listen $M_{\alpha\beta}$, das beim Ranken passiert, eingegangen, um die Übersichtlichkeit zu bewahren. Ganz zu Anfang werden die Suffixe jetzt allerdings in Typ S und Typ L unterschieden, wobei von Maniscalco und Puglisi vorgeschlagen wird, Typ L-Suffixe durch das Zeichen $\perp$ im ISA zu kennzeichnen. Beim Bilden der u-Chains werden dann nur die Typ S-Suffixe verwendet. Es muss eine Datenstruktur, die die Heads und Tails der Listen $M_{\alpha\beta}$ verwaltet, wir nennen sie hier VLists, initialisiert werden. Es wäre praktisch, wenn diese Datenstruktur die Listen bereits in lexikographischer Sortierung speichert. Auch wäre praktisch, wenn zudem Buch darüber geführt wird, ob Elemente in einer Liste enthalten sind, oder ob diese leer ist. Der Zugriff auf alle Listen $M_{\alpha\beta}$ zu einem gegebenen α und beliebigen β wird hier einfach VLists[α] genannt. Maniscalco und Puglisi schreiben, der Inhalt der Listen $M_{\alpha\beta}$ lässt sich wieder im ISA speichern. Die Datenstruktur VLists hat daher eine Größe die in $O(|\Sigma|^2)$ liegt, bzw. genauer bei maximal $\frac{|\Sigma|*(|\Sigma|-1)}{2}$. Für große Alphabete, so argumentieren die Autoren, sei die Idee erweiterbar, indem für jedes Symbol nur noch zwei Listen verwaltet werden; M_α für Suffixe mit Präfix $\alpha\beta$ wobei $\alpha \neq \beta$, und eine Liste mit $M_{\alpha\alpha}$.
Für Datensätze, bei denen Typ S Suffixe häufiger sind, kann man den Ansatz optimieren, indem man dann die Rollen der Typen umkehrt. Dafür muss man

```
1  classifyTypeUV()
2  formInitialChains(Type U suffixes)
3  initializeVLists()
4  repeat:
5    (h, l) ← chainStack.pop()
6    for all M_{T[h]β} ∈ VLists[T[h] ] ≠⊥} in lexicographical order :
7      rankAll(M_{T[h]β})
8
9    if ISA[h]  = ⊥ :
10     ISA[h]  ← nextRank()
11   else:
12     while h ≠⊥ :
13       if isRanked(h + l) :
14         noteSuffix(h, ISA[h + l] )
15       else:
16       sym ← getSymbol(h + l)
17       updateSubchain(sym, h)
18
19       h ← ISA[h]
20
21     pushSubChains()
22     rankNotedSuffixes()
23
24   if chainStack = ∅ :
25     activeVLists ← VLists ≠⊥
26     rankAll(activeVLists)
27     if activeVLists = ∅ :
28       return
29
```

Algorithmus 6.13: Erweitertes Induziertes Sortieren.

vor allem beachten, dass die Ränge dann auch anders herum verteilt werden müssen (absteigend statt aufsteigend).

Erkennen und Behandeln von Wiederholungen

In diesem Abschnitt wird die letzte und wohl komplizierteste Erweiterung des mSufSort erklärt: Der Umgang mit Wiederholungen im Input-Array. Dazu muss man zunächst wissen, dass es für vergleichsbasiertes Sortieren, wie es beim Bucket-Sort verwendet wird, sehr ungünstig ist, wenn sich Sequenzen wiederholen, wie zB.: „abcabcabc“. Für eine *abc*-Chain hätte man hier drei aufeinanderfolgende Einträge, jedoch beim Verfeinern immer noch zwei mal die Sequenz *abca*. Je länger diese Wiederholungssequenz, desto ungünstiger (da immer öfter weitere Verfeinerung zu Sub-Chains nötig wird). Auf der anderen Seite hat man mit den *u*-Chains schon ein mächtiges Tool, mit dem sich eben solche Sequenzen erkennen lassen, und kann dann die Wiederholungen aus den Verfeinerungen ausschließen und anders sortieren.
Wir definieren eine Wiederholungssequenz $S_{i,u}$ als Sequenz, die bei Index i (von rechts) beginnt, und jeweils den String u der Länge l direkt hintereinander enthält: $\mathrm{T}[i, i+l] = \mathrm{T}[i+l+1, i+2l] = ...$ Am letzten Beispiel: der String

i	0	1	2	3	4	5	6	7	8	9
T[i]	a	b	c	a	b	c	a	b	c	\$

entspricht einer Wiederholungssequenz $S_{7,abc} = abc^3$. Folgendes ist dagegen keine Wiederholungssequenz:

i	0	1	2	3	4	5	6	7	8	9
T[i]	a	b	d	a	b	c	a	b	a	\$

Hier kommen zwar die Buchstabenkombinationen „ab“ oft vor, aber es sind Lücken zwischen den *u*-Strings, mit unterschiedlichen Zeichen. Man unterscheidet bei einer Wiederholungssequenz zwischen der sogenannten *terminierenden* Position und den nicht-terminierenden Positionen. Die terminierende Position entspricht genau dem i zu $S_{i,u}$, also dem rechtesten Beginn des wiederholt auftretenden Substrings u. Alle weiteren Positionen j, wobei $j = i - r \cdot |u|$, die den Beginn weiterer Substrings u, die weiter links liegen, anzeigen, sind nichtterminierend.
Diese Unterscheidung wird getroffen, weil sich alle nicht-terminierenden Positionen direkt anhand der Ränge der terminierenden Positionen bestimmen lassen. Es gilt für Wiederholungssequenzen $S_{i,u}$ (mit $|u| = l$), deren terminierende Position, Suffix i, sich induktiv sortieren lässt, dass $\mathsf{T}[i,n] < \mathsf{T}[i-l,n] < ... < \mathsf{T}[i-l(|S|-1),n]$. Die Ränge der nicht-terminierenden Positionen werden also aufsteigend, beginnend bei der terminierenden Position, von rechts nach links vergeben. Im umgekehrten Fall, wenn Suffix i nicht induziert sortierbar ist, werden die Ränge genau anders herum vergeben: $\mathsf{T}[i,n] > \mathsf{T}[i-l,n] > ... > \mathsf{T}[i-l(|S|-1)]$. Wenn sich ein Suffix i induziert sortieren lässt, so bedeutet das, dass der Buchstabe hinter dem Präfix u kleiner ist, als Zeichen $\mathsf{T}[i]$, oder zumindest höchstens gleich groß, und das Suffix $i+l$ kleiner ist. Ansonsten wäre

dieses Suffix nicht vorher gerankt worden. Daher kommt im Induktionsfall das Suffix an terminierender Position zuerst. Lässt sich das Suffix i dagegen nicht induziert sortieren, dann muss das Zeichen (bzw. mindestens aber das Suffix) $i+l$ größer sein, da es nicht vorher abgearbeitet wurde. In diesem Fall ist das Suffix an terminierender Position also das mit dem höchsten Rang. Zwei Beispiele, um die Regeln zu verdeutlichen: Sei y der zuletzt vergebene Rang zum Zeitpunkt der Rangzuweisung an die Suffixe der Wiederholungssequenz. Zunächst der Fall, in dem sich per Induktion der Rang von Suffix i bestimmen lässt, wobei $\mathsf{T}[i] = \mathsf{T}[i+l]$ und erst $\mathsf{T}[i+1] > \mathsf{T}[i+l+1]$.

Rang	Suffix											
$y+1$	$\mathsf{T}[i,n]$	a	b	c	a	$						
$y+2$	$\mathsf{T}[i-l,n]$	a	b	c	a	b	c	a	$			
$y+3$	$\mathsf{T}[i-2l,n]$	a	b	c	a	b	c	a	b	c	a	$

Nun der andere Fall.

Rang	Suffix											
$y+3$	$\mathsf{T}[i,n]$	a	b	c	d	$						
$y+2$	$\mathsf{T}[i-l,n]$	a	b	c	a	b	c	d	$			
$y+1$	$\mathsf{T}[i-2l,n]$	a	b	c	a	b	c	a	b	c	d	$

In einer u-Chain können leider mehr als eine Wiederholungssequenz vorkommen, so sind zum Beispiel die Sequenzen $S_{6,ab}$ und $S_{0,ab}$ in dieser ab-Chain enthalten:

i	0	1	2	3	4	5	6	7	8	9
$\mathsf{T}[i]$	a	b	a	b	c	a	b	a	b	$

Schon dieser Fall ist komplexer. Die Ränge für $S_{6,ab}$ können direkt zugewiesen werden, da $ vorher gerankt wurde. Dadurch, dass dann nur noch eine terminierende Position, Suffix 2, in der Sub-Chain enthalten ist, wird schließlich diese Wiederholungssequenz gerankt, beginnend bei Suffix 0. Offen ist noch was geschieht in dem Fall, dass zwei Wiederholungssequenzen sich induziert sortieren lassen: Dann werden die Ränge der nicht-terminierenden Positionen *interleaved* (englisch für „in einander verzahnt“). Das bedeutet, dass die Ränge der nicht-terminierten Positionen abwechselnd für alle induziert sortierbaren Wiederholungssequenzen vergeben werden, und zwar entsprechend der Reihenfolge der Ränge der terminierten Positionen. Am Beispiel zweier Wiederholungssequenzen $S_{i,u}$ und $S_{j,u}$: Sei Suffix $i <$ Suffix j. Dann folgt: $rank(i-l) < rank(j-l) < rank(i-2l) < rank(j-2l) < ...$ und so weiter, bis zur letzten nicht-terminierten Position aus Suffix j.
Es bleibt der Fall, wenn mehrere Wiederholungssequenzen der selben u-Chain sich nicht per Induktion sortieren lassen. Dies ist der komplizierteste Fall. Hier wird nämlich zunächst eine Liste aus allen nicht-terminierenden Positionen der Wiederholungssequenzen, C, auf den Chainstack gelegt. Die terminierenden Positionen werden in Sub-Chains eingeordnet. Solange sich allerdings eine Liste C auf dem Stack befindet, werden keine Ränge zugewiesen, sondern es wird eine Liste Q gefüllt, indem jedes Suffix, das sonst gerankt werden würde, ans Ende

der Liste gehängt wird. Sobald C dann oben auf dem Stack liegt, wird die Reihenfolge der terminierenden Positionen aus Q genutzt, um die Suffixe aus C zu sortieren. Dabei wird ebenfalls interleaved, wie bei den induziert sortierbaren Wiederholungssequenzen - entsprechend anders herum. Maniscalco und Puglisi schreiben, um C und Q zu implementieren, werde nur konstanter zusätzlicher Speicherplatz benötigt, um Beginn und Ende der Listen zu speichern, während die eigentlichen Verlinkungen wieder im ISA gespeichert sind. Der Pseudocode (Algorithmus 6.14) dient nur der besseren Einordnung in den gesamten Algorithmus, da die Wiederholungserkennung (und -behandlung) an verschiedene Stellen verteilt wird.

6.7.3 Weitere Details

In diesem Kapitel finden sich alle wichtigen Details, die sich keinem speziellen Konzept zuordnen ließen, wie zum Beispiel dem grundlegenden Sortierverfahren, das an einigen Stellen (insbesondere Bucket-Sort) seine Anwendung findet. Aber auch ein paar erweiternde Ideen, zB. zur Steigerung der Effizienz des Bucket-Sort, werden hier vorgestellt. Es ist jedoch keine vollständige Wiedergabe des Kapitels 5 über „Entwicklungs- und Implementierungsdetails“ aus [37].

Sortierverfahren

Für vergleichsbasiertes Sortieren wird Introsort (siehe 4.1.2) verwendet. Zudem wird zur Optimierung Insertion-Sort für besonders kleine Partitionen genutzt, wo es schneller als Quicksort ist.

Eingabe-Transformation

Es macht Sinn, das Alphabet wenn möglich kompakter darzustellen, wie auch in 4.2.1 erläutert. Dazu wird es umcodiert auf fortlaufende Zahlenwerte $0..|\Sigma|-1$. Das \$-Zeichen wird nicht explizit ins Alphabet aufgenommen, da dem n-ten Suffix sein Rang direkt zu Beginn zugewiesen werden kann. Es wird nie zum Vergleich benötigt, da alle Suffixe, die es im Zuge von Bucket-Sort betrachten wollten bereits per Induktion sortiert werden können mit dem Rang als Sortierschlüssel.

```
classifyTypeUV()
formInitialChains(Type U-suffixes)
initializeVLists()
initializeQ()
repeat:
  if typeOf(chainStack.peak()) ≠ u-Chain :
    C ← chainStack.pop()
    cOnStack ← FALSE
    rankBy(C, Q)
    rank(Q)
  (h, l) ← chainStack.pop()
  for all M_αβ ∈ VLists[T[h] ] ≠⊥} :
    rankAll(M_αβ)
  if ISA[h]  = ⊥ & not cOnStack :
    ISA[h]  ← nextRank()
  else if ISA[h]  = ⊥ :
    Q ← h
  else :
    while h ≠⊥ :
      if isInRepSeq(h):
        if toBeInterleaved:
          noteNTSuffix(h)
        else:
          updateC(h)
        if isEndOfRepSeq(h):
          toBeInterleaved ← FALSE
      else :
        if isRanked(h + l) :
          noteSuffix(h, ISA[h + l] )
          if isHeadOfRepSeq(h):
            toBeInterleaved ← TRUE
        else :
          sym ← getSymbol(h + l)
          updateSubchain(sym, h)
          if isHeadOfRepSeq(h):
            C ← createC(h, l)
      h ← ISA[h]
          if(C≠ ∅) :
            pushC()
            cOnStack ← TRUE
      pushSubChains()
      rankNotedSuffixes()
      rankNotedNTSuffixes()
    if chainStack = ∅ :
      activeVLists ← VLists ≠⊥
      rankAll(activeVLists)
        if activeVLists = ∅:
          return
```

Algorithmus 6.14: Wiederholungserkennung und -behandlung.

Erweiterung für Bucket-Sort

Das Bucket-Sort lässt sich verbessern, indem statt nur einem Symbol jeweils zwei Symbole (sogenannte *Bigramme*) auf einmal betrachtet werden (siehe auch Abschnitt 4.2.1). Für kleine Alphabete mit $|\Sigma| \leq 256$ ist $|\Sigma|$ noch klein genug. Nicht nur werde durch das Kombinieren zweier Zeichen String-Sortieren beschleunigt, sondern es sei so auch möglich mehr Suffixe per Induktion zu sortieren (siehe nächsten Abschnitt).

6.7.4 Erweiterung für (einfaches) Induziertes Sortieren

Während dem Induzierten Sortieren werden Index des per Induktion zu sortierenden Suffix und der zugehörige Sortierschlüssel laut Pseudocode annotiert. Dazu werden Index und Schlüssel direkt nebeneinander in einem eigenen, dynamischen Array gespeichert, so Maniscalco und Puglisi. Dadurch werden unnötige Cache-Misses vermieden, was den Algorithmus beschleunigt. Allerdings wird dafür Speicherplatz benötigt, und zwar $8m$ Byte für m Suffixe die gerade per Induktion sortiert werden sollen. In der Praxis sollte m deutlich kleiner als n sein, insbesondere nachdem das Alphabet kompakter gemacht wurde. Für den Fall, dass diese $8m$ Byte problematisch werden könnten, schlagen Maniscalco und Puglisi zwei Alternativen vor; im einen Fall wird nur der Index des per Induktion zu sortierenden Suffixes gespeichert (mit $4m$ Byte Speicherplatz). Der andere Fall ist etwas komplexer, benötigt aber nur $2n+\mathrm{o}(n)$ Bits Speicher. Diese Methode wurde nicht tatsächlich in die Implementierung aufgenommen.
Durch das Betrachten von zwei Zeichen auf einmal kann nun während eine Chain mit Länge des gemeinsamen Präfix ℓ bearbeitet wird in drei Typen von Suffixen unterschieden werden: Suffix i ist von Typ A, wenn der Rang von Suffix $i+l-1$ bekannt ist. Es ist von Typ B, wenn der Rang von Suffix $i+l$ bekannt ist – und von Typ C ansonsten. Lexikographisch sind nun Typ A Suffixe kleiner als Typ B Suffixe, und diese wiederum kleiner als Typ C Suffixe. Es kann nun zuerst die Reihenfolge der Typ A Suffixe mittels Induziertem Sortieren bestimmt werden. Danach bestimmt man auf die selbe Art die Reihenfolge der Typ B Suffixe. Tatsächlich lässt sich dies in einem einzigen Aufruf kombinieren, indem als Sortierschlüssel für Typ B Suffixe nicht $i+l$ sondern $-(i+l)$ verwendet wird.

6.7.5 Eigene Implementierung und Ausblick

In unserer Implementierung des mSufSort haben wir u-Chain Bucket-Sort, Induziertes Sortieren und Erweitertes Induziertes Sortieren umgesetzt. Viele kleinere Ungenauigkeiten in der Beschreibung mussten dafür zunächst geklärt werden, zum Beispiel die verwendeten Datenstrukturen betreffend. Die Erkennung und Behandlung von Wiederholungen ist teilweise implementiert worden, für den einfacheren Fall in dem sich Wiederholungssequenzen induziert sortieren lassen. Die größte Herausforderung bei der Implementierung ist wohl der Umgang mit den einfach verlinkten Listen, die unter Anderem (aber nicht nur) für u-Chains verwendet werden. Diese selbstgebaute Datenstruktur ist sehr fehleranfällig, da

komplexe Operationen benötigt werden, zB. Einfügen noch während über die Liste iteriert wird.

Sämtliche hier beschriebene Optimierungen, bis auf die Verwendung eines effektiven Alphabets, wurden noch nicht implementiert, dafür wurde allerdings bei der initialen Befüllung des Chainstacks ein Trick angewandt. Statt diese nämlich über eine u-Chain Verfeinerung mittels Introsort (auf einer Eingabelänge von $O(n)$) zu lösen, wurde ein Array der Größe $O(|\Sigma|)$ eingeführt. Dieses enthält immer das rechteste Element einer u-Chain zum Zeitpunkt der Bearbeitung, dh. nach einem kompletten Scan von links nach rechts enthält es in sortierter Reihenfolge alle Elemente, die so direkt auf den Chainstack geschrieben werden können. Dadurch konnte der Speicherverbrauch im Vergleich zur Referenzimplementierung noch deutlich gesenkt werden. Erst für Alphabete, deren Größe die Anzahl der Typ S-Suffixe übersteigt, lohnt diese Optimierung nicht.
Zudem existiert die Variante mSufSort_scan, in der die Verfeinerung der u-Chains nur mit diesem Verfahren ausgeführt werden.

Eine weitere Optimierung könnte, solange keine Wiederholungserkennung benötigt wird, noch darin bestehen, die Sortieroperation für u-Chains zu vereinfachen, indem diese nicht mehr konsequent geordnet verlinkt werden, sondern auch chaotisch gelinkt werden darf. Dies würde eine Wiederholungserkennung, wie im Originalpaper beschrieben, unmöglich machen, aber sicher einige Laufzeit einsparen.

Auch wäre noch denkbar, ein Sortierverfahren zu verwenden, das direkt auf den einfach verlinkten Listen funktioniert, statt jeweils die Liste zu durchlaufen und damit einen Vektor zu befüllen, bevor darauf Introsort angewandt werden kann.

Zur Zeit verwenden wir den naiven Inplace Ansatz zur Umwandlung des ISA zu SA- es ist daher noch mit weiterer Verbesserung der Laufzeit zu rechnen, wenn andere Varianten eingebaut werden, auch wenn diese auf Kosten des Speicherverbrauchs gehen wird.

6.7.6 Naive Parallelisierung

Grundsätzlich ist es möglich, in mSufSort statt Introsort einen parallelen Sortieralgorithmus zu verwenden. Bei Versuchen wurde dazu IPs^{4}o [4] angewandt. An anderer Stelle, wo ein stabiler Sortierer verwendet wurde um unnötige Rechenzeit durch doppelte Vergleiche zu vermeiden, kann ebenfalls ein paralleles Gegenstück verwendet werden. Bei der Auswertung zeigt sich, dass IPs^{4}o sich nicht für den Einsatz im mSufSort eignet, da die Laufzeit der parallelen Varianten die der sequentiellen übersteigen.

6.7.7 Fazit

Dieser Ansatz zur Sortierung von Suffixen legt den Fokus nicht auf die Erzeugung des Suffix Arrays, sondern es wird statt dessen das Inverse Suffix Array gebildet, aus dem sich direkt effizient die Burrows-Wheeler-Transformation (eine Anwendung für SACAs) ableiten lässt. Dadurch unterscheidet sich dieser SACA ganz grundlegend von den meisten anderen Algorithmen, die direkt Suffix-Indizes verschieben, statt Ränge zu berechnen.
Insgesamt kann man dabei eigentlich dennoch nicht von einem Ansatz sprechen, dieser Algorithmus vereint vier verschiedene Konzepte in einem, man könnte sogar sagen fünf (mit dem Zusammenfassen zu Zeichenpaaren, Kapitel 6.7.3). Dadurch wird er leider etwas unübersichtlich. Auch die Mehrfachnutzung des ISA als Platz für diverse andere Datenstrukturen (verlinkte Listen) ist für eine bessere Verständlichkeit – sowie einfache Implementierung – nicht hilfreich. Aufgrund zahlreicher solcher Tricks wird, falls 32bit Typen verwendet werden, die Eingabegröße beschränkt auf $n < 2^{31}$ (durch negative Ränge). Bei Anwendung des empfohlenen, da schnellsten, Verfahrens zur Erzeugung des Suffix Arrays, wird die Eingabegröße sogar auf $n < 2^{30}$, unter Umständen sogar auf $n < 2^{29}$ beschränkt. Das sind nur noch $\frac{1}{8}$ der ursprünglichen (und üblichen) 2^{32} Zeichen.
Auf der anderen Seite kann mSufSort mit anderen SACAs, wie zB. DeepShallow [38], was Laufzeit und Speicherverbrauch angeht, mithalten, wie Experimente von Maniscalco und Puglisi zeigen konnten. Eine formale Laufzeitanalyse liegt zwar nicht vor, aber die Autoren schreiben, sie vermuten eine Schranke von $\Theta(n^2 log(n))$. Sie geben einen Speicherverbrauch von „etwas mehr als“ $4n + zn$ Byte an. Der tatsächliche Speicherverbrauch in den Experimenten liegt bei etwa 5-6n Byte, wobei beachtet werden muss, dass dies nur der Fall ist, wenn bei der Konstruktion des Suffix-Array der Eingabe-String überschrieben werden darf. Andernfalls muss mit Abstrichen in der tatsächlichen Laufzeit oder einem deutlichen Wachstum des benötigten Speicherplatzes gerechnet werden. Allgemein liegt bei dem Algorithmus eine Trade-Off Situation zwischen Speicherplatz und Laufzeit vor. Mit mehr Speicherplatz ließen sich unter anderem einige Konzepte deutlich Cache-freundlicher umsetzen. Der Algorithmus mSufSort scheint außerdem recht robust gegenüber dem Auftreten von Wiederholungen in der Eingabe zu sein. Ein weiterer Vorteil kann darin gesehen werden, dass es möglich ist, den Algorithmus (mit einigen Anpassungen) auch für große Alphabete mit $|\Sigma| > 256$ zu nutzen.
Der ursprüngliche Algorithmus, wie er hier beschrieben wurde, wurde inzwischen erweitert, es existiert nun auch eine parallelisierte Version von Maniscalco (unter `https://github.com/michaelmaniscalco/msufsort`).

6.7.8 Beispiel

Hier ein Beispiel, das zum Verständnis des Ablaufs im Algorithmus beitragen sollte.

Initialisierung

Typen L und S (bzw. V und U entsprechend) bestimmen, und u-Chains mit $|u| = 1$ aus Typ U (S) bilden:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T[i]	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
ISA[i]	V	$\perp$	1	V	2	V	V	4	7	V	8	V	V	V	$\perp$

Heads der u-Chains als Tupel (h,l) (mit h Index des rechtesten Elements der Chain, $l = |u|$) sortiert auf den Chain-Stack legen (Sortieren immer mit **Introsort**, für kleine Partitionen Insertionsort):
chainStack:

- (14, 1) - $
- (10, 1) - a

Hauptschleife

Nehme Tupel vom Stack: (14, 1).
Eintrag bei Index 14 im ISA ist $\perp$, also weise ersten Rang zu:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T[i]	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
ISA[i]	V	$\perp$	1	V	2	V	V	4	7	V	8	V	V	V	-0

Prüfe, ob Eintrag ISA[$h-1$] V ist: Ja.
Speichere Index 13 in $M_{a\$}$ (im ISA - nur Head/Tail der Liste extra)
Bearbeite alle (nicht-leeren) Listen M_{a*}: Weise Suffix 13 den nächsten Rang zu.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T[i]	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
ISA[i]	V	$\perp$	1	V	2	V	V	4	7	V	8	V	V	-1	-0

Prüfe, ob ISA am Index davor V ist: Ja. Speichere 12 in M_{aa}.
Bearbeite alle (nicht-leeren) Listen M_{a*}: Weise Suffix 12 den nächsten Rang zu.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T[i]	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
ISA[i]	V	$\perp$	1	V	2	V	V	4	7	V	8	V	-2	-1	-0

Prüfe, ob ISA am Index davor V ist: Ja. Speichere 11 in M_{ca}.
(Bearbeite nicht-leere Listen M_{a*}, es gibt keine.)
Hole nächstes Tupel vom Chain-Stack: (10,1) und bearbeite a-Chain:

u-Chain Verfeinerung Bei Index 10: Bilde Sub-Chain mit „ac“ (weil Eintrag kein Singleton, und Index $h + l = 11$ noch nicht gerankt).
Folge Chain: $h = 8$. Bilde Sub-Chain mit „ab“ und erkenne Beginn einer Wiederholungssequenz.
Folge Chain: $h = 7$. Speichere 7 in C (merke dabei Kopf der Sequenz, 8).
Folge Chain: $h = 4$. Bilde Sub-Chain „ac“.
Folge Chain: $h = 2$. Bilde Sub-Chain „ab“ und erkenne Beginn einer Wiederholungssequenz.
Folge Chain: $h = 1$. Speichere 1 in C (mit Verweis auf 2).
Ende der Chain erreicht, C nicht-leer: Lege C auf den Stack.
Sortiere und pushe Sub-Chains auf den
chainStack:

- (8, 2) - 'ab'
- (10, 2) - 'ac'

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T[i]	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
ISA[i]	V	$\perp$	$\perp$	V	$\perp$	V	V	$\perp$	2	V	4	V	-2	-1	-0

Bearbeite das nächste Tupel auf dem ChainStack: (8, 2).
Bilde Sub-Chain 'aba': (8, 3).
Folge Chain: $h = 2$. Bilde Sub-Chain „aba“ (verlinke).
Ende der Chain erreicht: Sortiere und pushe Sub-Chains.
Fahre fort mit Sub-Chain-Verfeinerung bis $l = 5$ und
chainStack:

- (8, 5) - „abaca“
- (2, 5) - „abacc“
- (10, 2) - „ac“

Weil C auf dem Stack liegt, füge 8 (Singleton) ans Ende von Liste Q (speichere wieder in ISA Verlinkungen, nur Head und Tail extra).
Füge dann 2 ans Ende von Q.
Bearbeite „ac“-Chain (10, 2).
Füge 10 in Q ein, da per Induktion sortiert (ISA[10+2]=-2 $<$ 0)
Folge weiter Chain: $h = 4$. Bilde Sub-Chain „acc“ (4, 3).
Füge 4 in Q ein (Singleton).
Q: 8, 2, 10, 4
C liegt oben auf dem Stack: Ranke C und Q miteinander.
8 ist Schlüssel für 7, 2 Schlüssel für 1.
Vergib Ränge *interleaved*:
7 Rang 3 - speichere 6 in M_{ca}
1 Rang 4 - speichere 0 in M_{ca}
8 Rang 5

2 Rang 6
10 Rang 7 - speichere 9 in M_{ba}
4 Rang 8 - speichere 3 in M_{ba}
$M_{ba} = 9, 3$ und $M_{ca} = 11, 6, 0$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T[i]	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
ISA[i]	V	-4	-6	V	-8	V	V	-3	-5	V	-7	V	-2	-1	-0

Ende: Abarbeitung übriger Listen M

Chain-Stack ist leer, arbeite übrige nicht-leere Listen M alphabetisch ab, vergib Ränge von Kopf der Liste bis Ende.
Ranke Liste M_{ba}:
9 Rang 9
3 Rang 10
Ranke Liste M_{ca}:
11 Rang 11
6 Rang 12 - speichere 5 in M_{cc}
0 Rang 13
Ranke Liste M_{cc}:
5 Rang 14
Fertiges ISA:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T[i]	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
ISA[i]	-13	-4	-6	-10	-8	-14	-12	-3	-5	-9	-7	-11	-2	-1	-0
SA[i]	14	13	12	7	1	8	2	10	4	9	3	11	6	0	5

Umwandlung zum SA

Hier durchgeführt: Zyklisches Vertauschen, bis Zyklus-Ende (nächster Eintrag im ISA bereits positiv), dann von links nach rechts nächsten Zyklus suchen usw. bis alle Einträge positiv. Ergebnis siehe oben.

6.8 nzSufSort

In diesem Kapitel wird der *nzSufSort* vorgestellt. Dieser wurde in [43] vorgestellt und kombiniert den Difference Cover Ansatz, welcher in Abschnitt 6.4.2 vorgestellt wurde, mit der L/S-Typisierung. Durch eine speichereffiziente Implementierung benötigt der Algorithmus bis auf den Speicherbereich für den Eingabetext und den Speicherbereich für das Suffix-Array nur konstanten zusätzlichen Speicher, wenn das Alphabet des Eingabetextes eine konstante Größe besitzt. Außerdem wird eine Laufzeit von $\mathcal{O}(n)$ erreicht. Dieser Algorithmus besitzt also eine theoretisch optimale Laufzeitschranke und ist möglichst leichtgewichtig.

Da in der referenzierten Arbeit viele Schritte des Algorithmus unklar beschrieben, oder teilweise ausgelassen wurden, war es eine Herausforderung diese Schritte zu implementieren. Daher wird in diesem Kapitel der Algorithmus vollständig und verständlich beschrieben und jeder Schritt an einem Beispiel erläutert.

6.8.1 Einleitung

Grundlagen

Um das Suffix-Array zu konstruieren wird zunächst das Suffix-Array der S-Typ-Positionen, welches wir mit SA_S bezeichnen, mit dem Difference Cover Ansatz bestimmt. Aus SA_S lässt sich dann durch einen Links-Induktions-Scan das vollständige Suffix-Array SA konstruieren. Um SA_S zu konstruieren wird ähnlich zum DC3 der Eingabetext in Triplets aufgeteilt. Im Unterschied zum DC3 werden aber nicht alle Positionen des Eingabetextes berücksichtigt, sondern nur die zu sortierenden S-Typ-Positionen.

Wir bezeichnen einen Teilstring $\mathsf{T}[i, j)$ als *S-String*, wenn i und $j-1$ S-Typ-Positionen sind. Um die Ordnung zwischen zwei S-Strings T_1 und T_2 festzulegen, wird die übliche lexikographische Ordnung verwendet. Falls aber T_1 und T_2 unterschiedliche Längen besitzen und ein String ein Präfix des anderen ist, erhält der kürzere String eine höhere Ordnung. Eine Konkatenation von k S-Strings bezeichnen wir als Z_k-String. Dann entsprechen die Triplets im DC3 hier den Z_3-Strings.

Auf diese Weise lässt sich T_{12} ähnlich definieren wie im DC3: T_{12} enthält die lexikographischen Ränge aller i-ten Z_3-Strings, wobei T_{12} in die Mengen mit i modulo $3 = 1$ und i modulo $3 = 2$ aufgeteilt ist und diese konkateniert werden. Analog enthält T_0 die lexikographischen Ränge aller i-ten Z_3-Strings mit i modulo $3 = 0$. Die Länge von T_0 bezeichnen wir mit n_0, bzw. die Länge von T_{12} mit n_{12}.

Die Positions-Arrays P_0 und P_{12} enthalten die Indizes aller S-Typ-Positionen, wobei P_0 die i-ten S-Typ-Positionen mit i modulo $3 = 0$ und P_{12} die i-ten S-Typ-Positionen mit i modulo $3 \neq 0$ enthält, wobei hier wie bei T_{12} in die Mengen mit i modulo $3 = 1$ und i modulo $3 = 2$ aufgeteilt wird und diese Mengen konkateniert werden.

Überblick über den Algorithmus

Der Algorithmus lässt sich in zwei Phasen aufteilen. In der ersten Phase wird T_{12} berechnet und das Suffix-Array SA_{12} von T_{12} mit der speichereffizienten Variante DC3-Lite des DC3 berechnet. In der zweiten Phase wird T_0 berechnet und mithilfe von T_0 und ISA_{12} das Suffix-Array SA_0 von T_0 analog zum DC3 induziert. Anschließend werden SA_0 und SA_{12} vereinigt und damit SA_S berechnet. Im letzten Schritt wird durch einen Links-Induktions-Scan das Suffix-Array SA konstruiert.

Methoden und Techniken

Bevor wir den Algorithmus im Detail beschreiben, werden zunächst einige Methoden und Techniken, die häufiger im Algorithmus verwendet werden, vorgestellt.

Zunächst beschreiben wir eine Funktion `retrieve_s_string`, die für eine gegebene S-Typ-Position i den S-String, der bei i beginnt, ausgibt. Dazu müssen wir die nächste S-Typ-Position j finden. Diese kann entweder die direkt auf i folgende Position sein, wenn $\mathsf{T}[i] = \mathsf{T}[j]$ gilt. Ansonsten lassen sich die Typen der auf i folgenden Positionen mit dem Ausdruck L*S*L beschreiben. Wir suchen zunächst die erste L-Typ-Position k nach der gesuchten S-Typ-Position j. Für die Position k muss die Bedingung $\mathsf{T}[k-1] < \mathsf{T}[k]$ gelten, da $\mathsf{T}[k-1]$ eine S-Typ-Position und $\mathsf{T}[k]$ eine L-Typ-Position ist. Anschließend suchen wir die Position j mit $i < j < k$, indem wir von $k-1$ ausgehend in Richtung i die erste Position j suchen, für die $\mathsf{T}[j-1] > \mathsf{T}[j]$ gilt. Dies ist die gesuchte S-Typ-Position j, da sich vor k nur ein Block von S-Typ-Positionen gefolgt von einem Block von L-Typ-Positionen befindet. Da $\mathsf{T}[j-1] > \mathsf{T}[j]$ gilt, ist $j-1$ eine L-Typ-Position und j damit die erste S-Typ-Position nach i.

Nun stellen wir eine Technik vor, um die lexikographischen Ränge der S-Typ-Positionen und zusätzliche benötigte Speicherbereiche effizient abzuspeichern. Diese werden wir im Algorithmus für das Bestimmen der lexikographischen Ränge und das Vereinigen der Suffix-Arrays SA_0 und SA_{12} verwenden. Wir teilen dazu den Speicherbereich von SA analog zum Eingabetext in S-Typ-Positionen und L-Typ-Positionen ein. Für einen Index i gilt, dass $\mathsf{SA}[i]$ eine S-Typ-Position ist, wenn $\mathsf{T}[i]$ eine S-Typ-Position ist. Ansonsten ist $\mathsf{SA}[i]$ eine L-Typ-Position. Wir bezeichnen $\mathsf{SA}[i]$ und $\mathsf{T}[i]$ als *Geschwister*. Um den lexikographischen Rang einer S-Typ-Position i abzuspeichern, reicht es, diese im Geschwister von $\mathsf{T}[i]$ abzuspeichern. Die zusätzlichen Speicherbereiche können nun in den Geschwistern der L-Typ-Positionen $\mathsf{T}[i]$ abgespeichert werden. Da die L- und S-Typen von T eindeutig definiert sind, ist die Position, an der ein Element in SA gespeichert wird, eindeutig bestimmt. Um aber auf ein einzelnes Element zugreifen zu können, muss im schlimmsten Fall das ganze SA von rechts durchlaufen werden, um die L- und S-Typen zu bestimmen.

6.8.2 Algorithmus

In diesem Abschnitt werden wir den *nzSufSort* beschreiben. Dazu werden die einzelnen Schritte zunächst beschrieben und anschließend jeweils ein Beispiel mit dem Eingabetext `caabaccaabacaa$$$` gegeben.

Vorberechnung

Für den Algorithmus muss gelten, dass es höchstens so viele S-Typ-Positionen wie L-Typ-Positionen geben darf. Also muss die Anzahl der S-Typ-Positionen durch $\lfloor \frac{n}{2} \rfloor$ beschränkt sein. Dies wird benötigt, damit der Algorithmus die Speicherschranke einhält. Falls diese Bedingung nicht gilt, lässt sich der Eingabetext durch eine Vorberechnung in diese Form bringen.

Dazu wird zunächst durch einen Rechtsdurchlauf die Anzahl der S-Typ-Positionen bestimmt. Diese Anzahl wird in das Feld `count_s_type_pos` geschrieben. Falls die Anzahl größer als $\lfloor \frac{n}{2} \rfloor$ ist, werden die Zeichen des Textes so überschrieben, dass sich die Ordnung der Zeichen umdreht. Also falls für zwei Positionen i und j vorher $\mathsf{T}[i] \leq \mathsf{T}[j]$ galt, gilt nach dem Überschreiben $\mathsf{T}[i] \geq \mathsf{T}[j]$ und umgekehrt. Dadurch dreht sich die Ordnung der Suffixe des Textes ebenfalls um. Also muss am Ende des Algorithmus das Suffix-Array SA ebenfalls umgedreht werden.

Wenn man die Vorberechnung auf den Beispieltext anwendet, sieht man, dass es mehr S-Typ-Positionen als L-Typ-Positionen im Beispieltext gibt.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{T}[i]$	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$	$	$
Typ	L	S	S	L	S	L	L	S	S	L	S	L	L	L	S	S	S

Also wird der Eingabetext entsprechend transformiert.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{T}[i]$	a	c	c	b	c	a	a	c	c	b	c	a	c	c	d	d	d
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

Erste Phase

In der ersten Phase des nzSufSort berechnen wir zunächst den reduzierten Text T_{12}. Dies funktioniert ähnlich wie im DC3, indem wir zunächst das Positions-Array der S-Typ-Positionen P_{12} berechnen. Dieses wird mit der Inplace-Variante des Radixsort Abschnitt 4.1.5 anhand der Z_3-Strings sortiert und die lexikographischen Ränge der S-Typ-Positionen werden anhand der Sortierung bestimmt.

Im ersten Schritt muss das Positions-Array P_{12} der S-Typ-Positionen in die ersten n_{12} Elemente von SA berechnet werden. Die S-Typ-Positionen lassen sich einfach durch einen Rechtsdurchlauf durch T bestimmen. Anhand von `count_s_type_pos` lässt sich zu einer S-Typ-Position i ermitteln, ob diese die j-te S-Typ-Position mit j modulo $3 = 1$ oder j modulo $3 = 2$ ist. Ist j modulo $3 = 1$, wird i an Position $\frac{j}{3}$ der ersten Hälfte von P_{12} geschrieben, falls j modulo $3 = 2$ ist an Position $\frac{j}{3}$ der zweiten Hälfte.

In der folgenden Tabelle sehen wir das Positions-Array des Beispieltextes. Der Index j bezeichnet die j-te S-Typ-Position und das Positions-Array ist aufgeteilt in die Mengen mit j modulo $3 = 1$ und j modulo $3 = 2$.

j	1	4	7	2	5
$\mathsf{P}_{12}[j]$	3	9	13	5	11

Im zweiten Schritt müssen wir die Inplace-Variante des Radixsort aufrufen. Da die Z_3-Strings an den Positionen, die in P_{12} gespeichert sind, unterschiedliche Längen besitzen, müssen wir die Längen dieser Strings bestimmen und speichern diese im Speicherbereich der Größe n_{12} direkt hinter P_{12} in SA ab. Dieses Längenarray bezeichnen wir mit H_{12}. Da für die j-te S-Typ-Position die Endposition des zugehörigen Z_3-Strings die S-Typ-Position mit $j + 3$ ist, lässt sich H_{12} durch $\mathsf{P}_{12}[i + 1] - \mathsf{P}_{12}[i] + 1$ berechnen, falls j nicht die letzte S-Typ-Position bezüglich modulo 3 ist. Ansonsten wird der Wert durch $n - \mathsf{P}_{12}[i] + 1$ gebildet.

In der folgenden Tabelle sehen wir das aus P_{12} berechnete Array H_{12}.

j	1	4	7	2	5
$\mathsf{P}_{12}[j]$	3	9	13	5	11
$\mathsf{H}_{12}[j]$	7	5	4	7	6

Um P_{12} zu sortieren, werden wir die Positionen von P_{12} im Speicherbereich der Größe n_{12} direkt hinter H_{12} in SA sortieren. Dabei verwenden wir eine Variation des Inplace-Radixsort, in dem das letzte Bucket nicht sortiert wird. Dieses Bucket enthält die Z_3-Strings, deren Längen kürzer als die aktuell betrachtete Position im Radixsort sind. Für den Aufruf von Radixsort benötigen wir noch eine `key_function` und eine `compare_function`. Die `key_function` gibt das Zeichen von T an der aktuell betrachteten Position zurück, falls die Position kleiner oder gleich der Länge des Z_3-Strings ist. Ansonsten wird $|\Sigma| + 1$ zurückgegeben, damit diese ins letzte Bucket einsortiert werden. Dadurch wird die oben beschriebene Ordnung zwischen Z_3-Strings beibehalten. Die `compare_function` gibt den Rest der Z_3-Strings ab der aktuell betrachteten Position zurück.

In der folgenden Tabelle sehen wir das sortierte Positions-Array P_{12}. Die Indizes i bezeichnen hier die Position in P_{12}.

i	0	1	2	3	4
$\mathsf{P}_{12}[i]$	5	11	3	9	13

Mit dem sortierten Positions-Array P_{12} lässt sich nun T_{12} berechnen. Dazu werden wir durch Vergleiche der Z_3-Strings in P_{12} die lexikographischen Ränge bestimmen und diese - wie in Abschnitt 6.8.1 bereits beschrieben - in die S-Typ-Positionen von SA schreiben. Damit P_{12} nicht überschrieben wird, schreiben wir diesen Speicherbereich umgedreht in L-Typ-Positionen von SA. Dann können wir P_{12} durchlaufen, indem wir von rechts nach links die Typen von T bestimmen und von zwei aufeinanderfolgenden Positionen i und j in P_{12} die jeweiligen Z_3-Strings T_i und T_j bestimmen. Falls $\mathsf{T}_i < \mathsf{T}_j$ erhält T_i einen

kleineren lexikographischen Rang beginnend ab 1. Bei Gleichheit erhalten T_i und T_j den gleichen lexikographischen Rang. Der lexikographische Rang von T_i wird dann an Position i von SA geschrieben. Nachdem alle lexikographischen Ränge bestimmt und in die S-Typ-Positionen von SA geschrieben wurden, teilen wir die lexikographischen Ränge in die beiden Mengen auf, sodass in der ersten Gruppe alle j-ten S-Typ-Positionen mit j modulo $3 = 1$ stehen und in der zweiten alle j-ten S-Typ-Positionen mit j modulo $3 = 2$. Diese schreiben wir zunächst in die L-Typ-Positionen, anschließend werden diese ans Ende von SA kopiert und zuletzt umgedreht an den Beginn von SA geschrieben, um zu vermeiden, dass benötigter Speicher überschrieben wird. Dann steht in den ersten n_{12} Positionen von SA der reduzierte Text T_{12}.

In den folgenden Tabellen wird die Berechnung von T_{12} am Beispieltext veranschaulicht. Zur Übersicht werden die Elemente von T_{12} in rot markiert und die berechneten lexikographischen Ränge in blau. Die Einträge von SA, die für den jeweiligen Schritt der Berechnung unbedeutend sind, werden mit '−' bezeichnet.

Zunächst wird der Zustand zu Beginn der Berechnung gezeigt. In SA ist zunächst nur das Positions-Array P_{12} gespeichert.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA[i]	5	11	3	9	13	-	-	-	-	-	-	-	-	-	-	-	-
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

Anschließend wird das Positions-Array P_{12} umgedreht in den L-Typ-Positionen von SA von rechts nach links gespeichert.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA[i]	-	-	-	-	-	-	-	-	13	-	9	-	-	-	3	11	5
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

Im nächsten Schritt wird P_{12} durchlaufen und die lexikographischen Ränge der Z_3-Strings, die an den jeweiligen Positionen von P_{12} beginnen, bestimmt und in die S-Typ-Positionen von SA geschrieben.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA[i]	-	-	-	3	-	1	-	-	13	4	9	2	-	5	3	11	5
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

Nun müssen die lexikographischen Ränge in die L-Typ-Positionen geschrieben werden.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA[i]	-	-	-	-	-	-	-	-	1	-	2	-	-	-	3	4	5
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

Die lexikographischen Ränge werden anschließend an das Ende von SA kopiert.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{SA}[i]$	-	-	-	-	-	-	-	-	-	-	-	-	1	2	3	4	5
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

Im letzten Schritt werden die lexikographischen Ränge von rechts nach links an den Anfang von SA geschrieben. Dabei werden die Gruppen der i-ten S-Typ-Positionen mit i modulo $3 = 1$ und i modulo $3 = 2$ umgedreht. Dadurch steht am Ende der Berechnung der lexikographischen Ränge am Beginn von SA der reduzierte Text T_{12}.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{SA}[i]$	3	4	5	1	2	-	-	-	-	-	-	-	-	-	-	-	-
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

Im letzten Schritt der ersten Phase wird das Suffix-Array SA_{12} von T_{12} durch einen Aufruf des DC3-Lite Abschnitt 6.4.4 bestimmt, falls die lexikographischen Ränge nicht eindeutig sind. Falls die lexikographischen Ränge eindeutig sind, entspricht T_{12} dem ISA_{12}. Also kann SA_{12} in diesem Fall direkt berechnet werden, indem das Inverse von SA_{12} berechnet wird.

In der folgenden Tabelle sehen wir das Suffix-Array SA_{12} von T_{12}. Da die lexikographischen Ränge von T_{12} eindeutig waren, war kein Aufruf des DC3-Lite nötig.

i	0	1	2	3	4
$\mathsf{SA}_{12}[i]$	3	4	0	1	2

Zweite Phase

In der zweiten Phase berechnen wir SA_0, das Suffix-Array von T_0, indem wir zunächst T_0 analog zu T_{12} berechnen. Anschließend wird SA_0 aus SA_{12} und T_0 analog zum DC3 induziert. SA_0 und SA_{12} werden zum Suffix-Array der S-Typ-Positionen SA_S vereinigt und schließlich durch einen Links-Induktions-Scan das vollständige Suffix-Array SA konstruiert.

Im ersten Schritt der zweiten Phase berechnen wir T_0 analog zu T_{12}, indem wir zunächst das Positions-Array P_0 berechnen, dieses anschließend mit Radixsort sortieren und die lexikographischen Ränge der Positionen bestimmen. Bei der Bestimmung der lexikographischen Ränge ist zu beachten, dass das Suffix-Array SA_{12} nicht überschrieben werden darf und daher ebenfalls in den L-Typ-Positionen von SA gespeichert werden muss. Der berechnete reduzierte Text T_0 ist dann wie folgt.

i	0	1	2
$\mathsf{T}_0[i]$	1	2	3

Dann lässt sich durch Induzieren wie im DC3 Abschnitt 6.4.3 aus T_0 und SA_{12} das Suffix-Array SA_0 bestimmen.

i	0	1	2
$\mathsf{SA}_0[i]$	0	1	2

Um SA_0 und SA_{12} vereinigen zu können, müssen wir die Z_3-Strings der Positionen des ursprünglichen Textes T miteinander vergleichen. SA_0 und SA_{12} beziehen sich aber auf die reduzierten Texte T_0 und T_{12}. Wir müssen also zunächst erneut P_0 und P_{12} berechnen und anschließend die Positionen von SA_0 und SA_{12} mit P_0 und P_{12} aktualisieren, damit sich diese auf T beziehen.

Die Positions-Arrays P_0 und P_{12} werden folgend zusammen mit den aktualisierten Suffix-Arrays SA_0 und SA_{12} dargestellt.

i	0	1	2
$\mathsf{P}_0[i]$	0	6	12
$\mathsf{SA}_0[i]$	0	6	12

i	0	1	2	3	4
$\mathsf{P}_{12}[i]$	3	9	13	5	11
$\mathsf{SA}_{12}[i]$	5	11	3	9	13

Der nächste Schritt ist es nun, SA_0 und SA_{12} zu vereinigen. Dazu benötigen wir das ISA_0 und ISA_{12} der S-Typ-Positionen von T. Dieses wird mit der in Abschnitt 6.8.1 beschriebenen Technik in die S-Typ-Positionen berechnet, indem wir zunächst SA_0 und SA_{12} umgedreht von rechts nach links in die L-Typ-Positionen von SA schreiben. Dabei müssen wir uns in dem Feld h merken, welches die erste Position von SA_0 ist. Anschließend lässt sich durch einen Durchlauf von rechts ISA_0 und ISA_{12} bestimmen. Dabei werden nicht die lexikographischen Ränge der Suffixe gespeichert, sondern die Positionen von SA_0 und SA_{12} in SA, damit man direkt auf die Elemente von SA_0 und SA_{12} zugreifen kann. Da SA_0 und SA_{12} umgedreht in SA abgespeichert wurden, dreht sich auch die Ordnung der Elemente in ISA_0 und ISA_{12} um. Für Positionen i und j die in ISA_0 gespeichert sind gilt $\mathsf{S}_i < \mathsf{S}_j$, wenn $\mathsf{ISA}_0[i] > \mathsf{ISA}_0[j]$. Analog für ISA_{12}.

Nun können wir SA_0 und SA_{12} von rechts nach links durchlaufen und die Positionen im vereinigten Suffix-Array bestimmen. Dazu teilen wir die Positionen von SA_{12} in die *Rest*-1 und in die *Rest*-2 Mengen auf. In der Rest-1 Menge sind die k-ten S-Typ-Positionen mit k modulo $3 = 1$ und in der Rest-2 Menge sind die k-ten S-Typ-Positionen mit k modulo $3 = 2$. Um zu bestimmen zu welcher Menge eine Position i in SA_{12} gehört, bestimmen wir mit der Funktion `retrieve_s_string` die nächste S-Typ-Position j von i. In $\mathsf{SA}[j]$ steht die Information an welcher Position in SA der Eintrag im dazugehörigen SA_0 bzw. SA_{12} gespeichert ist. Durch einen Vergleich mit h lässt sich nun herausfinden, ob j in SA_0 oder SA_{12} gespeichert ist. Wenn j in SA_0 gespeichert ist, gehört i zu der Rest-2 Menge, ansonsten zu der Rest-1 Menge.

Wir vergleichen nun Positionen i in SA_0 mit Positionen j in SA_{12}. Ist j in der Rest-1 Menge, vergleichen wir zunächst die Z_1-Strings T_i und T_j. Falls die Strings ungleich sind, ist die Position im vereinigten Suffix-Array direkt bestimmt. Bei Gleichheit müssen wir die Ordnung der Suffixe der Endpositionen

von T_i und T_j in ISA_0 und ISA_{12} vergleichen. Falls j in der Rest-2 Menge ist, verfahren wir analog zum ersten Fall, nur dass t_i und t_j hier die Z_2-Strings sind.

Die Positionen in SA_0 und SA_{12} werden auf diese Weise mit den Positionen im vereinigten Suffix-Array überschrieben. Indem wir nun ISA_0 und ISA_{12} durchlaufen und die Positionen mit den Positionen aus SA_0 und SA_{12} überschreiben, haben wir das ISA_S der S-Typ-Positionen von T berechnet. Dieses wird an das Ende von SA kopiert.

Das Vereinigen von SA_0 und SA_{12} wird nun am Beispieltext demonstriert. Zu Beginn sind SA_0 und SA_{12} hintereinander in SA gespeichert. Die Elemente von SA_0 werden im Folgenden immer in rot und die Elemente von SA_{12} immer in blau markiert sein.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{SA}[i]$	0	6	12	5	11	3	9	13	-	-	-	-	-	-	-	-	-
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

SA_0 und SA_{12} werden zu Beginn umgedreht von rechts nach links in die L-Typ-Positionen von SA kopiert. Im Feld h wird die Startposition von SA_0 in SA gespeichert. Dies ist im Beispiel die Position 7.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{SA}[i]$	-	-	12	-	6	-	-	0	13	-	9	-	-	-	3	11	5
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

Nun werden ISA_0 und ISA_{12} berechnet und in die S-Typ-Positionen von SA geschrieben. Die Elemente von ISA_0 sind im Folgenden mit orange und die Elemente von ISA_{12} mit grün markiert.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{SA}[i]$	7	-	12	14	6	16	4	0	13	10	9	15	2	8	3	11	5
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

Indem wir SA_0 und SA_{12} von rechts nach links durchlaufen, berechnen wir durch Vergleiche der Z_1-Strings, bzw. Z_2-Strings die Positionen im vereinigten Suffix-Array.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{SA}[i]$	7	-	6	14	2	16	4	1	7	10	5	15	2	8	4	3	0
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

Nun verknüpfen wir ISA_0 und ISA_{12} mit den Positionen von SA_0 und SA_{12} im vereinigten Suffix-Array. Dadurch erhalten wir das inverse Suffix-Array der S-Typ-Positionen ISA_S.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{SA}[i]$	1	-	6	4	2	0	2	1	7	5	5	3	6	7	4	3	0
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

Zuletzt wird das ISA_S an das Ende von SA kopiert.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{SA}[i]$	-	-	-	-	-	-	-	-	-	1	4	0	2	5	3	6	7
Typ	S	L	L	S	L	S	S	L	L	S	L	S	S	S	L	L	L

Durch einen Durchlauf durch ISA_S lässt sich das inverse Suffixarray SA_S berechnen, das Suffix-Array der S-Typ-Positionen von T. Für den nächsten Schritt muss noch das Positions-Array P der S-Typ-Positionen in T berechnet werden, damit die Positionen in SA_S auf T aktualisiert werden.
Im Beispiel wird zunächst gezeigt, wie ISA_S zu SA_S invertiert wurde.

i	0	1	2	3	4	5	6	7
$\mathsf{SA}_S[i]$	2	0	3	5	1	4	6	7

Anschließend wird das Positions-Array P berechnet und SA_S mit den Positionen in P verknüpft.

i	0	1	2	3	4	5	6	7
P	0	3	5	6	9	11	12	13
$\mathsf{SA}_S[i]$	5	0	6	11	3	9	12	13

Im letzten Schritt muss nun aus dem Suffix-Array der S-Typ-Positionen SA_S das vollständige Suffix-Array SA konstruiert werden. Dies geschieht durch einen Links-Induktions-Scan. Dabei werden die Suffixe der S-Typ-Positionen in den S-Teil ihrer Buckets kopiert. Um zu einer S-Typ-Position $k = \mathsf{SA}[i]$ speichereffizient bestimmen zu können, ob $h = \mathsf{SA}[i]-1$ eine L-Typ-Position ist, vergleichen wir zunächst $\mathsf{T}[k]$ und $\mathsf{T}[h]$. Gilt $\mathsf{T}[h] > \mathsf{T}[k]$, ist h eine L-Typ-Position. Gilt $\mathsf{T}[h] = \mathsf{T}[k]$, ist h nur dann eine L-Typ-Position, wenn k eine L-Typ-Position war. Dies lässt sich in konstanter Zeit erreichen, wenn wir den Bucketpointer b vom Zeichen $\mathsf{T}[k]$ mit der Position i vergleichen. Ist $i < b$, befindet sich i im L-Teil des Buckets. Also ist k eine L-Typ-Position. In allen anderen Fällen ist k eine S-Typ-Position.

Das fertig konstruierte Suffix-Array des transformierten Beispieltextes ist dann wie folgt.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{SA}[i]$	5	0	6	11	3	9	4	10	2	8	1	7	12	13	14	15	16

Um das Beispiel zu vervollständigen, muss das SA noch umgedreht werden, da der Beispieltext transformiert wurde. Dies ist das fertige SA vom Beispieltext.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathsf{SA}[i]$	16	15	14	13	12	7	1	8	2	10	4	9	3	11	6	0	5

6.9 DivSufSort

In vielen Bereichen der Algorithmik sind Laufzeiten der entscheidende Faktor beim Vergleich von Algorithmen. Dabei werden bei den Analysen zumeist eher theoretische worst-case Laufzeitschranken betrachtet. In der Praxis hingegen können die Ergebnisse anders aussehen. Ein bedeutendes Beispiel ist dabei der DivSufSort von Mori [39]. Mit einer worst-case Laufzeit von $O(n \log n)$ und einem Speicherbedarf von $5n + O(1)$ mit n als Textlänge liefert der Algorithmus bereits gute Schranken, jedoch nicht die Besten. In der Praxis hingegen liegt mit diesem Algorithmus eine der schnellsten Berechnungen für Suffix-Arrays vor. Da Mori den Algorithmus nur als (kaum kommentierten) Quellcode veröffentlicht hat, werden wir im Folgenden den Algorithmus anhand der Beschreibung von Fischer und Kurpicz [23] erläutern. Dafür werden wir uns zunächst die zugrunde liegenden Algorithmen im Abschnitt Grundlagen anschauen, bevor wir uns den Definitionen und einigen Vorüberlegungen zu Suffix-Arrays widmen. Danach beschreiben wir den eigentlichen Algorithmus.

6.9.1 Grundlagen

Bevor wir zum Algorithmus kommen, müssen zunächst noch einige Definitionen und Grundideen geklärt werden. Dafür betrachten wir zunächst die Präfixsumme:

Definition 16. *Betrachten wir eine Menge $\{a_0, a_1, \ldots, a_{n-1}\}$ und den + Operator, dann ist die Menge der Präfixsummen wie folgt definiert:*

$$[a_0, (a_0 + a_1), \ldots, (a_0 + a_1 + \cdots + a_{n-1})]$$

Die Präfixsumme kann demnach als ein Vektor betrachtet werden, welcher an der i-ten Stelle die ersten i Elemente aufsummiert hat [11]. Später werden wir die Präfixsumme in leicht abgewandelter Form sehen. Suffixe können in drei Typen eingeteilt werden: L-Suffixe, S-Suffixe sowie RMS-Suffixe. Da die L- und S-Suffixe bereits definiert wurden, müssen wir hier nur die RMS-Suffixe beschreiben.

Definition 17. *Suffix S_i ist ein RMS-Suffix (rightmost S-type), falls S_i ein S-Suffix und S_{i+1} ein L-Suffix ist. Falls $\mathsf{T}[i] = \mathsf{T}[i+1]$, so ist S_i je nach Typ von S_{i+1} entweder ein L- oder ein S-Suffix.*

Somit liegen drei verschiedene Typen von Suffixen vor, welche eine Abhängigkeit der Textfolge repräsentieren. Dabei können RMS-Suffixe nicht durch Gleichheit zweier aufeinander folgender Zeichen übertragen werden. Es können maximal $\frac{n}{2}$ RMS-Suffixe vorliegen, da diese durch L-Suffixe definiert sind. Insbesondere schränkt die Zuweisung von Typen zu den Suffixen die Verteilung dieser auf Buckets ein: Ein Bucket $\mathsf{b}_{c0,c1}$ kann nur L-Suffixe enthalten, wenn $|c0| \geq |c1|$ gilt, d.h. der Rang von $c0$ größer als der Rang von $c1$ ist. Ebenso können nur S-Suffixe enthalten sein, wenn $|c0| \leq |c1|$. RMS-Suffixe können nicht

bei $|c0| = |c1|$, d.h. in $\mathsf{b}_{c0,c0}$ vorliegen. Durch diese Einschränkungen wird eine partielle Ordnung unter den Suffixen erzwungen:

Lemma 7. *Seien $\mathsf{S}_i, \mathsf{S}_j$ zwei Suffixe. Dann gilt:*

- $\mathsf{S}_i < \mathsf{S}_j$*, falls* S_i *ein L-Suffix,* S_j *ein S-Suffix ist und* $\mathsf{T}[i] = \mathsf{T}[j]$
- $\mathsf{S}_i < \mathsf{S}_j$*, falls* S_i *ein RMS-Suffix,* S_j *ein S-, aber kein RMS-Suffix ist und* $\mathsf{T}[i, i+1] = \mathsf{T}[j, j+1]$

L- und S-Suffixe können nur gleichzeitig in $\mathsf{b}_{c0,c0}$ auftauchen. Wir nehmen an, S_i und S_j beginnen mit $c0c0$, gefolgt von beliebig vielen $c0$, und $\mathsf{S}_i, \mathsf{S}_j$ sind ein L- bzw. S-Suffix. Sei $u = \mathsf{T}[i + lcp(i,j)]$ und $v = \mathsf{T}[j + lcp(i,j)]$, d.h. $u \neq v$ sind die ersten Zeichen, wo sich S_i und S_j unterscheiden. Da S_i ein L-Suffix ist und sich zuvor vom S-Suffix S_j nicht unterschieden hat, muss $u \leq c0$ sein. Ebenso gilt $v \geq c0$. Eine der Ungleichungen ist strikt erfüllt, da $u \neq v$, damit ist $\mathsf{S}_i < \mathsf{S}_j$. Analog gilt für den zweiten Fall: Die ersten beiden Zeichen von $\mathsf{S}_i, \mathsf{S}_j$ sind identisch. Da S_i ein RMS-Suffix ist, gilt $\mathsf{T}[i] \neq \mathsf{T}[i+1]$. Ebenso gilt $\mathsf{T}[i+1] > \mathsf{T}[i+2]$, da nach der Definition von RMS-Suffixen S_{i+2} ein L-Suffix sein muss. Jedoch ist S_j kein RMS-Suffix, sodass $\mathsf{T}[i+2] < \mathsf{T}[i+1] = \mathsf{T}[j+1] \leq \mathsf{T}[j+2]$ und damit $\mathsf{S}_i < \mathsf{S}_j$.

Um RMS-Suffixe später sortieren zu können, müssen wir zunächst RMS-Teilstrings betrachten.

Definition 18. *Gegeben seien zwei aufeinander folgende RMS-Suffixe S_i und S_j, d.h. es existiert kein RMS-Suffix S_k mit $i < k < j$. Wir bezeichnen den Teilstring $\mathsf{T}[i, j+2)$ als RMS-Teilstring. Für das letzte RMS-Suffix S_i (S_k mit $i < k < n$ ist kein RMS-Suffix) ist der Teilstring $\mathsf{T}[i, n)$ ebenfalls ein RMS-Teilstring.*

Nun haben wir neben den Suffix-Typen und Buckets auch RMS-Teilstrings kennen gelernt. Damit können wir mit der Beschreibung des Algorithmus beginnen.

6.9.2 Algorithmus

Der Algorithmus selbst kann in drei Phasen unterteilt werden. In der ersten Phase müssen zunächst den Suffixen ihre Typen zugewiesen werden. Dafür muss der Text einmal durchlaufen werden. Während dieses Durchlaufes werden ebenfalls die Grenzen für die Buckets b_{c0} bzw. $\mathsf{b}_{c0,c1}$ berechnet. In der zweiten Phase werden die RMS-Suffixe in lexikografischer Reihenfolge sortiert und bereits an die korrekten Positionen im SA gesetzt. Hierfür werden zunächst die RMS-Teilstrings sortiert und deren Ränge bestimmt. Mit diesen Rängen können letztlich die RMS-Suffixe sortiert werden. In der dritten und letzten Phase müssen noch die L- und S-Suffixe an ihre richtigen Positionen gebracht werden. Dafür werden in einem Durchlauf von rechts nach links erst alle S-Suffixe und in einem zweiten Durchlauf von links nach rechts alle L-Suffixe induziert.

Damit diese drei Schritte funktionieren können, benötigen wir noch etwas zusätzlichen Speicher. Wir verwenden zwei zusätzliche Arrays $\mathsf{BUCKET\_L}$ für

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
SA	0	0	0	0	0	0	0	0	0	0	0	2	4	8	10
SA-Typ	L	S	RMS	L	RMS	L	L	S	RMS	L	RMS	L	L	L	L

Tabelle 6.15: Eingabetext mit Suffixtypen und initialisiertem Suffix-Array.

	$	a	b	c	(a,a)	(a,b)	(a,c)
Bucket_L	1	2	2	4			
Bucket_S					2		
Bucket_RMS						2	2
Bucket_L	0	1	9	11			
Bucket_S					2		
Bucket_RMS						2	4

Tabelle 6.16: Berechnung der Bucketgrößen und der Präfixsummen für L- und RMS-Buckets. Die L-Buckets enthalten die Präfixsummen über alle Suffix-Typen (linke Grenze des Buckets), die RMS-Buckets beinhalten die rechten Grenzen des jeweiligen Buckets (bestehend aus zwei Zeichen).

L-Suffixe und BUCKET_S für S- und RMS-Suffixe, um Informationen über die Buckets abspeichern zu können. BUCKET_L der Größe $\sigma = |\Sigma|$ wird über einen Character $c0$ abgerufen, BUCKET_S der Größe σ^2 hingegen über zwei Character $(c0, c1)$. Um zwischen Referenzen für S- und RMS-Suffixen zu unterscheiden, werden diese über $\mathsf{BUCKET\_S}[c0, c1] = \mathsf{BUCKET\_S}[|c0| \cdot \sigma + |c1|]$ bzw. $\mathsf{BUCKET\_RMS}[c0, c1] = \mathsf{BUCKET\_S}[|c1| \cdot \sigma + |c0|]$ abgerufen. Die Buckets für S- und RMS-Suffixe können in demselben Array abgespeichert werden, da in $\mathsf{b}_{c0,c0}$ keine RMS-Suffixe und in $\mathsf{b}_{c0,c1}$ mit $c0 > c1$ keine S-Suffixe enthalten sein können. Die Anzahl der RMS-Suffixe wird mit m bezeichnet.

Initialisierung

Wir beginnen mit einem Durchlauf des Textes T, welcher von rechts nach links durchgeführt wird. In diesem Durchlauf werden zum einen die Typen der Suffixe festgelegt, zum anderen die Größe der Buckets abgespeichert. Zusätzlich wird am Ende des Suffix-Arrays SA die Position jedes RMS-Suffixes im Text abgespeichert, d.h. in $\mathsf{SA}[n - m \dots n)$. Dies bezeichnen wir der Einfachheit halber mit $\mathsf{PAb}[i] = \mathsf{SA}[n - m + i]$. Der Durchlauf von rechts nach links ermöglicht es, auch bei $\mathsf{T}[i] = \mathsf{T}[i - 1]$ sofort den Typen bestimmen zu können, was bei einem Durchlauf von links nach rechts nicht so leicht möglich wäre. Tabelle 6.15 zeigt das initiale Suffix-Array sowie die Suffix-Typen für den Beispielstring $\mathsf{T} =$ caabaccaabacaa$.

Als nächstes werden die Präfixsummen für BUCKET_L und BUCKET_RMS berechnet. Dabei werden für $\mathsf{BUCKET\_L}[c0]$ die Mengen der Buckets aller vorherigen Symbole c_i aufaddiert (inkl. BUCKET_S und BUCKET_RMS). Für die Buckets $\mathsf{BUCKET\_RMS}[c0, c1]$ hingegen muss die Präfixsumme über al-

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	0	2	3	1	0	0	0	0	0	0	0	2	4	8	10

Tabelle 6.17: Finaler Schritt der Initialisierung: Sortierte Referenzen PAb auf Indizes der RMS-Suffixe für den Beispielstring caabaccaabacaa$.

	$	a	b	c	(a,a)	(a,b)	(a,c)
Bucket_L	0	1	9	11			
Bucket_S					2		
Bucket_RMS						0	2

Tabelle 6.18: Finaler Schritt der Initialisierung: Alle RMS-Buckets zeigen auf die linke Grenze des Buckets.

le vorigen BUCKET_RMS$[c_i, c_j]$ mit $c_i \leq c0, c_j \leq c1$ berechnet werden. Dies führt dazu, dass BUCKET_L$[c0]$ die linkeste Position jedes b_{c0} enthält. Für BUCKET_RMS$[c0, c1]$ wird die rechteste Position der RMS-Suffixe in Relation zu anderen RMS-Suffixen abgespeichert, d.h. die Positionen sind aus dem Intervall $[0, m)$. Tabelle 6.16 gibt die initiale Größe der Buckets sowie die berechneten Präfixsummen für den Beispielstring an.

Bevor wir mit dem vollständigen Sortieren der RMS-Suffixe beginnen, werden die Referenzen in PAb auf diese nach den ersten zwei Zeichen sortiert. Die Referenz auf das letzte RMS-Suffix wird an den Anfang des jeweiligen Buckets gesetzt, da kein nachfolgendes RMS-Suffix für dieses Suffix vorhanden ist, welcher jedoch für den Vergleich von RMS-Teilstrings benötigt wird. Nach dieser Sortierung zeigt BUCKET_RMS$[c0, c1]$ auf die linkeste Position aus $[0, m)$. In Tabelle 6.17 sind die Referenzen auf die RMS-Suffixe aus unserem Beispiel derart sortiert.

Nachdem die Initialisierung abgeschlossen wurde, folgt nun der Kern dieses Algorithmus: Das Sortieren der RMS-Suffixe.

RMS-Suffixe sortieren

Das Sortieren der RMS-Suffixe kann in drei Schritte unterteilt werden. Im ersten Schritt werden die RMS-Teilstrings für jeden Bucket $\mathsf{b}_{c0,c1}$ sortiert. Schritt zwei besteht aus der Erzeugung eines partiellen inversen Suffix-Arrays ISAb, in welchem die Ränge der nach den RMS-Teilstrings partiell sortierten RMS-Suffixe enthalten sind. Diese Ränge werden verwendet, um mit einem Verfahren, ähnlich zum Prefix-Doubling, die lexikografische Ordnung aller RMS-Suffixe zu bestimmen. Dies wird mit dem Ansatz der *repetition detection* erweitert.

Index	Referenzindex	RMS-Teilstring
2	0	abac
8	2	abac
10	3	acaa$
4	1	accaab

Tabelle 6.19: Lexikografisch sortierte RMS-Teilstrings. Die Suffixe an Position 2 und 8 haben denselben Teilstring und sind daher nicht eindeutig sortiert.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	0	2	3	1	0	0	0	0	0	0	0	2	4	8	10
SA	0	$\underline{2}$	3	1	0	0	0	0	0	0	0	2	4	8	10

Tabelle 6.20: RMS-Substrings sortiert. Bei identischen Substrings sind alle bis auf den ersten negiert (gekennzeichnet durch _).

Sortieren der RMS-Teilstrings

Der aktuell nicht verwendete Bereich im Suffix-Array $\mathsf{SA}[m \dots n-m)$ dient als Buffer für das Sortieren der RMS-Teilstrings. Den Buffer referenzieren wir folgend als $\mathsf{buf}[i] = \mathsf{SA}[m+i]$ mit $0 \leq i < n-2m$. In jedem BUCKET_RMS werden die RMS-Substrings sortiert. Dies bedeutet, dass mehrere BUCKET_RMS parallel sortiert werden können. Bei p Prozessen erhält jeder Prozess eine Buffergröße von $\frac{|\mathsf{buf}|}{p}$. Die Anzahl der gleichzeitig zu sortierenden Elemente ist beschränkt durch einen Parameter, dessen Default-Wert 1024 beträgt. Ist die Größe des verfügbaren Buffers kleiner als 1024 (d.h. als der Parameter) oder kleiner als die Größe des aktuell zu sortierenden Buckets, so wird der Bucket in kleinere Teilbuckets aufgeteilt, die nach dem Sortieren gemerged werden. Falls der aktuell sortierte Bucket den letzten RMS-Teilstring enthält, so wird dieser bereits an die richtige Position gesetzt, da dieser nicht verglichen werden kann.

Das richtige Sortieren wird über Intro Sort (*ISS*) durchgeführt. Dabei wird Multikey Quicksort $\lfloor \lg(\text{last} - \text{first}) \rfloor$ Male ausgeführt, bevor Heapsort verwendet wird. Es wird dabei nicht rekursiv aufgerufen, sondern über einen Stack implementiert, welcher die unsortierten Teilintervalle enthält. Damit werden die kleineren Teilintervalle immer zuvor verarbeitet. Dies garantiert eine maximale Stack-Größe von $\lg l$, wobei l die Größe des initialen Intervalls bezeichne. Unterschreitet die Größe des aktuell zu sortierenden (Teil-)Buckets einen Schwellwert (Defaultwert ist 8), so wird stattdessen Insertionsort verwendet. Beim Vergleich von Insertionsort werden die RMS-Teilstrings zeichenweise verglichen, angefangen bei der aktuellen Tiefe des Sortierens.

Beim Sortieren kann es vorkommen, dass einige der Teilstrings nicht vollständig sortiert werden können, d.h. sie sind identisch. Alle bis auf den ersten dieser Teilstrings in einem Intervall gleicher Teilstrings werden durch ihre bitweise negierte Referenz abgespeichert. Die erste Referenz in diesem Intervall repräsentiert den Beginn jenes Intervalls. Diese Uneindeutigkeit wird später auf-

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	0	2	3	1	0	0	0	0	0	0	0	2	4	8	10
SA	0	2	3	1	0	3	0	0	0	0	0	2	4	8	10
SA	0	2	3	1	0	3	0	2	0	0	0	2	4	8	10
SA	0	2	-2	1	0	3	1	2	0	0	0	2	4	8	10
SA	0	2	-2	1	1	3	1	2	0	0	0	2	4	8	10
	PAb				ISAb										

Tabelle 6.21: Berechnung des initialen, partiellen inversen Suffix-Arrays ISAb. Unterstrichene Werte wurden im vorigen Schritt nicht eindeutig sortiert, während negative Werte bereits sortierte Intervalle darstellen.

gelöst. Tabelle 6.19 listet alle Teilstrings für unser Beispiel an. RMS-Suffixe 2 und 8 haben dabei einen identischen Teilstring. In Tabelle 6.20 sind die sortierten Indizes in PAb enthalten. Da Index 8 (Referenzindex 2) identisch mit einem anderen war, wird dieser Index negiert.

Partielles inverses Suffix-Array berechnen

Als nächstes können wir ein partielles inverses Suffix-Array berechnen, welches die Ränge der RMS-Suffixe über die bereits partiell sortierten RMS-Teilstrings wiedergibt. Das inverse Suffix-Array repräsentieren wir als $\mathsf{ISAb}[i] = \mathsf{SA}[m+i]$ mit $0 \leq i < m$, d.h. es wird in $\mathsf{SA}[m \dots 2m)$ gespeichert. In $\mathsf{ISAb}[i]$ finden wir somit die Anzahl der kleineren RMS-Suffixe des i-ten RMS-Suffixes. Falls $m > \frac{n}{3}$ gilt, so überschneidet sich ISAb mit PAb. Dies stellt kein Problem dar, da wir die Textpositionen nicht mehr benötigen.

Es wird $\mathsf{SA}[0 \dots m)$ von rechts nach links gescannt. Wenn ein Wert kleiner 0 ist, so erreichen wir ein Intervall, in dem die RMS-Suffixe im vorherigen Schritt nicht eindeutig sortiert wurden. Wir weisen allen von ihnen den größtmöglichen Rang $m - i$ zu. i entspricht dabei der Anzahl der größeren RMS-Suffixe. Zusätzlich müssen die Referenzen bitweise negiert werden (da sie nun „vergleichbar“ sind). Falls der Wert jedoch ≥ 0 ist und dieser nicht nach einem negierten Wert gescannt wurde (d.h. nicht der Anfang eines unsortierten Intervalls ist), so weisen wir den korrekten Rang $m-i$ zu. Wann immer ein komplett sortiertes Intervall erkannt wird, so wird die Anfangsposition dieses Intervalls in $\mathsf{SA}[0 \dots m)$ mit $-k$ markiert, wobei k der Größe dieses Intervalls entspricht. Somit können alle sortierten Intervalle erkannt werden, da diese mit einem negativen Wert beginnen, dessen Absolutwert der Länge dieses Intervalls gleicht. Tabelle 6.21 zeigt die schrittweise Berechnung für das initiale partielle ISA. Die letzten beiden Indizes ergeben ein sortiertes Intervall, weshalb an $\mathsf{PAb}[2]$ der Wert -2 eingetragen wird. Die relativen Indizes 0 und 2 erhalten denselben Rang, da ihre Teilstrings identisch sind.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
SA	-4	0	-2	1	1	3	0	2	0	0	0	2	4	8	10
SA	8	2	10	4	1	3	0	2	0	0	0	2	4	8	10

Tabelle 6.22: Bestimmung der absoluten RMS-Suffixindizes in der korrekten Reihenfolge (in $\mathsf{SA}[0 \dots 4)$) anhand der Werte in ISAb. Unterstrichene Werte kennzeichnen die Negierung vor dem ersten Induzieren.

RMS-Suffixe sortieren

Zu guter Letzt können wir die korrekten Ränge aller RMS-Suffixe bestimmen und diese in ISAb abspeichern. Die Ränge sind dabei zum Sortieren ausreichend, d.h. PAb und der Zugriff auf den originalen Text wird nicht mehr benötigt. Der verfolgte Ansatz ähnelt dabei dem des Prefix-Doubling: in jeder Iteration k betrachten wir nicht die doppelte Präfixlänge, sondern die doppelte Anzahl der zu betrachtenden RMS-Teilstrings, welche eine beliebige Länge haben können. Mit $\mathsf{ISAd}[i]$ referenzieren wir den Rang des $i + 2^k$-ten RMS-Suffixes. Die Ränge der RMS-Suffixe müssen dabei aktualisiert werden, wenn die Anzahl der betrachteten RMS-Teilstrings verdoppelt wird. Da die Ränge der RMS-Suffixe in ISA in Textreihenfolge gegeben sind, kann der Rang des nächstgelegenen RMS-Teilstrings jederzeit für beliebige Teilstrings abgerufen werden.

Repetition-Detection

Bevor wir uns mit dem Induzieren der L- und S-Suffixe beschäftigen, folgt zuvor noch eine Anpassung mittels *repetition detection*. Diese Anpassung beschreibt den Sortierschritt im vorigen Absatz.

Definition 8 besagt, wie eine Wiederholung in einem Text definiert ist. Dies kann problematisch werden, falls S_i ein RMS-Suffix ist, denn dann wäre S_{kp} ebenfalls ein RMS-Suffix für $k \leq r$. Um dies aufzulösen, können wir das erste Symbol betrachten, welches nicht Teil der Wiederholung ist, d.h. $\mathsf{T}[i + l] \neq \mathsf{T}[i + rp + l]$. Falls $\mathsf{T}[i + rp + l] < \mathsf{T}[i + l]$, dann gilt für $1 < i \leq r$:$\mathsf{T}[i + (r - 1)p + 1, i + rp] < \mathsf{T}[(i - 1) + (r - 1)p + 1, (i - 1) + rp]$. Mit anderen Worten, die RMS-Suffixe dieser Wiederholung sind in absteigender Reihenfolge sortiert. Der analoge Fall gilt für $\mathsf{T}[i + rp + l] > \mathsf{T}[i + l]$, d.h. $\mathsf{T}[i + (r - 1)p + 1, i + rp] > \mathsf{T}[(i - 1) + (r - 1)p + 1, (i - 1) + rp]$, die RMS-Suffixe liegen in aufsteigender Reihenfolge vor.

Verwenden wir Quicksort mit den zuvor sortierten Rängen als Schlüssel, so können wir diese *repetition detection* verwenden. Wir verwenden den Rang des Medians der aktuell betrachteten RMS-Suffixe als Pivotelement. Ist der aktuelle Rang des ersten RMS-Suffixes im aktuell betrachteten Teilintervall gleich zu dem Pivotelement, so liegt eine Wiederholung vor ($\mathsf{ISAb}[i] = \mathsf{ISAd}[i]$). Wir verwenden wieder für $\lfloor \lg(\text{last} - \text{first}) \rfloor$ Male Quicksort, bevor wir Heapsort zum Sortieren verwenden. Nachdem alle RMS-Suffixe durch das Sortieren in lexikografischer Reihenfolge liegen, durchlaufen wir den Text T erneut von rechts nach links.

	$	a								b		c			
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
SA	8	2	10	4	1	8	2	10	4	0	0	2	4	8	10

Tabelle 6.23: Setzen der Indizes der sortierten RMS-Suffixe an die richtige Position in SA.

	$	a	b	c	(a,a)	(a,b)	(a,c)
Bucket_L	0	1	9	11			
Bucket_S					4	6	8
Bucket_RMS						3	2

Tabelle 6.24: Berechnung der S- und RMS-Buckets für das Induzieren. Für die S-Buckets wurden die rechten Grenzen der jeweiligen Buckets bestimmt (grün markiert). Bei RMS-Buckets wird nur der Bucket $\mathsf{b}_{c0,c0+1}$ berechnet (blau markiert), um die linke Grenze für S-Suffixe zu markieren (läuft beim Induzieren der S-Buckets nur bis zu dieser Grenze für b_{c0})

Beobachten wir RMS-Suffix S_i an Position j, so speichern wir den Index ab in $\mathsf{SA}[\mathsf{ISAb}[i]] = j$. Da wir im darauf folgenden Durchlauf zunächst nur S-Suffixe, aber keine L-Suffixe induzieren möchten, speichern wir die bitweise Negation von j, falls S_{j-1} ein L-Suffix ist. Die korrekten RMS-Indizes für unser Beispiel sind in Tabelle 6.22 dargestellt.

Nun sind die Positionen aller RMS-Suffixe im Text in $\mathsf{SA}[0 \ldots m)$ in lexikografischer Reihenfolge abgespeichert. Diese müssen als nächstes an ihre richtige Position gebracht werden. Das Ergebnis für unser Beispiel sieht man in Tabelle 6.23. Währenddessen können BUCKET_S sowie BUCKET_RMS derart angepasst werden, dass alle S-Buckets die rechte Grenze des jeweiligen Buckets beinhalten, um beim Induzieren das S-Suffix direkt an die richtige Position zu setzen. Für RMS-Suffixe wird dabei nur der Bucket $\mathsf{b}_{c0,c0+1}$ aktualisiert, da diese die linke Grenze eines Buckets b_{c0} beinhalten, bis zu welcher S-Suffixe eingefügt werden können. Tabelle 6.24 gibt die Berechnung der Buckets für unseren Beispielstring an.

Induzieren von L- und S-Suffixen

Durch die Typen der Suffixe wissen wir, dass in beliebigen Buckets $\mathsf{b}_{c0,c1}$ L-Suffixe lexikografisch kleiner als S-Suffixe sowie RMS-Suffixe kleiner als S-Suffixe sind. Wir wissen ebenfalls, dass bei lexikografischer Ordnung alle direkt aneinandergereihten S-Suffixe links von mindestens einem RMS-Suffix sind (welches zu einem L-Suffix wechselt), da S_{n-1} ein L-Suffix ist. Ebenso sind (in lexikografischer Reihenfolge) alle L-Suffixe rechts von mindestens einem S-Suffix. Das Suffix-Array wird nun zwei Mal durchlaufen. Beim ersten Mal laufen wir von

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Bucket_S[a, a]
SA	14	2	10	4	1	8	2	10	**4**	0	0	2	4	8	10	4
SA	14	13	10	4	1	8	2	**10**	4	0	0	2	4	8	10	4
SA	14	13	10	4	1	8	**2**	10	4	0	0	2	4	8	10	4
SA	14	13	10	4	1	**8**	2	10	4	0	0	2	4	8	10	3
SA	14	13	10	7	**1**	8	2	10	4	0	0	2	4	8	10	2
SA	14	13	10	**7**	1	8	2	10	4	0	0	2	4	8	10	2
SA	14	13	10	7	1	8	2	10	4	0	0	2	4	8	10	2
SA	14	2	10	7	1	8	2	10	4	0	0	2	4	8	10	2

Tabelle 6.25: Induzieren der S-Suffixe. Neu eingefügte Indizes sind grün hinterlegt, wohingegen blau hinterlegte Werte negiert wurden. Es wird immer der fett markierte Index zum Induzieren betrachtet.

rechts nach links und induzieren alle S-Suffixe, beim zweiten Mal laufen wir von links nach rechts und induzieren die L-Suffixe.

Beim ersten Scan des SA speichern wir jedes Mal den Eintrag $i-1$ an die rechteste freie Position im Bucket $\mathsf{b}_{c0,c1}$, wenn wir einen Eintrag $i > 0$ lesen. Falls $\mathsf{T}[i-2] > \mathsf{T}[i-1]$, so ist S_{i-2} ein L-Suffix und wir negieren den Wert von $i-1$ bitweise, da S_{i-2} in diesem Schritt nicht induziert wird. Bei diesem Durchlauf wird jeder Wert durch seinen bitweise negierten Wert überschrieben. Falls eine Stelle bereits bitweise negiert war, so wird sie im nächsten Scan betrachtet, da es induziert wurde und das dazugehörige Suffix ein L-Suffix ist (daher in diesem Durchlauf nicht relevant). Alle Suffixe, welche zur Induzierung verwendet wurden, haben ihre Position bitweise negiert, da sie ein S-Suffix induziert haben. Alle Anderen werden hingegen durch ihre Position repräsentiert und sind für den nächsten Durchlauf relevant. Alle induzierten Suffixe sind dabei lexikografisch kleiner als jene, von denen induziert wurde, da in diesem Durchlauf S-Suffixe betrachtet wurden und damit $c0 \leq c1$ für alle Buckets $\mathsf{b}_{c0,c1}$ gelten muss. Ebenso können wir nur in $\mathsf{b}_{c0,c1}$ mit $c1 \leq c0$ induzieren, da nur S-Suffixe betrachtet wurden. Für unser Beispiel können wir in Tabelle 6.25 sehen, wie Schrittweise die Indizes für b_a überprüft und induziert werden. Der letzte zu prüfende Index 3 ist in BUCKET_RMS$[a, b]$ gespeichert (s. Tabelle 6.24).

Vor dem zweiten Durchlauf von links nach rechts wird $n-1$ an den Anfang des $\mathsf{T}[n-1]$-Buckets gesetzt. Falls S_{n-2} ein L-Suffix ist, so speichern wir $n-1$, da wir S_{n-2} induzieren wollen. Andernfalls soll der bitweise negierte Wert von $n-1$ abgespeichert werden. Nun kann der Durchlauf beginnen. Liegt ein Eintrag $i < 0$ vor, so wurde dieser bereits an seine richtige Position gesetzt und bitweise negiert, damit die korrekte Position an dieser Stelle steht. Ist $i > 0$, so muss das Suffix S_{i-1} an die linkeste freie Position im Bucket $\mathsf{b}_{\mathsf{T}[i-1]}$ induziert werden. Alle übrigen Suffixe werden in diesem Durchlauf induziert, sodass die linke Grenze für b_{c0}, welche in BUCKET_L$[c0]$ gespeichert wird, ausreichend ist. Wenn das induzierte Suffix S_{i-1} ein S-Suffix induzieren würde, so wird stattdessen der bitweise negierte Wert induziert, da das Suffix beim scannen des Indizes übersprungen (nur negiert) wird. In Tabelle 6.26 sind im oberen Teil die Schritte für

Schritt	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	SA	**14**	2	10	7	1	8	2	10	4	0	0	2	4	8	10
1	SA	14	**13**	10	7	1	8	2	10	4	0	0	2	4	8	10
2	SA	14	13	**12**	7	1	8	2	10	4	0	0	2	4	8	10
3	SA	14	13	12	**7**	1	8	2	10	4	0	0	11	4	8	10
4	SA	14	13	12	7	**1**	8	2	10	4	0	0	11	6	8	10
5	SA	14	13	12	7	1	**8**	2	10	4	0	0	11	6	0	10
6	SA	14	13	12	7	1	8	**2**	10	4	0	0	11	6	0	10
7	SA	14	13	12	7	1	8	2	**10**	4	0	0	11	6	0	10
8	SA	14	13	12	7	1	8	2	10	**4**	9	0	11	6	0	10
9	SA	14	13	12	7	1	8	2	10	4	**9**	3	11	6	0	10
10	SA	14	13	12	7	1	8	2	10	4	9	**3**	11	6	0	10
11	SA	14	13	12	7	1	8	2	10	4	9	3	**11**	6	0	10
12	SA	14	13	12	7	1	8	2	10	4	9	3	11	**6**	0	10
13	SA	14	13	12	7	1	8	2	10	4	9	3	11	6	0	5

Schritt	Bucket_L[$]	Bucket_L[a]	Bucket_L[b]	Bucket_L[c]
0	1	1	9	11
1	1	2	9	11
2	1	3	9	11
3	1	3	9	12
4	1	3	9	13
5	1	3	9	14
6	1	3	9	14
7	1	3	9	14
8	1	3	10	14
9	1	3	11	14
10	1	3	11	14
11	1	3	11	14
12	1	3	11	14
13	1	3	11	15

Tabelle 6.26: Induzieren der L-Suffixe. Grün markierte Indizes wurden neu eingefügt, fett hervorgehobene Indizes werden zum Induzieren verwendet und blau hinterlegte Werte wurden negiert.

die Indizes beim Induzieren dargestellt. Im unteren Teil werden die Änderungen der jeweiligen BUCKET_L nach Einfügen eines Suffixes aufgelistet.

6.9.3 Implementierung

In unserer Implementierung wurden einige Stellen vereinfacht umgesetzt als bei der Referenz und in den vorigen Abschnitten beschrieben. Die Typen wurden (bis auf den Linksdurchlauf beim Induzieren der L-Suffixe) anhand von externen Methoden bestimmt, welche den Typen des Nachfolgers übergeben bekommt, um den Typen bei gleichen Zeichen korrekt zu bestimmen. Beim Sortieren der Teilstrings wurde statt des Multikey-Quicksorts innerhalb des Introsorts eine Vergleichsfunktion gewählt, welche alle Teilstrings enthält und diese (bei zwei gegebenen Suffix-Indizes) Zeichenweise vergleicht. Der größte Unterschied liegt beim Sortieren der RMS-Suffixe, nachdem die Teilstrings vorsortiert und das initiale partielle ISA bestimmt wurde. In der Referenz wurde die Repetition-Detection innerhalb des Quick- bzw. Introsorts integriert. In unserer Implementierung hingegen ist noch eine einfachere, iterative Idee verbaut: Wir durchlaufen das PAb, d.h. die Referenzindizes bzw. Indikatoren für sortierte Intervalle und suchen unsortierte Intervalle bei einem Durchlauf von links nach rechts. Bei sortierten Intervallen können über den eingetragenen negierten Wert das komplette Intervall übersprungen werden. Bei unsortierten Intervallen müssen wir zum Einen prüfen, ob es sich um ein einzelnes Intervall handelt. Dies können wir daran erkennen, dass alle Ränge identisch sind. Bei einem unterschiedlichen Rang beginnt ein weiteres, unsortiertes Intervall. Ist ein komplettes unsortiertes Intervall bestimmt, kann anhand des Teilstring-Doublings über den Rang des nächsten Teilstrings durch Verdoppelung sortiert werden. Danach müssen in diesem Intervall alle Ränge neu berechnet werden, um zwischen neuen sortierten und unterschiedlichen unsortierten Intervallen weiterhin unterscheiden zu können. Dies wird solange wiederholt, bis in einem vollständigen Durchlauf der Referenzen kein unsortiertes Intervall vorhanden ist. Dann erst sind alle Ränge eindeutig bestimmt und die Suffixe können an die richtige Position ins Suffix-Array gesetzt werden.

6.10 SAIS

Der **Suffix Array Induced Sorting-Algorithmus**[44, Kap. 3] (kurz: SAIS) ist unter den Induzierern einzuordnen. Dabei verfolgt er einen rekursiven Ansatz, um eine lineare Laufzeit bei der Berechnung des Suffix-Arrays zu erzielen. Im Folgenden wird diese Strategie beleuchtet und anhand eines Beispiels nachvollzogen.

6.10.1 Framework

```
1  def SAIS(T,SA):
2  # T is the input string;
3  # SA is the output suffix array of T
4  # t: array[0, n − 1] of boolean;
5  # P_1, T_1: array[0, n_1 − 1] of integer;
6  # B: array[0, |Σ(T)| − 1] of integer;
7  Scan T once to classify all the characters as L- or S-Type into
   ↪ t;
8  Scan t once to find all the LMS-substrings in T into P_1;
9  Induced sort all the LMS-substrings using P_1 and B;
10 Name each LMS-substring in T by its bucket index to get a new
   ↪ shortened string T_1;
11 if Each character in T_1 is unique
12   Directly compute SA_1 from T_1;
13 else
14   SAIS(T_1, SA_1); # Fire a recursive call
15 Induce SA from SA_1
```

Abbildung 6.41: SAIS-Framework

Der oben stehende Algorithmus [44, Fig. 1] berechnet das Suffix-Array des Strings T. Grundsätzlich ist das Problem in zwei Teile aufgeteilt. Das Reduzieren des Problems zu Teilproblemen (Divide and Conquer) und das Induzieren der LMS-Substrings und der Teilsuffix-Arrays. Am deutlichsten wird die Funktionsweise des Algorithmus durch ein Beispiel.

6.10.2 Verfahren

Sei T = *caabaccaabacaa*\$ der String zu dem das Suffix-Array konstruiert werden soll und SA das zugehörige Suffix-Array. Zunächst wird für jedes Zeichen des Strings T der Typ t, nach den Definitionen aus Kap. 5.2.3, ermittelt.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	c	a	a	b	a	c	c	a	a	b	a	c	a	a	\$
t	L	S	S	L	S	L	L	S	S	L	S	L	L	L	S

Tabelle 6.27: Klassifizierung des Strings T

Der nächste Schritt besteht daraus, alle LMS-Substrings zu finden. Besonders am SAIS ist, dass die LMS-Substrings eine variable Länge besitzen. Diese Länge wird beim später beschriebenen SADS beschränkt. Diese Schritte besitzen eine obere Laufzeitschranke von $\mathcal{O}(n)$, da ein einmaliger Durchlauf von rechts nach links ausreicht, um alle Typen(L, S und LMS) zu bestimmen.
Für die LMS-Substrings variabler Länge ergibt sich folgende Berechnung, wobei der Anfang des LMS durch einen Stern gekennzeichnet ist. Das Ende des LMS-Substrings ergibt sich aus dem nächstfolgenden Stern bzw. dem Ende des Wortes.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	c	a	a	b	a	c	c	a	a	b	a	c	a	a	\$
	L	S	S	L	S	L	L	S	S	L	S	L	L	L	S
		*			*			*			*				*

Tabelle 6.28: LMS-Berechnung

Nachdem die LMS-Substrings bestimmt sind, folgt das Unterteilen in Buckets und Buckettypen. Die verschiedenen im Text vorkommenden Zeichen bilden die *Basisbuckets*. Jeder dieser *Basisbuckets* ist außerdem in einen L- und einen S-Teil aufgeteilt. Der L-Teil eines Buckets steht immer vor dem S-Teil eines Buckets. Außerdem ist die Bucketgröße gleich der entsprechenden Anzahl der im Text vorkommenden Zeichen. Die Buckets selbst sind lexikographisch sortiert.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	c	a	a	b	a	c	c	a	a	b	a	c	a	a	\$
	L	S	S	L	S	L	L	S	S	L	S	L	L	L	S
		*			*			*			*				*
Bucket	\$	a								b		c			
Typ	S	L		S						L		L			

Tabelle 6.29: Bucket-Berechnung

Die zugrundeliegenden LMS-Substrings müssen im nächsten Schritt sortiert werden. Problem: Sortiere LMS-Substrings $[14, 10, 7, 4, 1]$

Es ergibt sich also folgende Konstellation:

1	4	7	10	14
aaba	acca	aaba	acaa$	$
SSLS	SLLS	SSLS	SLLLS	S

Tabelle 6.30: LMS-Substring-Sortierung

Durch das Vergleichen der jeweiligen Zeichen und Typen kann so folgende Reihenfolge berechnet werden: $[14, 7*, 1*, 10, 4]$
Problem: Dadurch, dass die Substrings identisch sind (hier mit einem Stern markiert), ist noch keine eindeutige Reihenfolge entstanden. Es muss ein weiterer Rekursionsschritt durchgeführt werden.
Analog zum Anfang werden den Substrings Buckets zugeordnet:

T_1	1	3	1	2	0
	1	4	7	10	14

Tabelle 6.31: Substring-Bucket-Berechnung

Die Zahlen 1 und 7 werden demselben Bucket zugeordnet, da diese noch nicht sortiert werden konnten. Analog dazu wird der 14 der *kleinste* Bucket zugeordnet, da das $-Zeichen das lexikographisch kleinste Zeichen ist. Auf den so gewonnenen String T_1 wird nun der SAIS angewendet. Zunächst werden die LMS-Indizes $[4, 2]$ ihren *Buckets* zugeordnet. Es folgt eine Iteration von links nach rechts. Entspricht das Vorgängerzeichen Typ L, wird es an der nächst linken freien Position des jeweiligen Buckets eingeordnet. Anschließend folgt eine Iteration von rechts nach links, bei der das Vorgängerzeichen an der nächst rechten freien Position einsortiert wird, wenn es vom Typ S ist.

	1	4	7	10	14
i	0	1	2	3	4
T_1	1	3	1	2	0
t_1	S	L	S^*	L	S^*
Buckets	0	1		2	3
	S	S		L	L
	4		(2)	3	1
		2	0		

Tabelle 6.32: Induzieren von SA_1

Es ist zu beachten, dass die rot markierte 2 im zweiten Durchlauf von der 0 überschrieben wird, im nächsten Schritt aber direkt wieder von der 3 induziert wird.

Das Ergebnis des Induzierens ist das Suffix-Array $\mathsf{SA}_1 = [4, 2, 0, 3, 1]$. Damit kann auf die ursprüngliche Reihenfolge geschlossen werden. Demnach ist das sortierte LMS-Array: $[14, 7, 1, 10, 4]$
Die Rekursionstiefe von 1 ändert sich wieder auf die Tiefe von 0 und das ursprüngliche Problem wird gelöst, da jetzt die Reihenfolge eindeutig bestimmt ist. Analog zum Rekursionsschritt, werden zunächst die LMS-Indizes in die jeweiligen *S-Buckets* einsortiert, um dann durch eine links-rechts und rechts-links-Iteration das Suffix-Array SA zu berechnen:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
	L	S	S	L	S	L	L	S	S	L	S	L	L	L	S
		*			*			*			*				*
Bucket	$	a								b		c			
Typ	S	L		S						L		L			
	14			7	1	(10)	(4)								
		13								9	3		6	0	
			12												5
												11			
						8	2	10	4						

Tabelle 6.33: Induzieren von SA

Von links nach rechts ergibt sich das endgültige Suffix-Array
$\mathsf{SA} = [14, 13, 12, 7, 1, 8, 2, 10, 4, 9, 3, 11, 6, 0, 5]$.

6.10.3 Optimierungsmöglichkeiten

Der SAIS benötigt nur noch an zwei Stellen Extra-Speicher. Zum einen benötigen die Buckets zusätzlichen Speicher in Größe der Alphabetgröße. Zum anderen benötigt der SAIS einen Bitvektor, um die L- und S-Typen zu speichern. Alle anderen Berechnungen geschehen *on the fly*, benötigen also keinen Extra-Speicher. So können die LMS-Typen leicht berechnet werden, indem sich vom aktuellen Typ des Zeichens der Vorgänger angeschaut wird. Wenn dieser vom Typ L ist, dann ist das Zeichen ein LMS. Aufwändig an dieser Stelle ist, dass momentan noch geteilte Schleifen, die das Wort von rechts nach links bzw. von links nach rechts durchlaufen kompaktifiziert werden müssen, damit die Typen, die berechnet wurden für alle Berechnungsschritte genutzt werden können. Interessant an dieser Stelle ist der Kompromiss zwischen Laufzeit und Extra-Speicher. Der Algorithmus kann auf Grund des Designs unseres Frameworks nicht nur für 8-Bit, sondern auch für 16-, 32-, 48-Bittypen usw. ausgeführt werden.

6.11 SADS

6.11.1 D-Critical

Der Kern des **Radix Sorting Fixed Length D-Critical Substring-Algorithmus**[44, Kap. 4] (kurz: SADS) sind die **d-critical** (kurz: dc) Zeichen des Strings T. Ein Zeichen ist dc, für ein beliebiges aber festes $d \geq 2$, wenn

- $\mathsf{T}[i]$ ist LMS oder
- $\mathsf{T}[i-d]$ ist dc, $\mathsf{T}[i+1]$ ist kein LMS und kein weiteres Zeichen in $\mathsf{T}[i-d+1, i-1]$ ist dc. Um die Notation einfach zu halten, sei $d_1 = d+1$ für den Rest des Kapitels.

Für den Rest des Kapitels gilt $\mathbf{d = 2}$.

Es folgt ein Beispiel für eine Klassifizierung, die dc-Zeichen (hier mit ** markiert) enthält:

i	0	1	2	3	4	5
T	b	a	c	a	a	\$
	L	S	L	L	L	S
		*		**		*

Tabelle 6.34: Beispiel einer dc-Berechnung

$\mathsf{T}[1]$ und $\mathsf{T}[5]$ sind Zeichen, die als LMS klassifiziert werden. Dadurch sind sie auch dc-Zeichen. Auch das Zeichen $\mathsf{T}[3]$ ist ein dc-Zeichen, denn $\mathsf{T}[3-2] = \mathsf{T}[1]$ ist ein dc-Zeichen, $\mathsf{T}[3+1] = \mathsf{T}[4]$ ist kein LMS-Zeichen und in dem Intervall $\mathsf{T}[3-2+1, 3] = \mathsf{T}[2, 3]$ ist kein weiteres dc-Zeichen. Damit ist $\mathsf{T}[3]$ auch ein dc-Zeichen. Im Gegensatz zu den Zeichen $\mathsf{T}[1]$ und $\mathsf{T}[5]$ ist $\mathsf{T}[3]$ also ein Zeichen, das nicht vom SAIS als LMS, sondern nur vom SADS als dc-Zeichen klassifiziert wird.

Es folgen Definitionen, die für den weiteren Verlauf des Algorithmus notwendig sind:

- Ein Suffix $\mathsf{T}[i, n-1]$ gilt als dc, genau dann wenn $\mathsf{T}[i]$ dc ist.
- Seien $\mathsf{T}[i]$ und $\mathsf{T}[j]$ zwei dc-Zeichen. Sie gelten als **benachbart**, wenn zwischen ihnen kein weiteres dc-Zeichen vorkommt.
- Der Substring $\mathsf{T}[i, i+d_1]$ ist der dc-Substring des Zeichens $\mathsf{T}[i]$. Dabei entspricht die Länge des Substrings immer $d_1 + 1$, sodass er wenn nötig mit dem Sentinel ergänzt wird.
- Sei $\mathsf{T}[i]$ dc, dann gilt, dass $\mathsf{T}[i-1]$ und $\mathsf{T}[i+1]$ nicht dc sind.
- $\mathsf{T}[0]$ ist kein dc und $\mathsf{T}[n]$ ist ein dc mit $n = |\mathsf{T}| - 1$

Definition ω-Gewichtung

Sei $\omega(\mathsf{T}, i)$ die ω-gewichtende Funktion, definiert als $\omega(\mathsf{T}, i) = 2\mathsf{T}[i] + t[i]$. Ein S-Typ entspricht dabei einer 1 und ein L-Typ einer 0.

6.11.2 Framework

Ähnlich des Frameworks des SAIS-Algorithmus in Kap. 6.10.1, ist auch der SADS-Algorithmus in einem Framework[44, Fig. 3] modelliert:

```
1  def SADS(T, SA)
2  # T is the input string;
3  # SA is the output suffix array of T
4  # t: array[0, n − 1] of boolean;
5  # P1, S1: array[0, n1 − 1] of integer;
6  # B: array[0, |Σ(T)| − 1] of integer;
7  Scan T once to classify all the characters as L- or S-Type into
   ↪ t;
8  Scan t once to find all the d-critical-substrings in T into P1;
9  Bucket sort all the d-critical substrings using P1 and B;
10 Name each d-critical-substring in T by its bucket index to get a
   ↪ new shortened string T1;
11 if |T1| = Number of Buckets
12    Directly compute SA1 from T1;
13 else
14    SADS(T1, SA1); # Fire a recursive call
15 Induce SA from SA1
```

Abbildung 6.42: SADS-Framework

Analog zum ersten Algorithmus wird das Problem in Probleme kleinerer Größe aufgeteilt, um abschließend die Lösung des Problems zu induzieren. Sei der String $\mathsf{T} = caabaccaabacaa\$$ wieder der String, zu dem das Suffix-Array konstruiert wird. Wie oben festgelegt gilt $d = 2$. Der erste Schritt besteht daraus die Typen aus T zu bestimmen, sodass folgende, schon aus dem SAIS bekannte, Klassifizierung entsteht:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	c	a	a	b	a	c	c	a	a	b	a	c	a	a	\$
t	L	S	S	L	S	L	L	S	S	L	S	L	L	L	S

Tabelle 6.35: Klassifizierung des Strings T

Mit Hilfe der Definitionen aus Kap. 6.11.1 werden die d-critical Zeichen bestimmt:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
t	L	S	S	L	S	L	L	S	S	L	S	L	L	L	S
P_1		*			*			*			*		**		*

Tabelle 6.36: dc-Berechnung

Die einfachen dc-Zeichen sind hier *normale* LMS-Zeichen, die genauso auch beim SAIS berechnet wurden. Das doppelte dc-Zeichen entsteht durch die Anpassungen des SADS, die dafür sorgt, dass maximal Substrings der Größe d_1 entstehen. Durch die Definitionen aus Kap. 6.11.1 wird außerdem deutlich, dass es maximal $\lfloor \frac{n}{2} \rfloor$ dc-Zeichen geben kann.

Die gefundenen sechs Substrings müssen nun sortiert werden. Dazu nutzt der Algorithmus ein Bucket-Sort mit vier Passes, beginnend mit dem LSC(Least Significant Character). Zunächst werden die Wörter anhand ihres jeweiligen LSC ihren Buckets zugeordnet. Dort werden sie anhand der ω-Gewichtung aus Kap. 6.11.1 in ihren Buckets sortiert. In den darauffolgenden Schritten wird das nächste Zeichen des jeweiligen Wortes analysiert und das Wort wird dem passenden Bucket zugeordnet. Angewandt auf die dc-Substrings ergibt sich so folgende Reihenfolge:

$	14 $$$**$** 12 aa$**$**	14 $$**$**$$ 12 aa**$**$$	14 $**$**$$$	14 **$**$$$	0
a	10 aca**a** 7 aab**a** 4 acc**a** 1 aab**a**	10 ac**a**a	12 a**a**$$ 7 a**a**ba 1 a**a**ba	12 **a**a$$ 7 **a**aba 1 **a**aba 10 **a**caa 4 **a**cca	1 2 2 3 4
b		7 aa**b**a 1 aa**b**a			
c		4 ac**c**a	10 a**c**aa 4 a**c**ca		
	Gewichten	Zeichensortierung	Zeichensortierung	Zeichensortierung	Bucketing

Tabelle 6.37: Radix-Sorting

Im ersten Schritt findet die ω-Gewichtung statt. Dadurch findet eine initiale Vorsortierung in den jeweiligen LSC-Buckets statt. Es folgt die eigentliche Radix-Sortierung. Die Suffixe $[14, 12, 10, 7, 1, 4]$ werden anhand des vorletzten Zeichens sortiert. Dabei bleiben $[14, 12, 10]$ an derselben Stelle und in denselben Buckets, da ihr Zeichen identisch zum Nachfolgerzeichen ist. Es findet also keine

Umsortierung statt. Substrings $[7, 1]$ gehören aufgrund ihres Zeichens nun zum *b-Bucket* und sind aufgrund ihrer Reihenfolge im Schritt davor implizit in ihren Buckets sortiert worden. Analog dazu befindet sich 4 als einziger String im *c-Bucket*. Im nächsten Schritt werden $[12, 7, 1]$ zum *a-Bucket* zusortiert. Dabei ist die 12 an erster Stelle, da sie aus dem obersten Bucket kommt. Äquivalent dazu ist im letzten Schritt die Sortierung final und die Indizes können ihren Substrings zugeordnet werden. Dabei ist zu beachten, dass identische Strings denselben Index erhalten.
Aus dem Bucketsort ist folgende Bucket-Zuordnung der dc-Zeichen enstanden:

T_1	2	4	2	3	1	0
	1	4	7	10	12	14

Tabelle 6.38: dc-Bucket-Zuordnung

Ähnlich zum SAIS ist auch hier eine Rekursion der Tiefe 1 nötig, da der Bucket 2 doppelt vergeben ist, wodurch noch nicht auf die eindeutige Reihenfolge geschlossen werden kann. Das Teilproblem beschreibt sich wie folgt:

i	0	1	2	3	4	5
T_1	2	4	2	3	1	0
t_1	S	L	S^*	L	L	S^*

Tabelle 6.39: Beschreibung des Teilproblems mit String T_1

Analog zum vorherigen Bucket-Sort werden auch hier die dc-Zeichen sortiert, sodass sich folgende Berechnung ergibt:

SA_1	1	0
dc	2	5

Tabelle 6.40: dc-Sortierung

Der letzte Schritt des Algorithmus besteht darin, das Suffix-Array SA rekursiv aus den Teillösungen zu induzieren. Dazu wird das bestehende T_1 aus dem Rekursionsschritt sortiert, sodass sich die Buckets $[0, 1, 2, 2, 3, 4]$ ergeben. Folgender Algorithmus wird nun für den ersten Schritt des Induzierens angewandt[44, Kap. 4.5]:

```
1 Initialize each item of SA as −1;
2 Find the end of each bucket in SA for all the suffixes in T;
3 Scan SA_1 once from right to left;
4 if suf(T, P_1[SA_1[i]]) = LMS-suffix
5    put P_1[SA_1[i]] to the current end of the bucket for
     ↪ suf(T, P_1[SA_1[i]]) in SA and forward the buckets end one item
     ↪ to the left;
```

Abbildung 6.43: Algorithmus zur Induktion

Aus Gründen der Übersichtlichkeit, entsprechen leere Felder einem unbeschriebenen Feld, anstelle von -1.

SA_1	1	0				
Bucket	0	1	2		3	4
t	S	L	S		L	L
	5					
		4			3	1
			2	0		

Tabelle 6.41: Induzieren von SA_1

Es ergibt sich das Suffix-Array $\mathsf{SA}_1 = [5, 4, 2, 0, 3, 1]$, mit dem jetzt das finale Suffix-Array SA induziert wird:

SA_1	5	4	2	0	3	1									
Bucket	$	a								b		c			
Typ	S	L		S						L		L			
	14					(7)	(1)	(10)	(4)						
		13	12							9	3	11	6	0	5
				7	1	10	4	2	8						

Tabelle 6.42: Induzieren von SA

Von links nach rechts ergibt sich das endgültige Suffix-Array
$\mathsf{SA} = [14, 13, 12, 7, 1, 8, 2, 10, 4, 9, 3, 11, 6, 0, 5]$.

6.11.3 Optimierungsmöglichkeiten

Analog zum SAIS werden beim SADS die Typen in einem Bitvektor gespeichert. Für die dc-Zeichen wird kein eigenes Array benötigt, da hier das SA-Array genutzt werden kann. Der Algorithmus kann auf Grund des Designs unseres Frameworks nicht nur für 8-Bit, sondern auch für 16-, 32-, 48-Bittypen usw. ausgeführt werden. Im Gegensatz zum SAIS scheint eine Optimierung des SADS nicht lohnenswert, da auch die Implementierung von Yuta Mori schlechtere Ergebnisse liefert als der SAIS.

6.12 SACA-K

Ähnlich wie der SAIS-Algorithmus beruht auch der SACA-K-Algorithmus auf der Technik des **Induced Sortings**. Die wichtige Frage – und mitunter der größte Unterschied von SAIS und SACA-K – ist, wie sich mit möglichst wenig Platzverbrauch eine Sortierung der LMS-Suffixe für T herleiten lässt. Insgesamt kommt der SACA-K-Algorithmus auf eine nur von der Alphabetgröße $|\Sigma|$ abhängige Platzschranke für den zusätzlich benötigten Speicher bei weiterhin linearer Laufzeit. Im originalen Paper wird das Alphabet als K bezeichnet, daher hat der SACA-K seinen Namen. In Abbildung 6.44 ist der SACA-K-Algorithmus in Pseudocode angegeben. Im Folgenden werden die einzelnen Komponenten des Algorithmus erklärt. Da der SACA-K mit Rekursionseingaben und damit auch mit verschiedenen Strings arbeitet, benutzen wir hier für einen String T den Ausdruck $\text{suf}(\mathsf{T}, i)$ um das Suffix $\mathsf{T}[i, |\mathsf{T}|]$ anzugeben.

Insgesamt ergibt sich für den SACA-K-Algorithmus ein Laufzeitaufwand von $\mathcal{O}(n)$, da die Aufrufe von Induced Sort lineare Zeit haben und jeder konstruierte String T_1 für die Rekursionsinstanzen höchstens die Hälfte der Länge von T hat. Im Worst-Case ist die Laufzeit des SACA-K-Algorithmus also $\mathsf{T}(n) = \mathsf{T}(\lfloor n/2 \rfloor) + \mathcal{O}(n) = \mathcal{O}(n) + \mathcal{O}(\lfloor n/2 \rfloor) + \mathcal{O}(\lfloor n/4 \rfloor) + \ldots = \mathcal{O}(n)$.
Der Speicherbedarf des Algorithmus ist – neben der Eingabe T und der Ausgabe $\mathsf{SA}(\mathsf{T})$ – nur abhängig von der Größe des Alphabets, wegen der Erstellung des Arrays *bkt*. In tieferen Rekursionsstufen wird das *bkt*-Array nicht benötigt, da wir dort die Pointer der Bucketgrenzen in den Buckets selbst speichern. Die Ersparnis des Speicherplatzes im Vergleich zum SAIS kommt genau durch dieses Verhalten des Algorithmus. Für unsere Anwendungsfälle betrachten wir konstante Alphabete, daher können wir von konstantem Speicherplatzbedarf $\mathcal{O}(1)$ ausgehen. Um dies zu erzielen wird mehrmals der Speicher, welcher für $\mathsf{SA}(\mathsf{T})$ reserviert ist, in den Rekursionsinstanzen mit Zwischenberechnungen gefüllt und danach wieder überschrieben.

Bei Terminierung des Algorithmus ist das Array $\mathsf{SA}(\mathsf{T})$ in den Speicherplatz SA geschrieben. Im Folgenden werden wir zuerst beleuchten, wie sich die Induced Sort-Technik auf den tieferen Rekursionsstufen vom SAIS unterscheidet. Danach werden das Benennungsverfahren für die Symbole von T_1 und einige Techniken, welche den Platzverbrauch des Algorithmus reduzieren, erklärt.

```
def saca_k(T, SA, A, ε):
# Input text T
# Allocated space SA for suffixarray
# Alphabet A of text T
# Recursion Depth ε

# Stage 1 - First Induced Sorting
 if ε = 0
   Induced Sorting SA(T) using bucket-pointer-array bkt
   bkt stores a bucket pointer for each σ ∈ K
 else
   Induced Sorting SA(T)
   using the first/last bucket entrys as bucket counters

# Stage 2 - Renaming the LMS Substrings
 Assign individual supersymbols to distinct LMS-Substrings
 This creates string T1 with new alphabet K1

# Stage 3 - Recursion
 if all supersymbols in T1 are distinct (length(T1) = size(A1))
   No recursion needed, calculate SA(T1) directly
 else
   saca_k(T1, SA1, A1, ε+1)
   where SA1 reuses the space of SA

# Now we know the correct order of the LMS-Substrings, given by SA(T1)
# Stage 4 - Second Induced Sorting
 if ε = 0
   Induced Sorting of SA(T) from SA(T1)
   with Bucket-Pointer-Array bkt
 else
   Induced Sorting of SA(T) from SA(T1)
   with bucket counters inside the bucket
```

Abbildung 6.44: SACA-K Algorithmus [42]

6.12.1 Induced Sort im SACA-K

Wir beschäftigen uns im Folgenden mit den Details der Induced Sort-Technik. In diesem Abschnitt erläutern wir die generelle Funktionsweise von Induced Sorting auf einem String T. Zuerst wiederholen wir die normale Vorgehensweise des Induced Sortings, so wie sie im SACA-K in der obersten Stufe und auch im SAIS angewendet wird. Darauf aufbauend zeigen wir danach, wie das Induced Sorting auf den tieferen Ebenen funktioniert. Wir gehen dabei davon aus, dass die geordneten LMS-Suffixe als ein geordnetes Index-Array SA_1 vorliegen, welches die Länge $n_1 < n$ hat und in den ersten 0 bis $n_1 - 1$ Stellen von SA gespeichert ist. Es ist also $\mathsf{SA}[0, n_1 - 1] = \mathsf{SA}_1$.

Wir verwenden die Abkürzungen RTL und LTR für *Right-To-Left* und *Left-To-Right*, als Iterationsvarianten, welche ein Array der Länge n entweder mit der Iterationsfolge $n-1, n-2, \ldots, 0$ oder $0, 1, \ldots, n-1$ durchgehen.

Induced Sort für Rekursionstiefe $\varepsilon = 0$

Anders als im SAIS-Algorithmus werden im SACA-K die L- und S-Typen der Suffixe immer wieder neu bestimmt und nicht abgespeichert. Es existiert also kein Array t, welches die Typen beinhaltet und initial berechnet werden muss. Um sich die Bucket-Anfänge und -Enden zu merken, allokiert das Programm Speicherplatz für das Array *bkt* der Größe $|\Sigma|$. Da es maximal $|\Sigma|$-viele Buckets geben kann, wird *bkt* so befüllt, dass für $i \in \{0, \ldots, K-1\}$ der Wert $bkt[i]$ genau die End-Position des Buckets von Symbol $\Sigma[i]$ wiedergibt. Die SACA-K-Version von Induced Sort ergibt sich dann wie folgt:

- **1) – Initialisierungsphase**
  Initialisiere $\mathsf{SA}[n_1, n-1]$, als *null* bzw. leer.
- **2) – RTL Einordnungsphase (mit *bkt*)**
  Füge in *bkt* alle **Endpositionen** der jeweiligen Buckets ein, wie oben beschrieben. Iteriere SA_1 RTL und füge jeden Index eines LMS-Suffixes, welches mit dem Symbol $\Sigma[k]$ beginnt, in die Stelle $bkt[k]$ ein. Danach sei $bkt[k] = bkt[k] - 1$.
- **3) – LTR L-Phase (mit *bkt*)**
  Füge in *bkt* alle **Startpositionen** der jeweiligen Buckets ein. Iteriere SA LTR und für jedes nicht-leere $\mathsf{SA}[i]$, mit $j = \mathsf{SA}[i] - 1$, sodass $\mathsf{T}[j]$ ein L-Typ ist, schreibe j an die Stelle $bkt[k]$, falls $\mathsf{T}[j] = \Sigma[k]$. Danach sei $bkt[k] = bkt[k] + 1$.
- **4) – RTL S-Phase (mit *bkt*)**
  Füge in *bkt* alle **Endpositionen** der jeweiligen Buckets ein. Iteriere SA RTL und für jedes nicht-leere $\mathsf{SA}[i]$, mit $j = \mathsf{SA}[i] - 1$, sodass $\mathsf{T}[j]$ ein S-Typ ist, schreibe j an die Stelle $bkt[k]$, falls $\mathsf{T}[j] = \Sigma[k]$. Danach sei $bkt[k] = bkt[k] - 1$.

Ein Beispiel dieser Art des Induced Sortings lässt sich im SAIS-Kapitel 6.10 finden.

Induced Sort für Rekursionstiefe $\varepsilon > 0$

Wir beschäftigen uns nun mit den Feinheiten von Induced Sort für Suffixe auf den tieferen Rekursionsebenen. Beschrieben wird hier nur die Sortierung der Suffixe für Rekursionstiefe $\varepsilon = 1$, da das Vorgehen für alle anderen Rekursionstiefen analog erfolgt. Der String, den wir betrachten ist T_1, welcher aus vollkommen neuen Symbolen besteht – je ein Symbol pro verschiedenem Substring von den zu sortierenden Suffixen. Würden wir weiterhin ein Array *bkt* zum Zählen verwenden, so würden wir im Worst-Case einen Speicheraufwand von $\mathcal{O}(n)$ riskieren, da es bis zu $\mathcal{O}(\frac{n}{2}) = \mathcal{O}(n)$ viele verschiedene LMS-Substrings von T – und damit auch eine von n linear abhängige Alphabetgröße K_1 von T_1 – geben kann. Die Technik, mit der wir auch weiterhin einen konstanten Speicherverbrauch in linearer Zeit beibehalten können, ist eine spezielle Konstruktion des Strings T_1 (zu dem Konstruktionsmechanismus werden wir später kommen), sodass jeder L- bzw. S-Typ in T_1 auch ein Pointer zu dem start- bzw. dem end-Element seines Buckets ist.
Wie schon vorher erwähnt wird für SA_1 kein zusätzlicher Speicherplatz benötigt, da wir die bisherigen, nicht mehr gebrauchten Daten in SA überschreiben können. Da $n_1 \leq \lfloor n/2 \rfloor$ gilt, kann es in T_1 nur noch halb so viele zu sortierende Suffixe geben wie in T. Dies ermöglicht uns, da die in T_1 betrachteten Indizes nur noch maximal halb so groß werden können wie in T, das letzte Bit jedes Eintrages in SA für eine zusätzliche Information zu benutzen: Es ist 0, falls dieser Eintrag ein Suffix-Index ist und es ist 1, falls der Eintrag leer ist (hier mit *null* dargestellt, kann als größte negative Zahl implementiert werden) oder als bucket counter gebraucht wird. Im Folgenden werden nur die beiden kritischen Phasen 3) und 4) (genannt L- und S-Phase) von Induced Sort und ihre Implementierung für tiefere Rekursionsebenen betrachtet.

L-Phase für Rekursionstiefe $\varepsilon > 0$

Wie in der ursprünglichen Form des Algorithmus vorgesehen iterieren wir durch SA_1 LTR. Für jedes Paar $e = \mathsf{SA}[i], j = e - 1$, für das gilt

$$e \neq \mathit{null} \wedge e > 0 \wedge \mathrm{suf}(\mathsf{T}_1, j) \text{ ist L-Typ}$$

wird j in seinen entsprechenden Bucket in SA platziert. In diesem Fall ist $\mathrm{suf}(\mathsf{T}_1, j)$ genau dann ein L-Typ, wenn $\mathsf{T}_1[j] \geq \mathsf{T}_1[j+1]$ gilt. Da das Zeichen $c = \mathsf{T}_1[j]$ auf das start-Element seines eigenen Buckets zeigt, muss dieses in den Bucket der Stelle $\mathsf{SA}[c]$ eingefügt werden. Bei diesem Vorgang können die folgenden drei verschiedenen Fälle auftreten:

- Wenn $\mathsf{SA}[c] = \mathit{null}$ ist, dann ist j der erste in diesen Bucket einzufügende Index, da an dieser Stelle bisher weder ein Zähler in den Bucket noch ein Element des Buckets ist. Anstatt j jedoch genau an dieser Stelle einzufügen, prüfen wir ob auch $\mathsf{SA}[c+1] = \mathit{null}$ ist. Ist dies nicht der Fall, so beginnt an der Stelle $c+1$ bereits ein neuer Bucket und j ist das einzige Element seines Buckets (da der Bucket nur Größe 1 haben kann). Dann

schreiben wir j and die Stelle $\mathsf{SA}[c]$. Ist an der Stelle $c+1$ jedoch noch kein Element enthalten, so benutzen wir $\mathsf{SA}[c]$ vorerst als einen Bucketzähler – dies ist eine negative Zahl, welches die Zähler von den echten Elementen unterscheidet, welche angibt wie viele Elemente bereits im zugehörigen Bucket stehen. Außerdem setzen wir $\mathsf{SA}[c] = -1$ und $\mathsf{SA}[c+1] = j$. Im weiteren Verlauf des Algorithmus wird $\mathsf{SA}[c]$ dann entweder weiter dekrementiert, wenn mehr Elemente in den Bucket hinzukommen. Sobald der Bucket mit allen seinen Elementen befüllt wurde, werden alle Elemente nach links geshiftet und überschreiben damit den Zähler.

- Wenn $0 > \mathsf{SA}[c] \neq \mathit{null}$ ist, dann wird $\mathsf{SA}[c]$ für einen Zähler benutzt und enthält die momentane (negative) Anzahl der Elemente im Bucket. Es sei $p = c + |\mathsf{SA}[c]| + 1$, dann ist p die Position an die der Index j einsortiert wird, falls $\mathsf{SA}[p] = \mathit{null}$ ist und $\mathsf{SA}[c] = \mathsf{SA}[c]-1$. Sollte der Wert in $\mathsf{SA}[p]$ jedoch nicht *null* sein, so haben wir das Ende unseres Buckets erreicht und würden mit dem Schreiben von j in $\mathsf{SA}[p]$ über die Grenzen unseres Buckets hinausgehen. In diesem Fall ist j der letzte in den Bucket einzusortierende Index und wir führen für den gesamten Bucket einen links-shift aus, so dass j in $\mathsf{SA}[p-1]$ geschrieben werden kann und alle anderen Elemente einen Eintrag nach links wandern, bis der Zähler in $\mathsf{SA}[c]$ von $\mathsf{SA}[c+1]$ überschrieben wurde. Danach ist dieser Bucket vollständig.

- Wenn $0 < \mathsf{SA}[c] \neq \mathit{null}$ ist, so wird $\mathsf{SA}[c]$ weder als Indikator gebraucht noch ist es leer – in diesem Fall benutzt der links anliegende Bucket also $\mathsf{SA}[c]$ für seine eigene Auslagerung. Daraus lässt sich schließen, dass der links anliegende Bucket bereits voll ist und wir können (wie oben) für diesen Bucket einen links-shift ausführen, bis zu dem ersten nicht leeren, negativen Eintrag. Dann verfahren wir genauso wie im Fall $\mathsf{SA}[c] = \mathit{null}$ weiter für den Bucket von $\mathsf{SA}[c]$.

Der Zeitaufwand ist hierbei linear, da das Einfügen jedes Index in konstanter Zeit passiert und jeder eingefügte Index maximal einmal geshiftet wird. Da maximal ein Index pro Iteration eingefügt wird, wird die Laufzeit von der Länge von SA_1 dominiert, welche $\mathcal{O}(n_1)$ ist.

S-Phase für Rekursionstiefe $\varepsilon > 0$

Für diese Phase der Sortierung kommt uns die Eigenschaft von T_1 zuhilfe, dass für jeden S-Typ $\mathrm{suf}(\mathsf{T}_1, i)$ der Eintrag $\mathsf{SA}[i]$ das letzte Element des Buckets von $\mathsf{T}_1[i]$ ist. Wie im Ursprungsalgorithmus vorgesehen iterieren wir RTL durch SA_1. Für jedes Element $\mathsf{SA}[i] \neq \mathit{null}$ und $\mathsf{SA}[i] > 0$ mit $j = \mathsf{SA}[i]-1$, sodass $\mathrm{suf}(\mathsf{T}_1, j)$ ein S-Typ ist (dies ist der Fall, wenn $\mathsf{T}_1[j] > \mathsf{T}_1[j+1]$ gilt oder $\mathsf{T}[j] = \mathsf{T}[j+1]$ und $\mathsf{T}[j] > i$ ist), wird j in seinen entsprechenden Bucket in SA ähnlich wie oben platziert und zwar für $c = \mathsf{T}[j]$ wie folgt:

- Wenn $\mathsf{SA}[c] = \mathit{null}$ ist, dann ist j der erste in diesen Bucket einzufügende Index. Wir prüfen ob $\mathsf{SA}[c-1]$ leer ist. Falls ja, dann setzen wir $\mathsf{SA}[c-1] = j$

und benutzen $\mathsf{SA}[c]$ als einen Zähler indem wir initial $\mathsf{SA}[c] = -1$ setzen. Falls nein, so umfasst der S-Teil des Buckets nur ein Element, nämlich j, und wir setzen $\mathsf{SA}[c] = j$.

- Wenn $0 > \mathsf{SA}[c] \neq \mathit{null}$ ist, dann wird $\mathsf{SA}[c]$ für einen Zähler benutzt und enthält die momentane (negative) Anzahl der Elemente im Bucket. Falls für $p = c + \mathsf{SA}[c] - 1$ der Wert von $\mathsf{SA}[p]$ leer ist, so ist $\mathsf{SA}[p] = j$ und wir setzen $\mathsf{SA}[c] = \mathsf{SA}[c] - 1$. Falls $\mathsf{SA}[p]$ nicht leer ist, so ist $\mathsf{SA}[p-1]$ das Ende des S-Teils dieses Buckets und wir führen einen rechts-shift aller Elemente im Bereich $\mathsf{SA}[c, p]$ aus, bis $\mathsf{SA}[c]$ durch $\mathsf{SA}[c-1]$ überschrieben wurde und setzen dann $\mathsf{SA}[p-1] = j$.
- Wenn $0 < \mathsf{SA}[c] \neq \mathit{null}$ ist, so wird $\mathsf{SA}[c]$ weder als Indikator gebraucht noch ist es leer – in diesem Fall benutzt der rechts anliegende Bucket also $\mathsf{SA}[c]$ für seine eigene Auslagerung. Der rechts anliegende Bucket ist also bereits voll und wir können für diesen rechten Bucket einen rechts-shift ausführen, bis zu dem ersten nicht leeren, negativen Eintrag (dem Zähler dieses Buckets). Dann verfahren wir für den Bucket von $\mathsf{SA}[c]$ genauso wie im Fall $\mathsf{SA}[c] = \mathit{null}$ weiter.

Der Zeitaufwand ist $\mathcal{O}(n)$, analog zu dem der L-Phase.

6.12.2 Benennungsverfahren der LMS-Substrings

In diesem Abschnitt werden wir erfahren, wie sich die neuen Symbole des Alphabets Σ_1 für die Strings T_1 ergeben, sodass die oben vorausgesetzte Eigenschaft gilt, dass jeder L- bzw. S-Typ in T_1 auch ein Pointer zu dem start- bzw. dem end-Element seines Buckets ist. Es werden die folgenden Definitionen gebraucht:

Definition 19. ***S-Rang und SE-Rang***
Für einen String T *der Länge* n *und einen Index* $i \in \{0, \ldots, n-1\}$ *ist der S-Rang bzw. der SE-Rang von* $\mathsf{T}[i]$ *die Anzahl von* $\mathsf{T}[j]$ *für* $j \neq i$ *mit* $\mathsf{T}[j] < \mathsf{T}[i]$ *bzw.* $\mathsf{T}[j] \leq \mathsf{T}[i]$.

Um den String T_1 aus den geordneten LMS-Substrings in SA von T zu erzeugen, wird jedem LMS-Substring wie folgt sein S- bzw. SE-Rang zugewiesen:

- Iteriere über SA_1 LTR um jeden LMS-Substring den start-Index seines Buckets zuzuweisen in welchen er sortiert wurde.
- Iteriere über den entstandenen String RTL und ersetze jedes Symbol mit S-Typ durch den end-Index des Buckets auf den es zeigt.

Um den ersten Schritt ausführen zu können, muss der start-Index der jeweiligen Buckets korrekt identifiziert werden, indem je zwei benachbarte LMS-Substrings in T miteinander verglichen werden. Um das Ende eines LMS-Substring feststellen zu können, wird der Substring durchlaufen bis zum ersten mal ein L-Typ mit einem darauf folgenden S-Typ erkannt wird. Damit ist das Benennungsverfahren mit einer Laufzeit von $\mathcal{O}(n)$ ausführbar.

6.13 pSAIS

Der pSAIS – kurz für **parallel SAIS** – ist ein SACA, der sich aus dem SAIS entwickelt hat und mithilfe von parallel laufenden Threads ein Suffixarray berechnet [32]. Die grundlegende Idee des pSAIS ist es, das SA für das Induced Sorting zuerst in mehrere Blöcke aufzuteilen mit Blockgröße β (diese wurde in unserer Implementierung fest auf 10 MiB gesetzt). Für diese Blöcke kann die Methode des Induced Sortings dann teilweise parallelisiert werden. Das Framework des pSAIS entspricht dabei dem des SAIS bis auf die hier aufgeführten Veränderungen und Neuerungen.
Es werden Buffer-Listen r und w für das Lesen aus dem Text und das Schreiben in das SA eingeführt, welche die selbe Länge β haben und Tupel der Art $\langle chr, pos\rangle$ bzw. $\langle idx, pos\rangle$ enthalten. In dem Lese-Buffer r gibt chr ein Zeichen des Textes und pos seine Position im Text an. In dem Schreib-Buffer w ist pos eine Position im Text und idx ist die Position des SAs, an welcher pos eingeordnet werden soll. Für jeden Block B werden dann die folgenden drei Phasen nacheinander durchgeführt:

- **Preparing:** Hier werden alle Paare $\langle chr, pos\rangle$ in einen Lese-Buffer r hineingeschrieben, welche die Bedingung erfüllen, dass $pos = B[j] - 1$ ist, für eine Position j von Block B. Je nachdem um welches Induce Sorting es sich handelt muss $\mathsf{T}[pos]$ außerdem ein L- bzw. S-Typ sein. Das gesamte Preparing wird hierbei **parallel** für jede Stelle j des Blocks B aufgerufen. Das Preparing lädt also bestimmte Zeichen in den Lese-Buffer, welche dann in dem folgenden Schritt effizient und ohne direkt auf den Text zugreifen zu müssen herausgelesen werden können.

- **Inducing:** Das Induzieren folgt auf das Preparing des Blockes B. Es erfolgt **nicht parallel** und benutzt den vorher beschriebenen Read-Buffer r. Block B wird durchlaufen und für jede Position i in B wird wie beim Induced Sorting überprüft, ob $chr = B[i] - 1$ ein L- bzw. ein S-Typ ist. Falls dem so ist, wird die in $r[i].chr$ gespeicherte Information benutzt um direkt das passende Zeichen zu dieser Position zu erhalten. Ist $r[i]$ leer – dies kann z.B. passieren wenn $B[i]-1$ gar nicht mehr in Block B hineingehört – wird stattdessen das Zeichen aus dem Text gelesen, also $\mathsf{T}[B[i]-1]$. Danach wird zu dem gelesenen Zeichen die passende Position pos für chr im SA gesucht und das Tupel $\langle chr, pos\rangle$ wird in einen Schreib-Buffer w eingefügt, falls pos in den Bereich des momentanen Blocks B oder den des darauffolgenden Blocks B' fällt. Fällt pos nicht in den Bereich einer der beiden Blöcke, wird direkt in das SA geschrieben: $SA[pos] = chr$.

- **Updating:** Im Updating wird **parallel** der Schreib-Buffer w durchlaufen und für jedes Paar $\langle chr, pos\rangle$ in w wird $SA[pos] = chr$ gesetzt. Das Updating hilft, genau wie das Preparing, die Zugriffe auf das SA bzw. den Eingabetext effizienter zu machen, da andernfalls bei direkten Zugriffen viele Cache Misses entstehen können.

Diese drei Stufen, welche wir das **Pipelined Inducing** nennen, ersetzen vollständig das Induzieren im pSAIS. Wird SA in die Blöcke $B_0, \ldots, B_m - 1$ eingeteilt, so müssen für das Induzieren der L-Typen die Blöcke in der angegebenen Reihenfolge durchlaufen werden, für das Induzieren der S-Typen müssen sie jedoch in umgekehrter Reihenfolge durchlaufen werden. Wir stellen dafür im Folgenden in 6.45 nur das Pipelined Inducing der L-Typen dar, da sich das Pipelined Inducing der S-Typen daraus herleiten lässt.

6.13.1 Verschränktes Induzieren

Eine weitere Technik des pSAIS, um viele Berechnungsschritte parallel machen zu können, ist das gleichzeitige Ausführen einiger Schritte des Pipelined Inducings für aufeinander folgende Blöcke. Um dies möglich zu machen, werden je zwei anstatt nur je ein Lese- und Schreibbuffer benutzt. Diese Buffer nennen wir r_1, r_2 bzw. w_1, w_2. Ein Pipelined-Inducing Vorgang für einen festen Block B_i verwendet dabei immer die selben Lese- und Schreibbuffer w_j, r_j. Der darauffolgende Block B_{i+1} verwendet jedoch die Buffer $w_{\bar{j}}, r_{\bar{j}}$, mit $\bar{j} = 3 - j$, also genau die vom Block B_i nicht benutzten Buffer. So können die folgenden Teile des Pipelined Inducings parallel berechnet werden:

- Während das Inducing des Blockes B_i mit den Buffern r_j, w_j ausgeführt wird, kann das Preparing des Blockes B_{i+1} mit dem Buffer $r_{\bar{j}}$ ausgeführt werden.
- Während das Updating des Blockes B_i mit dem Buffer w_j ausgeführt wird, kann das Inducing des Blockes B_{i+1} mit den Buffern $r_{\bar{j}}, w_{\bar{j}}$ ausgeführt werden.

Eine graphische Darstellung dieses Prozesses findet sich unter Abbildung 6.46.

6.13.2 Parallele Substring-Benennung

In [32] wird eine Methode gezeigt um das Benennen der LMS-Substrings vor dem Rekursionsaufruf zu parallelisieren. Dafür werden die geordneten Substrings in gleiche Blöcke zerteilt und für jeden Block wird ein Differenzen-Bitvektor berechnet. Für jeden Substring in dem Block, der verschieden von seinem nachfolgenden Substring ist, ist das zugeordnete Bit 1 in dem Vektor, andernfalls ist es 0. Aus den entstehenden Vektoren kann die Anzahl verschiedener Namen in diesem Block berechnet werden. Damit können die neuen Namen der jeweils ersten Substrings aller Blöcke sequentiell berechnet werden. Danach können parallel für jeden Block, ausgehend von dem bereits umbenannten ersten Substring des Blocks, die Namen der anderen Substrings des Blocks berechnet werden.

Diese Methode wurde in unserer Version des pSAIS aus Zeitmangel nicht implementiert. Die dort vorzufindende Art der Umbenennung erfolgt wie beim SAIS sequentiell.

```
1  def pipelined_l_inducing(T, k, t, BA, SA):
2  # Input text T
3  # Blocknumber k
4  # Type Array t
5  # Bucket Counter Array BA
6  # Suffixarray SA with LMS suffixes already sorted in
7
8  # Parallel: Preparing
9  parallel for i = 0 to β − 1 do
10    Initialize r and w as empty
11    j = k * β + i                  # Global current Position in T
12    if (SA[j] is non-empty and (pos = SA[j]-1) ≤ 0 and t[pos] is L-Type)
13     chr = T[pos]
14     Insert ⟨chr, pos⟩ into r[i]
15
16 # Sequential: Inducing
17 for i = 0 to β − 1 do
18  if (B_k[i] is non-empty and (pos = B_k[i] -1)  ≤ 0 and t[pos] is L-Type)
19   if (r[i].chr is empty)
20    chr = T[pos]
21   else
22    chr = r[i].chr # preceding character was already stored in read buffer
23 idx = BA[chr]++
24 if(idx is in Block B_k or B_{k+1})
25  Write SA[idx] = pos
26 else
27  Insert ⟨idx, pos⟩ into w[i]
28
29 # Parallel: Updating
30 parallel for i = 0 to β − 1 do
31  if w[i] is non-empty then
32   SA[w[i].idx] = w[i].pos
```

Abbildung 6.45: Pipelined L-Inducing nach [32]

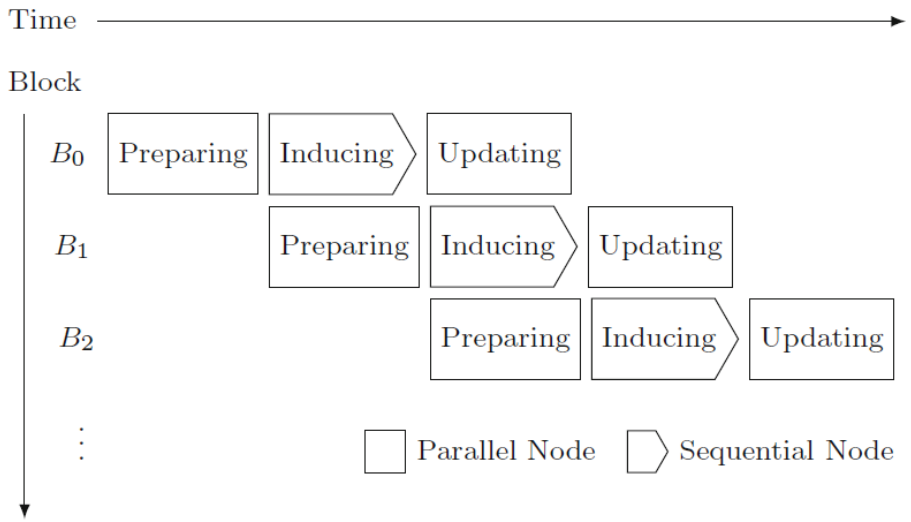


Abbildung 6.46: Graphische Darstellung des verschränkten Induzierens in [32]

6.13.3 Paralleles Klassifizieren

Um den SAIS auch beim initialen Klassifizieren zu beschleunigen, wird das Zuordnen von L- und S-Typen parallelisiert. Dabei wird der Text in Blöcke aufgeteilt, die selbständig von ihren Threads abgearbeitet werden. Um speichereffizient arbeiten zu können, werden die Typen in einen Bitvektor gespeichert. Das bedeutet die Threadgrenzen der Blöcke müssen immer einer Zweierpotenz entsprechen. Dies ist nötig, da die Datenstrukturen intern immer auf Bytes arbeiten und es beim Manipulieren der Bytes über Threadgrenzen hinweg zu *Race-Conditions* kommen kann, das Ergebnis der Berechnung also davon abhängt, welcher Thread zuerst das jeweilige Byte bearbeitet. Das parallele Klassifizieren funktioniert in zwei Durchgängen.

Im ersten Durchgang klassifiziert jeder Thread seinen Block soweit es geht. Dabei ist es jedem Thread erlaubt, ein Zeichen über seinen eigenen Block zu schauen. Es ist ersichtlich, dass der letzte Thread, der den Block mit dem Sentinel bearbeitet, immer dazu in der Lage ist, seinen gesamten Block zu klassifizieren. Bei allen anderen Blöcken ist es möglich, dass sie kein oder nur einen Teil der Zeichen klassifizeren können. Jeder Thread speichert also zusätzlich bis zu welchem Zeichen er klassifizieren konnte. Falls er klassifizieren konnte, speichert er sich zusätzlich noch den klassifizierten Typ an der Threadgrenze, der für den zweiten Durchgang benötigt wird. Exemplarisch für den ersten Durchgang folgt ein Beispiel. Der Einfachheit halber wurden die Threadgrenzen nach jeweils 4 Zeichen gesetzt und nicht byteweise:

0	1	2	3
aaaa	aaaa	aaba	acc$
xxxx	xxxx	SSLx	SLLS
x	x	**S**	**S**

Dadurch, dass das Sentinel das lexikographisch kleinstmögliche Zeichen ist und im pSAIS nur einmalig vorkommt, kann der 3. Block in seiner Gesamtheit klassifiziert werden. Der zweite Block versucht das letzte a zu klassifizieren. Dies ist jedoch nicht möglich, da das nächste Zeichen auch ein a ist. An dieser Stelle ist es noch nicht möglich, auf dessen Typ zuzugreifen und diesen zu übernehmen, da alle Threads gleichzeitig arbeiten und zu diesem Zeitpunkt nicht gewährleistet ist, dass der Typ schon geschrieben wurde. An den jeweiligen Stellen, die nicht klassifizert werden konnten steht ein x. Die Blöcke 0 und 1 konnten nichts klassifizieren, da auch mit dem nächsten Zeichen ihres Blocks keine Informationen über den Vorgängertypen ermitteln konnten. Alle Threads speichern zudem den ermittelten Typ an ihren Threadgrenzen. Da nur Block 2 und 3 klassifizieren konnten, steht nur bei ihnen kein x.

Im zweiten und finalen Durchgang des Klassifizierens werden die Informationen der benachbarten Blöcke genutzt, um den gesamten Text zu klassifizieren, der noch nicht klassifiziert werden konnte:

0	1	2	3
aaaa	aaaa	aaba	acc$
SSSS	**SSSS**	SSL**S**	SLLS

Im ersten Durchgang fehlte im Block 2 noch die Information, welchem Typ das a entsprach. Diese Information ist nun vorhanden und das letzte Zeichen des Blocks kann klassifiziert werden. Das fettgedruckte rote S konnte also durch die Information aus dem dritten Block klassifiziert werden. Analog verhält es sich mit den ersten beiden Blöcken, deren Zeichen alle S-Typen sind. Hier wird die Information aus dem zweiten Block auf die vorherigen beiden Blöcke übertragen.

6.14 GSACA

6.14.1 Einleitung

Der Algorithmus GSACA verfolgt bei der Konstruktion des Suffix-Arrays einen neuen Ansatz, um dieses in linearer Zeit zu erstellen, ohne rekursiv zu arbeiten. Vorgestellt wurde dieser Algorithmus das erste mal im Paper *Linear-time Suffix Sorting - A New Approach for Suffix Array Construction* von Uwe Baier [7]. Eine vollständige Implementierung des Algorithmus lässt sich auf seiner GitHub-Seite [6] finden.

Zunächst wird in Kapitel 6.14.2 das allgemeine Vorgehen des Algorithmus informell beschrieben. Kapitel 6.14.3 wendet ihn dann beispielhaft an dem Wort Banane an. Danach wird in Kapitel 6.14.4 der Algorithmus detailliert erklärt. In Kapitel 6.14.5 werden anschließend weitere Details der Implementierung besprochen. Darauf folgt das Kapitel 6.14.6, in dem GSACA auf die Eingabe *caabaccaabacaa* angewendet wird, wie es zuvor auch schon bei anderen Algorithmen der Fall war. Zum Schluss werden in Kapitel 6.14.7 mögliche Änderungen und Optimierungen besprochen.

6.14.2 Das Vorgehen

Bevor der konkrete Algorithmus präsentiert wird, wird in diesem Kapitel das allgemeine Vorgehen beschrieben. Das Suffix-Array für ein gegebenes Wort wird in zwei Phasen konstruiert. Folgende Definitionen werden hierfür benötigt:

Definition 20. $\widehat{i}$
Für einen String S der Länge n sei i ein Index zwischen 1 und n. Dann bezeichnet $\widehat{i}$ *den Index des nächsten lexikografisch kleineren Suffix.*

Definition 21. ***Gruppenkontext***
Sei S ein String. Dann ist der Substring $S[i..\widehat{i}]$ *ein Gruppenkontext, also der Präfix eines Suffixes bis zum Beginn des nachfolgenden lexikografisch kleineren Suffixes.*

Definition 22. ***Gruppe***
Sei S ein String und C ein Gruppenkontext. Dann bezeichnet die zu C gehörende Gruppe die Menge der Startindices aller Vorkommen von C im String S.

Für das Wort Banane haben wir beispielsweise die beiden Suffixe *Banane* und *anane*. Da der Gruppenkontext vom Suffix *Banane* der Präfix bis zum nächsten lexikografisch kleineren Suffix ist, umfasst der Gruppenkontext von *Banane* nur den Buchstaben *B*. Die Gruppe des Gruppenkontextes *B* umfasst die Startindices aller Vorkommen dieses Gruppenkontextes. In diesem Fall besteht diese Gruppe also nur aus dem Index 1.

Die initialen Gruppenkontexte sind die einzelnen Buchstaben des Wortes in lexikografischer Reihenfolge, beginnend mit dem Terminalsymbol $. Hieraus ergeben sich die anfänglichen Gruppen als Mengen der Indices der Vorkommen

des entsprechenden Buchstabens im Wort. Anschließend bearbeitet der Algorithmus die Gruppen in lexikografisch absteigender Reihenfolge. Dazu wird für jeden Buchstaben der Gruppe das direkte vorherige Zeichen im Wort betrachtet. Die Gruppe, zu der dieses vorherige Zeichen gehört, wird als Vorgängergruppe bezeichnet. Der Gruppenkontext dieser Vorgängergruppe wird nun um den Kontext der aktuell vom Algorithmus bearbeiteten Gruppe erweitert. Dies bedeutet, dass der Gruppenkontext der Vorgängergruppe ersetzt wird durch die Konkatenation aus dem vorherigen Kontext der Vorgängergruppe und dem Kontext der aktuellen Gruppe. Zusätzlich wird ein sogenannten prev pointer gespeichert. Dies ist ein Zeiger von einem Buchstaben zu dem Beginn des Gruppenkontext des direkt vorherigen Zeichens beginnt. Es kann vorkommen, dass nicht alle Buchstaben der Vorgängergruppe im Wort direkt vor den Buchstaben der aktuell durch den Algorithmus bearbeiteten Gruppe liegen. Falls nur ein Teil von der Vorgängergruppe getroffen wurde, findet nur bei dem getroffenen Teil der Vorgängergruppe eine Kontexterweiterung statt und sie wird in zwei Gruppen aufgeteilt. Die erste Gruppe ist die Teilgruppe mit erweitertem Kontext, sie wird als Nachfolger der Teilgruppe ohne Kontexterweiterung hinzugefügt. Durch dieses Vorgehen werden iterativ alle Gruppen bearbeitet, deren Kontexte erweitert und gegebenenfalls in neue Gruppen aufgeteilt.

In der zweiten Phase wird die zuvor in Phase 1 berechnete Gruppenstruktur genutzt, um das finale Suffix-Array zu erstellen. Hierzu werden die Gruppen durchlaufen, diesmal allerdings in lexikografisch aufsteigender Reihenfolge. Zu Beginn ist SA leer für alle Indices, bis auf die erste Stelle, an der der Index des Terminationssymbols $ steht. Anschließend wird über SA iteriert. Dazu wird in jedem Durchlauf das Zeichen des originalen Wortes an der Position, die in SA gespeichert ist, gesucht und das vorherige Zeichen betrachtet. Dann wird die Kette der prev pointer ausgehend von diesem Zeichen durchlaufen, bis diese leer ist und die Indices der gefundenen Zeichen in SA gespeichert.

6.14.3 Suffix Array Konstruktion am Beispiel Banane

Nachdem im vorherigen Kapitel das Vorgehen beschrieben wurde, wird in diesem Kapitel das Suffix Array für das kurze Beispielwort Banane konstruiert. Dazu wird der Algorithmus schrittweise durchlaufen und die jeweiligen Zustände in tabellarischer Form visualisiert.

Phase 1:

In der ersten Phase werden die Gruppen und Gruppenkontexte für das Wort Banane erstellt. Dies geschieht in insgesamt vier Iterationen.

Iteration 1:
Schritt 1: Das Wort Banane besteht aus vier verschiedenen Buchstaben (a, b, e, n), zusammen mit dem Terminalsymbol $\$$ sind somit am Beginn der ersten Iteration fünf Gruppen vorhanden. Die Tabelle 6.43 zeigt den Ausgangszustand. In ihr sind in der oberen Hälfte die initialen Gruppen und Gruppenkontexte dargestellt und in der unteren Hälfte das Wort zusammen mit den Indices der Buchstaben. Die initialen Gruppenkontexte sind die einzelnen Buchstaben. Diese bestimmen auch die anfänglichen Gruppen, welche die Indices der Buchstaben im Wort beinhalten. Der Algorithmus beginnt, indem der lexikographisch letzte Gruppenkontext betrachtet wird. In diesem Fall ist das der Buchstabe n, welcher an den Indices 3 und 5 in dem Wort vorkommt. Die entsprechenden Zellen der Tabelle sind grün hervorgehoben.
Schritt 2: Dann werden die Vorkommen der aktuell betrachteten Gruppe im Wort untersucht. In dieser ersten Iteration ist es das Vorkommen von dem Zeichen n an den Positionen 3 und 5.
Schritt 3: Danach werden die direkten Vorgänger im Wort untersucht. In diesem Schritt ist das der Buchstabe a an den Positionen 2 und 4. Diese Indices sind zusammen mit den Buchstaben an den entsprechenden Positionen in der Tabelle rot hinterlegt.
Schritt 4: Im letzten Schritt dieser Iteration wird der Gruppenkontext der im vorherigen Schritt markierten Buchstaben betrachtet. Dieser wird um den Kontext der Gruppe n erweitert. Da alle Vorkommen der Gruppe a Vorgänger von der Gruppe n waren, findet diese Kontexterweiterung bei allen Elementen statt. Somit wird aus der Gruppe a die Gruppe an. Das finale Ergebnis zeigt die Änderungen vor gelben Hintergrund an.

Iteration 2:
Schritt 1: In der zweiten Iteration, deren Schritte in Tabelle 6.44 verdeutlicht werden, wird die nächste Gruppe bezüglich der lexikographisch absteigenden Ordnung betrachtet. Dies ist die Gruppe mit Kontext e.
Schritt 2: Der Buchstabe e kommt nur an Index 6 im Wort vor.
Schritt 3: Für diesen Index wird die Vorgängergruppe gesucht. Diese ist die Gruppe an, die im vorherigen Schritt gebildet wurde.
Schritt 4: Anders als in der vorherigen Iteration werden in diesem Durchlauf nicht alle Vorkommen der Vorgängergruppe getroffen. Damit muss diese Gruppe in zwei neue Gruppen aufgeteilt werden. Der erste Teil der Gruppe bleibt an und der zweite Teil wird um den Kontext von e erweitert. Da die Gruppe mit dem erweiterten Kontext lexikographisch größer ist, wird sie zum neuen Nachfolger der Teilgruppe ohne Kontexterweiterung.

Iteration 3:
Schritt 1: Als nächstes wird die Gruppe *b* betrachtet. Die Zwischenergebnisse dieser Iteration sind in Tabelle 6.45 zu sehen.
Schritt 2: Diese Gruppe hat keinen Vorgänger im betrachteten Wort. Daher endet diese Iteration ohne Änderungen an den Kontextgruppen.

Iteration 4:
Schritt 1: Es wird die nächste Gruppe betrachtet. Dies ist die Gruppe mit Kontext *ane*.
Schritt 2: Diese Gruppe umfasst die Indices 4 bis 6 im Wort Banane.
Schritt 3: Der Vorgänger dieser Gruppe ist die Gruppe *an*.
Schritt 4: Wie in der ersten Iteration wird die gesamte Vorgängergruppe getroffen. Diese wird um den Kontext der Gruppe *ane* erweitert. Das Ergebnis dieser Iteration und der gesamten Phase 1 ist in Tabelle 6.46 gezeigt. Am Ende dieser Phase hat man somit 6 Gruppenkontexte: *$*, *anane*, *ane*, *b*, *e* und *n*. Zusätzlich wurden für alle Buchstaben des Wortes die prev pointer erzeugt und gespeichert. Die entstandenen Pointer-Ketten lassen sich in Abbildung 6.47 sehen. Auf diesem Ergebnis aufbauend wird in Phase 2 das finale Suffix-Array berechnet werden.

Phase 1 - Iteration 1 - Schritt 1

Kontext	$	a		b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 1 - Schritt 2

Kontext	$	a		b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 1 - Schritt 3

Kontext	$	a		b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 1 - Schritt 4

Kontext	$	a		b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 1 - Ergebnis

Kontext	$	an		b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Tabelle 6.43: Konstruktion des Suffix-Arrays für das Wort Banane: Phase 1, Iteration 1

Phase 1 - Iteration 2 - Schritt 2

Kontext	$	an		b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 2 - Schritt 2

Kontext	$	an		b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 2 - Schritt 3

Kontext	$	an		b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 2 - Schritt 4

Kontext	$	an		b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 2 - Schritt 4

Kontext	$	an	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Tabelle 6.44: Konstruktion des Suffix-Arrays für das Wort Banane: Phase 1, Iteration 2

Phase 1 - Iteration 3 - Schritt 1

Kontext	$	an	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 3 - Schritt 1

Kontext	$	an	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 3 - Schritt 1

Kontext	$	an	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Tabelle 6.45: Konstruktion des Suffix-Arrays für das Wort Banane: Phase 1, Iteration 3

Phase 1 - Iteration 4 - Schritt 1

Kontext	$	an	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 4 - Schritt 2

Kontext	$	an	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 4 - Schritt 3

Kontext	$	an	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 4 - Schritt 4

Kontext	$	an	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 1 - Iteration 4 - Ergebnis

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Tabelle 6.46: Konstruktion des Suffix-Arrays für das Wort Banane: Phase 1, Iteration 4

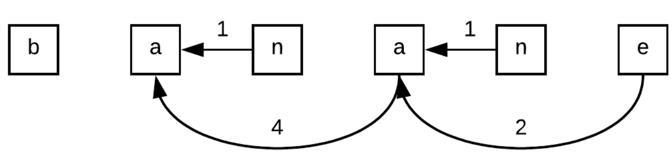


Abbildung 6.47: Konstruktion des Suffix-Arrays für das Wort Banane: prev pointer. Die Zahlen geben die Nummer der Iteration an, in welcher der prev pointer berechnet wurde.

Phase 2:

In der zweiten Phase wird das Suffix-Array mithilfe der zuvor erstellten Gruppen und Gruppenkontexte sowie der prev pointer konstruiert. In diesem Beispiel werden hierfür 5 Iterationen durchlaufen.

Iteration 1:
Schritt 1: Die Abläufe der Phase 2 können an einer ähnlichen Tabelle visualisiert werden, wie dies bei Phase 1 der Fall war. Die untere Hälfte der Tabelle enthält, genau wie die in Phase 1, das betrachtete Wort Banane und die Indices der Buchstaben. Die obere Hälfte besteht aus den Gruppen und Gruppenkontexten, wie sie aus Phase 1 übernommen wurden und aus der Liste SA. Diese enthält den Index des ersten Buchstaben des Suffixes. Wie zuvor beschrieben, werden in dieser Phase die Werte von SA von links nach rechts durchlaufen, beginnend mit der ersten Gruppe, in diesem Beispiel also mit der Gruppe *$* an Index 7. Die Tabelle und die einzelnen Schritte dieser Iteration sind in Tabelle 6.47 gezeigt. Die aktuell betrachtete Gruppe ist durch grünen Hintergrund gekennzeichnet.
Schritt 2: Es wird das Zeichen *$* an Position 7 im Wort Banane betrachtet. Auch die aktuell bearbeitete Position ist durch grüne Farbe hervorgehoben.
Schritt 3: Um das nächste Element von SA zu bestimmen wird der direkte Vorgänger dieses Zeichens untersucht. Dabei bezieht sich diese Iteration also auf das Element 6, welches in der Zelle mit blauem Hintergrund zu sehen ist.
Schritt 4: Von diesem Element ausgehend werden die prev pointer aus Phase 1 durchlaufen bis das letzte Element dieser Liste gefunden ist. Das Vorgängerelement von dem Zeichen *e* an Index 6, also das Zeichen, auf das der prev pointer von *e* zeigt, ist das Zeichen *a* an Index 4. Dieses wiederum hat einen prev pointer auf das Zeichen *a* an Index 2. Die betrachteten Indices, die in der Tabelle rot hinterlegt sind, werden in die Liste SA übertragen. Somit enthält diese am Ende der ersten Iteration drei weitere Indices, welche durch gelben Hintergrund hervorgehoben sind.

Iteration 2:
Schritt 1: In diesem Durchlauf wird das zweite Element in SA betrachtet, dies ist Index 2.
Schritt 2: Es wird dieser Index im Wort betrachtet.
Schritt 3: Der Vorgänger des Buchstaben *a* an Index 2 ist das Zeichen *b* an Index 1. Da *b* kein vorheriges Element hat, der prev pointer also auf kein Element zeigt, endet diese Iteration hier. Der Index von Zeichen *b* wird in SA eingetragen. Das Ergebnis lässt sich in Tabelle 6.48 sehen.

Iteration 3:
Schritt 1: Als nächstes wird das dritte Element von SA untersucht. Aus der Tabelle 6.49 lässt sich erkennen, dass dies das Element an Index 4 ist.
Schritt 2: Das Zeichen an dem betrachteten Index ist *a*.
Schritt 3: Der Vorgänger von *a* an Index 4 ist *n* an Index 3.
Schritt 4: Der prev pointer von *n* an Index 3 zeigt auf das *a* an Index 2. Der

Index 2 ist bereits in SA enthalten, Index 3 aber noch nicht, daher wird dieser in die Liste aufgenommen.

Iteration 4:
Wie sich in der Tabellen 6.50 sehen lässt, wird das Zeichen an Index 1 betrachtet. Dies ist das Zeichen b, welches kein Element hat, auf das sein prev pointer zeigt. Somit endet diese Iteration ohne Änderung der Tabelle.

Iteration 5:
Schritt 1: In dieser Iteration wird das Element an Index 6 untersucht.
Schritt 2: Das Zeichen an Index 6 im Wort ist e.
Schritt 3: Es wird der Vorgänger von diesem Zeichen betrachtet. Die Tabelle zu dieser Iteration, Tabelle 6.51, zeigt, dass dies das Zeichen n an Index 5 ist.
Schritt 4: Dieses Zeichen hat einen Zeiger auf das Vorgängerelement a an Index 4. Wie in einer vorherigen Iteration hat dieses a einen Zeiger auf das Zeichen a an Index 2. Da die Zeichen an Index 4 und 2 bereits in der Liste SA enthalten sind, wird in diesem Schritt nur der Index 5 zu dieser Liste hinzugefügt. Somit ist die Liste vollständig gefüllt.

In diesem einfachen und kurzen Beispiel weicht die Liste der Gruppen, welche das Ergebnis von Phase 2 ist, nicht von der Liste SA ab. Dies ist jedoch nicht immer der Fall, vor allem in längeren und komplexeren Beispielen unterscheiden sich diese teilweise. Diese Unterschiede werden bei der Berechnung des Suffix-Arrays für das Wort *caabaccaabacaa* in Kapitel 6.14.6 deutlich. Je mehr Gruppen mit mehr als einem Index existieren, umso häufiger kann die Reihenfolge der Indices innerhalb dieser Gruppe gewechselt werden. Im Beispiel zum Wort Banane gab es jedoch nur eine Gruppe mit mehr als einem Index. In dieser Gruppe mit dem Buchstaben n fand keine Veränderung durch diese Schritte statt.

Phase 2 - Iteration 1 - Schritt 1

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7						
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 1 - Schritt 2

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7						
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 1 - Schritt 3

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7						
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 1 - Schritt 4

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7						
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 1 - Ergebnis

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4		6		
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Tabelle 6.47: Konstruktion des Suffix-Arrays für das Wort Banane: Phase 2, Iteration 1

Phase 2 - Iteration 2 - Schritt 1

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4		6		
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 2 - Schritt 2

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4		6		
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 2 - Schritt 3

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4		6		
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 2 - Ergebnis

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6		
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Tabelle 6.48: Konstruktion des Suffix-Arrays für das Wort Banane: Phase 2, Iteration 2

Phase 2 - Iteration 3 - Schritt 1

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6		
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 3 - Schritt 2

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6		
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 3 - Schritt 3

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6		
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 3 - Schritt 4

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6		
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 3 - Ergebnis

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6	3	
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Tabelle 6.49: Konstruktion des Suffix-Arrays für das Wort Banane: Phase 2, Iteration 3

Phase 2 - Iteration 4 - Schritt 1

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6	3	
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 4 - Schritt 2

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6	3	
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 4 - Ergebnis

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6	3	
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Tabelle 6.50: Konstruktion des Suffix-Arrays für das Wort Banane: Phase 2, Iteration 4

Phase 2 - Iteration 5 - Schritt 1

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6	3	
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 5 - Schritt 2

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6	3	
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 5 - Schritt 3

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6	3	
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 5 - Schritt 1

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6	3	
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Phase 2 - Iteration 5 - Ergebnis

Kontext	$	anane	ane	b	e	n	
Gruppe	7	2	4	1	6	3	5
SA	7	2	4	1	6	3	5
Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$

Tabelle 6.51: Konstruktion des Suffix-Arrays für das Wort Banane: Phase 2, Iteration 5

6.14.4 Der Algorithmus

Nachdem zuvor das allgemeine Vorgehen beschrieben und an einem Beispiel verdeutlicht wurde, wird in diesem Kapitel der konkrete Algorithmus als Pseudocode vorgestellt.

In Abbildung 6.48 ist der Pseudocode zu Phase 1 von GSACA zu sehen. Zunächst wird die initiale Einteilung in die lexikografisch sortierten Gruppen vorgenommen. Anschließend werden in die Gruppen in einer for-Schleife in absteigender Reihenfolge durchlaufen. In der ersten inneren for-Schleife speichert für jedes Zeichen der aktuellen Gruppe einen Zeiger zum direkt vorhergehenden Zeichen im Wort. Gibt es kein vorheriges Zeichen, wird die 0 als prev pointer gespeichert. Dies ermöglicht die Überprüfung, ob ein Zeichen ein vohergehendes Zeichen hat, indem der prev pointer mit 0 verglichen wird. Dies geschieht in einer späteren Codezeile in Phase 2. Anschließend werden alle lexikografisch kleineren Gruppenkontexte, auf die ein prev pointer eines Elements der gerade betrachteten Gruppe zeigt, in Untermengen geteilt. Diese Untermengen enthalten Elemente der lexikografisch kleineren Gruppen, auf die dieselbe Anzahl an prev pointern zeigt. In einer weiteren for-Schleife wird über diese Untermengen iteriert. Dabei wird jede Untermenge weiter unterteilt, so dass jede Teilmenge einer Untermenge Suffixe der gleichen Gruppe beinhaltet. Abschließend wird in einer letzten for-Schleife jede dieser neuen Teilmengen bearbeitet. Dazu wird jedes Suffix dieser Teilmengen aus seiner ursprünglichen Gruppe entfernt und zu einer neuen Gruppe als direkter Nachfolger der alten Gruppe hinzugefügt.

Abbildung 6.49 zeigt den Pseudocode zu Phase 2 von GSACA, in der das Suffix-Array aus den zuvor gebildeten Gruppen konstruiert wird. Hierzu wird zunächst das letzte Zeichen des Wortes, also das Terminalsymbol \$, als erstes Element der Liste SA gespeichert. Anschließend werden die Elemente dieser Liste in einer for-Schleife durchlaufen. Auch wenn zu Beginn nur ein einziges Element in der Liste enthalten ist, wird jeder Durchlauf weitere Elemente hinzufügen. In jeder Iteration ein das nächste Zeichen in SA bearbeitet und zu diesem Zeichen wird der Vorgänger im Wort betrachtet. Die Liste der prev pointer wird in einer nachfolgenden while-Schleife durchlaufen. In dieser while-Schleife wird zunächst die Position des gerade betrachteten Zeichens bestimmt. Die Position ergibt sich aus der Kardinalität der Menge aller Suffixe in kleineren Gruppen, inkrementiert um 1. Anschließend wird überprüft, ob SA bereits ein Zeichen an diesem Index enthält. Falls dies der Fall ist, wurden dieses Zeichen und die Elemente der prev pointer schon betrachtet und die aktuelle Iteration der for-Schleife wird abgebrochen. Gibt es jedoch noch kein Zeichen an dieser Stelle in der Liste SA, wird das aktuell betrachtete Zeichen dort gespeichert. Dann wird dieses Zeichen aus seiner aktuellen Gruppe entfernt und in eine neue, direkt vor der alten Gruppe liegende Gruppe hinzugefügt. Am Schluss der while-Schleife wird das Zeichen, auf den der prev pointer des aktuellen Zeichens zeigt, zum betrachteten Element für die nächste Iteration der while-Schleife.

```
def phase_1():
  # Order suffixes into groups according to first character
  groups = setupGroups()
  groups.sortByContext()
  for group in groups.reversed():
    for char in group:
      prev(char) = max({ j ∈ [1 ... char] | group(j) <
      ↪  group(char) } ∪ {0})
    # prevSuffixes is the set of previous suffixes from group
    prevSuffixes = { j ∈ [1 ... n] | prev(char) = j for any char
    ↪  ∈ group }
    # each subset prevSuffixes_l contains suffixes whose number of
    # prev pointers from group pointing to them is equal to l
    listOfSubsets = prevSuffixes.splitIntoSubsets()
    for subset in listOfSubsets.reversed:
      # split each subset into new subsets, such that
      # suffixes of same group are gathered in the same subset
      listOfSubgroups = subset.sortByGroup()
      for subGroup in listOfSubgroups:
        # create new group, to which all
        # suffixes of subGroup will be added to
        newGroup = Group()
        for suffix in subGroup:
          # remove suffixes of subGroup from their group
          # and put them into new group
          subGroup.remove(suffix)
          newGroup.append(suffix)
          # place new group as successor of their old group
          insertPosition = listOfSubgroups.indexOf(subGroup) + 1
          groups.insert(newGroup, insertPosition)
```

Abbildung 6.48: Phase 1 von GSACA

```
def phase_2():
  SA[1] = n
  for i = 1 up to n:
    j = SA[i] - 1
    while j ≠ 0:
      # number of suffixes placed in lower groups
      sr =  |{ s ∈ [1 ... n] | group(s) < group(j) }|
      if SA[sr + 1] ≠ nil:
        break
      SA[sr + 1] = j
      # remove j from its group and put it into a new group
      oldGroup = group(j)
      group(j).remove(j)
      newGroup = Group()
      newGroup.add(j)
      # place newGroup as immediate predecessor of oldGroup
      insertPosition = groups.indexOf(oldGroup) - 1
      groups.insert(newGroup, insertPosition)
      j = prev(j)
```

Abbildung 6.49: Phase 2 von GSACA

Im folgenden wird die erste Iteration der zweiten Phase exemplarisch am Beispiel Banane durchlaufen:
SA[1] = 7
for-Schleife wird mit $i = 1$ begonnen
j = SA[i] - 1 = SA[1] - 1 = 6
$j \neq 0$, in while-Schleife wird eingetreten
$sr = |\{\$, \text{anane}, \text{ane}, \text{b}\}| = 4$
SA[sr+1] = nil, daher wird die 1. Iteration der for-Schleife nicht abgebrochen
SA[sr+1] = j = 6
j wird aus der aktuellen Gruppe entfernt und in eine neue Gruppe eingefügt, die ein direkter Vorgänger der aktuellen Gruppe ist. Da j das einzige Element dieser Gruppe ist, findet keine Änderung statt.
j = prev(j) = 4. Die nächste Iteration der while-Schleife wird mit $j = 4$ durchlaufen.

6.14.5 Implementierung

In diesem Kapitel werden einige Details für die Implementierung besprochen.

Wie Uwe Baier in seiner Abhandlung über GSACA beschreibt, werden für eine Umsetzung des Algorithmus insgesamt folgende sechs Arrays benötigt:
SA enthält die nach der Reihenfolge der Gruppen sortierten Startpositionen der Suffixe.
ISA ist die inverse Permutation zum Array SA.
GSIZE speichert die Anzahl der Elemente jeder Gruppe. Diese wird am Anfang der Gruppe gespeichert und bis zur Ende mit Nullen aufgefüllt.
GLINK enthält Zeiger von den Suffixen zu den ersten Elementen ihrer entsprechenden Gruppen.
PREV beinhaltet die in Phase 1 berechneten prev pointer, die zu Beginn des Algorithmus alle mit nil initialisiert sind.
PC zählt die prev pointer von einer Gruppe zu einer anderen. Die Werte sind alle mit 0 initialisiert.
Die anfänglichen Werte der Arrays für das zuvor vorgestellte Beispiel Banane sind in Tabelle 6.52 zu sehen.

Index	1	2	3	4	5	6	7
Zeichen	b	a	n	a	n	e	$
Kontext	$	a		b	e	n	
SA	7	2	4	1	6	3	5
GSIZE	1	2	0	1	1	2	0
GLINK	4	2	6	3	7	5	1
ISA	4	2	6	3	7	5	1
PREV	nil	nil	nil	nil	nil	nil	nil
PC	0	0	0	0	0	0	0

Tabelle 6.52: Initiale Befüllung der verwendeten Datenstrukturen am Beispiel Banane

Der Autor beschreibt weiterhin, wie man diese Datenstrukturen nutzen kann, um verschiedene Aufgaben des Algorithmus effizient umzusetzen.

Das erste Problem, welches angesprochen wird, ist die initiale Sortierung der Gruppen nach absteigender lexikografischer Reihenfolge. Hierzu sei die Variable gs der Startindex der aktuellen Gruppe und ge der Endindex. Um nun zur nächsten Gruppe zu gelangen, wird der neue Endindex der nächsten Gruppe auf den direkten Vorgänger des Startindex der aktuellen Gruppe gesetzt, also ist $ge_{neu} = gs$ - 1. Der neue Startindex gs_{neu} kann bestimmt werden, indem zu dem neuen Endindex das Element aus dem Suffix-Array SA gesucht und für dieses mit GLINK der Zeiger auf das erste Element der Gruppe bestimmt wird. Somit ist gs_{neu} = GLINK[SA[gs - a]]. Das Bestimmen der Gruppengrenzen der

nächsten Gruppe geschieht in $\mathcal{O}(1)$ und die Iteration über alle Gruppen benötigt somit $\mathcal{O}(n)$ Zeit.

Als nächstes wird beschrieben, wie sich der prev pointer für ein Element berechnen lässt. Der prev pointer für ein Index kann bestimmt werden, indem der vorherige Index betrachtet wird. Von diesem neuen Index ausgehend wird die Kette der prev pointer so lange verfolgt, bis ein Index gefunden ist, dessen zugehöriges Zeichen im Wort zu einer niedrigeren oder der gleichen Gruppe gehört wie die Gruppe des ursprünglich betrachteten Index. Ob eine Gruppe kleiner als eine andere Gruppe ist, kann über die Liste GLINK überprüft werden. Falls der gefundene Index zu einer kleineren Gruppe gehört, wurde das Element, auf den der prev pointer des Elements am ursprünglichen Index zeigt, gefunden. Falls der gefundene Index aber zur gleichen Gruppe gehört, wird dieses Verfahren mit diesem neuen Index als Ausgangspunkt durchgeführt. Auf das dann gefundene Element wird von den prev pointern aller so in einem Schritt betrachteten Indices gezeigt. Dieses als *pointer jumping* bekannte Verfahren benötigt $\mathcal{O}(n)$ Zeit. Dieses Vorgehen kann am Beispiel aus Kapitel 6.14.3 gezeigt werden. In Tabelle 6.44 ist die zweite Iteration der ersten Phase dargestellt. Zu Beginn der zweiten Iteration wurde die Gruppe n schon betrachtet, in dieser Iteration wird also die Gruppe e untersucht. Der entsprechende Index im Wort ist 6, daher wird der vorherige Index 5 betrachtet. Das Zeichen an diesem neuen Index ist n und der prev pointer zeigt auf das Zeichen a an Index 4. Dieses Element gehört zu einer niedrigeren Gruppe, daher wurde der prev pointer von Index 6 bestimmt.

Anschließend wird im Originalpaper das Problem behandelt, wie im nächsten Teil des Algorithmus die Gruppen und Teilgruppen erstellt werden können. Dies geschieht in zwei Schritten. Zunächst wird das Array PC und dabei gleichzeitig das Set P erstellt. Dieser erste Schritt geschieht, indem alle Elemente des Wortes durchlaufen und deren prev pointer betrachtet werden. Währenddessen wird für ein Element i der Wert an PC[PREV[i]] um 1 inkrementiert und falls PC[PREV[i]] zuvor 0 war, wird PREV[i] in das Set P aufgenommen.
Für das Beispiel aus Kapitel 6.14.3 funktioniert das folgendermaßen:
PREV[1] = 0
PREV[2] = 0
PREV[3] = 2 $\Rightarrow$ PC[2] = 1 und P = P $\cup$ {2} = {2}
PREV[4] = 2 $\Rightarrow$ PC[2] = 2
PREV[5] = 4 $\Rightarrow$ PC[4] = 1 und P = P $\cup$ {4} = {2, 4}
PREV[6] = 4 $\Rightarrow$ PC[4] = 2
PREV[7] = 0
In einem zweiten Schritt werden nun die Teilmengen von P gebildet, in denen auf jedes Element einer Teilmenge gleich oft von einem prev pointer gezeigt wurde. Dazu wird das Set P durchlaufen, solange es nicht leer ist. In jeder Iteration wird für jedes Element aus P der Wert an der entsprechender Stelle in der Liste PC um 1 dekrementiert. Wenn der Wert von PC 0 wird, wird das Element aus dem Set P entfernt und zu einem neuen Set P_l hinzugefügt, wobei der Subscript l dem Index der Iteration entspricht.

Angewendet auf das Beispiel passiert folgendes:
Vor der 1. Iteration: PC[2] = PC[4] = 2
1. Iteration: PC[2] = 1 und PC[4] = 1 $\Rightarrow$ $P_1 = \{\}$
2. Iteration: PC[2] = 0 und PC[4] = 0 $\Rightarrow$ $P_2 = \{2, 4\}$

Danach geht der Autor darauf ein, wie die Suffixe neu angeordnet werden können. In der for-Schleife werden alle Teilmengen des Sets P durchlaufen, welche im vorherigen Schritt erstellt wurden. Um ein Element p aus einer dieser Teilmengen zu entfernen, wird es zunächst mit dem letzten Element seiner Gruppe unter Hilfe von SA und ISA getauscht. Das letzte Element der Gruppe von p ist bestimmbar durch GLINK[p] + GSIZE[GLINK[p]], da GLINK auf das erste Element einer Gruppe zeigt und GSIZE die Größe der Gruppe beinhaltet. Nun ist das Element p das letzte Element seiner Gruppe. Indem der Wert in GSIZE[GLINK[p]] um 1 dekrementiert wird, wird das Element p quasi aus seiner Gruppe entfernt. Im Anschluss muss GLINK[p] auf GLINK[p] + GSIZE[GLINK[p]] gesetzt werden, so dass es wieder auf den Beginn der neuen Gruppe zeigt. Abschließend wird auch die Größe der neuen Gruppe angepasst, indem GSIZE[GLINK[p]] erhöht wird.

Zum Schluss werden kurz verschiedene Einzelheiten zu Phase 2 erklärt. Es wird gesagt, dass der Wert sr, welcher die Anzahl der Suffixe aus niedrigeren Gruppen darstellt, aus SA und ISA berechnet werden kann. Außerdem führt der Autor auf, dass die Überprüfung, ob ein Element bereits im Suffix-Array enthalten ist, durchgeführt werden kann, indem überprüft wird, ob die Liste ISA an dem Index den Wert 0 enthält. Um ein Element j aus seiner aktuellen Gruppe zu entfernen und als direkten Nachfolger zu einer neuen Gruppe hinzuzufügen, wird zunächst der Wert von SA[ISA[j]] in der Variable sr gespeichert. Anschließend wird dieser Wert SA[ISA[j]] inkrementiert, das Element j in SA[sr] gespeichert und ISA[j] auf 0 gesetzt. Dies führt dazu, dass die zuvor erwähnte Überprüfung feststellt, dass j bereits in SA enthalten ist.

6.14.6 Illustration des Algorithmus an einem komplexen Beispiel

Nachfolgend wird der Algorithmus an dem Text *caabaccaabacaa* als weiteres und komplexeres Beispiel demonstriert. Dabei werden auch die zuvor beschriebenen Datenstrukturen eingebunden. Wie zuvor werden die aktuellen Schritte und daraus resultierende Änderungen farblich hervorgehoben. Allerdings werden die einzelnen Iterationen aufgrund des Umfangs dieses Beispiels nicht so detailliert beschrieben, wie in 6.14.3. Zunächst zeigen die Tabellen 6.53 bis 6.63 schrittweise die Abläufe in Phase 1, in der die Gruppen gebildet und die Liste der prev pointer aufgebaut wird. Anschließend werden in den Tabellen 6.64 bis 6.77 die einzelnen Iterationen zur Bildung des Suffix-Arrays in Phase 2 dargestellt.

Phase 1 - Start

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Kontext	$				a					b		c			
Gruppe	15	2	3	5	8	9	11	13	14	4	10	1	6	7	12
GSIZE	1	8	0	0	0	0	0	0	0	2	0	4	0	0	0
GLINK	12	2	2	10	2	12	12	2	2	10	2	12	2	2	1
ISA	12	2	3	10	4	13	14	5	6	11	7	15	8	9	1
PREV	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil

Tabelle 6.53: Konstruktion des Suffix-Arrays für das Wort caabaccaabacaa: Beginn von Phase 1

Phase 1 - Ergebnis Iteration 1

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Kontext	$				a			ac	acc	b		c			
Gruppe	15	2	3	8	9	13	14	11	5	4	10	1	6	7	12
GSIZE	1	6	0	0	0	0	0	1	1	2	0	4	0	0	0
GLINK	12	2	2	10	9	12	12	2	2	10	8	12	2	2	1
ISA	12	2	3	10	9	13	14	4	5	11	8	15	6	7	1
PREV	0	nil	nil	nil	nil	5	5	nil	nil	nil	nil	11	nil	nil	nil

Tabelle 6.54: Phase 1, Iteration 1. Beginnend mit der lexikografisch größten Gruppe spaltet sich der Kontext *a* in die Kontexte *a*, *ac* und *acc*. Dies spiegelt sich auch in den Listen GSIZE, GLINK, ISA und PREV wieder.

Phase 1 - Ergebnis Iteration 2

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Kontext	$			a			ab		ac	acc		b		c	
Gruppe	15	2	8	13	14	3	9	11	5	4	10	1	6	7	12
GSIZE	1	4	0	0	0	2	0	1	1	2	0	4	0	0	0
GLINK	12	2	6	10	9	12	12	2	6	10	8	12	2	2	1
ISA	12	2	6	10	9	13	14	3	7	11	8	15	4	5	1
PREV	0	nil	nil	3	nil	5	5	nil	nil	9	nil	11	nil	nil	nil

Tabelle 6.55: Phase 1, Iteration 2. Die Bearbeitung der nachfolgenden Gruppe mit dem Kontext *b* führt zu einer weiteren Aufteilung des Kontextes *a* in die Kontexte *a* und *ab*.

Phase 1 - Ergebnis Iteration 3

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Kontext	$			a		ab	abacc	ac	acc		b			c	
Gruppe	15	2	8	13	14	9	3	11	5	4	10	1	6	7	12
GSIZE	1	4	0	0	0	1	1	1	1	2	0	4	0	0	0
GLINK	12	2	7	10	9	12	12	2	6	10	8	12	2	2	1
ISA	12	2	7	10	9	13	14	3	6	11	8	15	4	5	1
PREV	0	nil	nil	3	3	5	5	nil	nil	9	nil	11	nil	nil	nil

Tabelle 6.56: Phase 1, Iteration 3. In diesem Schritt wird eines der beiden Elemente des Kontextes *ab* durch den lexikografisch größeren Kontext *acc* erweitert und bildet so den Kontext *abacc*.

Phase 1 - Ergebnis Iteration 4

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Kontext	$			a		abac	abacc	ac	acc		b			c	
Gruppe	15	2	8	13	14	9	3	11	5	4	10	1	6	7	12
GSIZE	1	4	0	0	0	1	1	1	1	2	0	4	0	0	0
GLINK	12	2	7	10	9	12	12	2	6	10	8	12	2	2	1
ISA	12	2	7	10	9	13	14	3	6	11	8	15	4	5	1
PREV	0	nil	nil	3	3	5	5	nil	nil	9	9	11	nil	nil	nil

Tabelle 6.57: Phase 1, Iteration 4. Auch der Kontext des anderen Elements der ehemaligen Gruppe *ab* wird erweitert und gehört nun zum Kontext *abac*.

Phase 1 - Ergebnis Iteration 5

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Kontext	$	a			aabacc	abac	abacc	ac	acc	b		c			
Gruppe	15	8	13	14	2	9	3	11	5	4	10	1	6	7	12
GSIZE	1	3	0	0	1	1	1	1	1	2	0	4	0	0	0
GLINK	12	5	7	10	9	12	12	2	6	10	8	12	2	2	1
ISA	12	5	7	10	9	13	14	2	6	11	8	15	3	4	1
PREV	0	nil	2	3	3	5	5	nil	nil	9	9	11	nil	nil	nil

Tabelle 6.58: Phase 1, Iteration 5. Wieder wird der Kontext *a* unterteilt. Der neue Kontext *aabacc* ist durch eine Erweiterung des Kontextes *a* durch einen lexikografisch größeren Kontext entstanden und ist somit selbst lexikografisch größer als *a*.

Phase 1 - Ergebnis Iteration 6

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b		c			
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
GSIZE	1	2	0	1	1	1	1	1	1	2	0	4	0	0	0
GLINK	12	5	7	10	9	12	12	4	6	10	8	12	2	2	1
ISA	12	5	7	10	9	13	14	4	6	11	8	15	2	3	1
PREV	0	nil	2	3	3	5	5	nil	8	9	9	11	nil	nil	nil

Tabelle 6.59: Phase 1, Iteration 6. Ein weiteres Element spaltet sich von der Gruppe mit Kontext *a* ab und ist nun dem Kontext *aabac* zugehörig.

Phase 1 - Ergebnis Iteration 7

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b		c			
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
GSIZE	1	2	0	1	1	1	1	1	1	2	0	4	0	0	0
GLINK	12	5	7	10	9	12	12	4	6	10	8	12	2	2	1
ISA	12	5	7	10	9	13	14	4	6	11	8	15	2	3	1
PREV	0	0	2	3	3	5	5	nil	8	9	9	11	nil	nil	nil

Tabelle 6.60: Phase 1, Iteration 7. Auch wenn die Bearbeitung des nächsten Kontextes keine weitere Änderung der Gruppen erzeugt, führt dies zur Füllung der prev pointer-Liste.

Phase 1 - Ergebnis Iteration 8

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b		c			
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
GSIZE	1	2	0	1	1	1	1	1	1	2	0	4	0	0	0
GLINK	12	5	7	10	9	12	12	4	6	10	8	12	2	2	1
ISA	12	5	7	10	9	13	14	4	6	11	8	15	2	3	1
PREV	0	0	2	3	3	5	5	0	8	9	9	11	nil	nil	nil

Tabelle 6.61: Phase 1, Iteration 8. Wie auch schon in der vorherigen Iteration ergeben sich keine Änderungen der Gruppenkontexte.

Phase 1 - Ergebnis Iteration 9

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b		c			
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
GSIZE	1	2	0	1	1	1	1	1	1	2	0	4	0	0	0
GLINK	12	5	7	10	9	12	12	4	6	10	8	12	2	2	1
ISA	12	5	7	10	9	13	14	4	6	11	8	15	2	3	1
PREV	0	0	2	3	3	5	5	0	8	9	9	11	0	0	nil

Tabelle 6.62: Phase 1, Iteration 9. Die Betrachtung des vorletzten Kontextes füllt weiter die Liste der prev pointer.

Phase 1 - Ergebnis Iteration 10

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b		c			
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
GSIZE	1	2	0	1	1	1	1	1	1		0	4	0	0	0
GLINK	12	5	7	10	9	12	12	4	6	10	8	12	2	2	1
ISA	12	5	7	10	9	13	14	4	6	11	8	15	2	3	1
PREV	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0

Tabelle 6.63: Phase 1, Iteration 10. Schließlich sind alle Kontexte in lexikografisch absteigender Reihenfolge bearbeitet worden. Sowohl Gruppenkontexte als auch die prev pointer-Liste wurden für die nächste Phase vorbereitet.

Phase 2 - Start

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b		c			
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Tabelle 6.64: Konstruktion des Suffix-Arrays für das Wort caabaccaabacaa: Beginn von Phase 2. Element 15 wird als Ausgangspunkt für die nächsten Iterationen in SA aufgenommen.

Phase 2 - Ergebnis Iteration 1

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b		c			
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	-	-	-	-	-	-	-	-	-	-	-	-	-

Tabelle 6.65: Phase 2, Iteration 1. Betrachteter Index: 1, enthaltener Wert: 15, Vorgängerelement: 14, prev pointer-Kette: 0. Element 14 wird in SA aufgenommen.

Phase 2 - Ergebnis Iteration 2

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b		c			
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	13	-	-	-	-	-	-	-	-	-	-	-	-

Tabelle 6.66: Phase 2, Iteration 2. Betrachteter Index: 2, enthaltener Wert: 14, Vorgängerelement: 13, prev pointer-Kette: 0. Element 13 wird in SA aufgenommen.

Phase 2 - Ergebnis Iteration 3

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b		c			
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	13	8	-	9	-	11	-	-	-	12	-	-	-

Tabelle 6.67: Phase 2, Iteration 3. Betrachteter Index: 3, enthaltener Wert: 13, Vorgängerelement: 12, prev pointer-Kette: $11 \rightarrow 9 \rightarrow 8 \rightarrow 0$. Elemente 8, 9, 11 und 12 werden in SA aufgenommen.

Phase 2 - Ergebnis Iteration 4

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b		c			
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	13	8	2	9	3	11	5	-	-	12	7	-	-

Tabelle 6.68: Phase 2, Iteration 4. Betrachteter Index: 4, enthaltener Wert: 8, Vorgängerelement: 7, prev pointer-Kette: $5 \rightarrow 3 \rightarrow 2 \rightarrow 0$. Elemente 2, 3, 5 und 7 werden in SA aufgenommen.

Phase 2 - Ergebnis Iteration 5

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b			c		
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	13	8	2	9	3	11	5	-	-	12	7	1	-

Tabelle 6.69: Phase 2, Iteration 5. Betrachteter Index: 5, enthaltener Wert: 2, Vorgängerelement: 1, prev pointer-Kette: 0. Element 1 wird in SA aufgenommen.

Phase 2 - Ergebnis Iteration 6

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b			c		
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	13	8	2	9	3	11	5	-	-	12	7	1	-

Tabelle 6.70: Phase 2, Iteration 6. Betrachteter Index: 6, enthaltener Wert: 9, Vorgängerelement: 8, prev pointer-Kette: 0. Keine neuen Elemente werden in SA aufgenommen.

Phase 2 - Ergebnis Iteration 7

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b			c		
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	13	8	2	9	3	11	5	-	-	12	7	1	-

Tabelle 6.71: Phase 2, Iteration 7. Betrachteter Index: 7, enthaltener Wert: 3, Vorgängerelement: 2, prev pointer-Kette: 0. Keine neuen Elemente werden in SA aufgenommen.

Phase 2 - Ergebnis Iteration 8

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b			c		
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	13	8	2	9	3	11	5	10	-	12	7	1	-

Tabelle 6.72: Phase 2, Iteration 8. Betrachteter Index: 8, enthaltener Wert: 11, Vorgängerelement: 10, prev pointer-Kette: $9 \rightarrow 8 \rightarrow 0$. Element 10 wird in SA aufgenommen.

Phase 2 - Ergebnis Iteration 9

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b			c		
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	13	8	2	9	3	11	5	10	4	12	7	1	-

Tabelle 6.73: Phase 2, Iteration 9. Betrachteter Index: 9, enthaltener Wert: 5, Vorgängerelement: 4, prev pointer-Kette: $3 \rightarrow 2 \rightarrow 0$. Element 4 wird in SA aufgenommen.

Phase 2 - Ergebnis Iteration 10

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b			c		
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	13	8	2	9	3	11	5	10	4	12	7	1	-

Tabelle 6.74: Phase 2, Iteration 10. Betrachteter Index: 10, enthaltener Wert: 10, Vorgängerelement: 9, prev pointer-Kette: 8. Keine neuen Elemente werden in SA aufgenommen.

Phase 2 - Ergebnis Iteration 11

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b			c		
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	13	8	2	9	3	11	5	10	4	12	7	1	-

Tabelle 6.75: Phase 2, Iteration 11. Betrachteter Index: 11, enthaltener Wert: 4, Vorgängerelement: 3, prev pointer-Kette: 2. Keine neuen Elemente werden in SA aufgenommen.

Phase 2 - Ergebnis Iteration 12

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b			c		
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	13	8	2	9	3	11	5	10	4	12	7	1	-

Tabelle 6.76: Phase 2, Iteration 12. Betrachteter Index: 12, enthaltener Wert: 12, Vorgängerelement: 11, prev pointer-Kette: $9 \rightarrow 8 \rightarrow 0$, Es werden keine neuen Elemente in SA aufgenommen.

Phase 2 - Ergebnis Iteration 13

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	c	a	a	b	a	c	c	a	a	b	a	c	a	a	$
Prev	0	0	2	3	3	5	5	0	8	9	9	11	0	0	0
Kontext	$	a		aabac	aabacc	abac	abacc	ac	acc	b		c			
Gruppe	15	13	14	8	2	9	3	11	5	4	10	1	6	7	12
SA	15	14	13	8	2	9	3	11	5	10	4	12	7	1	6

Tabelle 6.77: Phase 2, Iteration 13. Betrachteter Index: 13, enthaltener Wert: 7, Vorgängerelement: 6, prev pointer-Kette: $5 \rightarrow 3 \rightarrow 2 \rightarrow 0$. Element 6 wird in SA aufgenommen.

6.14.7 Optimierungen

Nachdem in den vorherigen Kapitel die Einzelheiten zu GSACA vorgestellt wurden, werden in diesem Kapitel Optimierungen und Änderungen am Algorithmus vorgestellt. Dabei wurde das Augenmerk auf eine Reduktion des Speicherverbrauchs gelegt. Ausgangspunkt war eine abgewandelte Implementierung des GSACA-Algorithmus, in dem die Liste GSIZE als Vektor gespeichert wurde. Wie zuvor beschrieben, speichert GSIZE die Größen der Gruppen. Dabei ist die eigentliche Größe lediglich am Startindex der Gruppe gespeichert, der Rest der Gruppeneinträge ist 0. Hierdurch bot dieser Bestandteil des Algorithmus einen guten Einstiegspunkt, um den verbrauchten Speicherplatz zu verringern. Die Idee zur Verbesserung, die verfolgt wurde, war, die Größe der einzelnen Elemente dieses Arrays zu reduzieren. Ursprünglich wurden alle Elemente des Vektors als Integer gespeichert, so dass je nach Größe der Integer bis zu 64 Bit pro Eintrag benötigt wurden. Durch geschickte Kodierung der Elemente gelang es, diese Größe auf zwei Bit pro Eintrag zu reduzieren.

Wie auch in der alten Version, wurden die Einträge in der neuen GSIZE-Liste mit geringerem Speicherverbrauch als Vektor gespeichert. Jedoch wurden sie nicht als Integer sondern als boolsche Werte gespeichert. Je zwei aufeinanderfolgende Bool-Werte kodieren dabei einen Eintrag in GSIZE, somit enthält der Vektor zwar doppelt so viele Einträge wie zuvor, aber jeder Eintrag belegt nur ein Bit. Das erste Bit kennzeichnet den Start einer Gruppe. Ist es auf true gesetzt, ist das entsprechende Element an dem Index das erste Element der Gruppe. In der ursprünglichen Version enthielte dieser Index die Anzahl der Elemente der Gruppe. Das zweite Bit markiert das Ende einer Gruppe. Diese Information ist wichtig, da beim Umsortieren der Gruppenkontexte in Phase 1 von GSACA die hinteren Elemente der Gruppen abgespalten werden, ohne dass die Gruppengröße direkt aktualisiert wird. Da die ursprüngliche Gruppengröße aber weiter verwendet wird, kann zu dem Zeitpunkt noch nicht das erste Element der Gruppe aktualisiert werden, es wird also gleichzeitig die alte Gruppengröße und die Größe der kleinen abgespaltenen Gruppen benötigt. Wird nun die Größe einer Gruppe gespeichert, wird das erste Bit des ersten Elements der Gruppe auf 1 gesetzt. Anschließend werden für alle weiteren Elemente der Gruppe sowohl das erste als auch das zweite Bit auf 0 gesetzt. Schließlich wird das zweite Bit des letzten Elements der Gruppe auf 1 gesetzt. Ein Beispiel für den Vergleich einer Gruppe mit der alten und der neuen Variante lässt sich in Tabelle 6.78 sehen. Das Lesen eines Wertes ist etwas komplexer als das Speichern eines Wertes. Wenn an der Index an einer beliebigen Stelle innerhalb einer Gruppe gelesen wird, ist das erste Bit 0, somit ist es nicht der Beginn der Gruppe. Wie bei der ursprünglichen Variante wird als Gruppengröße 0 zurückgegeben. Wird aber am Beginn einer Gruppe gelesen, ist also das erste Bit 1, muss die Größe der Gruppe erst bestimmt werden. Hierzu wird der Vektor von dem abgefragten Index an durchlaufen, solange bis ein Element erreicht wird, dessen erstes Bit 1 ist, das also den Beginn der nächsten Gruppe darstellt. Bei diesem Durchlauf wird die Anzahl der Elemente gezählt. So kann aus der Struktur die Gruppengröße bestimmt werden.

Index	0	1	2	3	4	5
GSIZE mit Integern	3	0	0	2	0	1
GSIZE mit Boolean	1 0	0 0	0 1	1 0	0 1	1 1

Tabelle 6.78: Vergleich der Speicherung verschiedener Gruppengrößen zwischen der ursprünglichen Speicherung mit Integern und der neuen Speicherung mit boolschen Werten.

Um diesen neuen Ansatz zu entwickeln, war es nötig, die Gleichheit der alten und der neuen Version nach jeder Operation sicherzustellen. Hierzu wurde zunächst eine Klasse GSIZE_LIST erstellt, welche die Verwaltung der alten Version kapselt. Diese Klasse bot Methoden zum lesenden und schreibenden Zugriff auf die einzelnen Element und wurde anschließend in dem Algorithmus verwendet. Dann wurde eine neue Klasse GSIZE_BOOL hinzugefügt, welche die gleichen Methoden wie GSIZE_LIST bereitstellte. Hierdurch konnte innerhalb des Algorithmus an nur einer Stelle die verwendete Klasse ausgewechselt werden, indem statt einer Instanz von GSIZE_LIST eine Instanz von GSIZE_BOOL initialisiert wurde. Jetzt konnte zwar die Implementierung einfach ausgetauscht werden, jedoch konnten die beiden Versionen noch nicht direkt miteinander verglichen werden. Um dies zu tun wurde als drittes die Klasse GSIZE_COMPARE erstellt. Auch diese Klasse enthielt die gleichen Methoden wie GSIZE_LIST und GSIZE_BOOL und konnte so ohne große Änderungen vom Algorithmus verwendet werden. GSIZE_COMPARE verwaltet je eine Instanz der beiden anderen Klassen und leitet alle Operationen an beide weiter. Verwendet GSACA nun eine Instanz dieser dritten Klasse und nutzt diese beispielsweise um einen Wert von GSIZE auszulesen, wird der Wert an dem abgefragten Index sowohl von GSIZE_LIST als auch von GSIZE_BOOL ausgelesen. Auch beim Setzen eines neuen Wertes wendet GSIZE_COMPARE diese Änderungen auf beide Implementierungen an. Auf diese Art können in jedem Schritt die Werte von der alten und der neuen Variante verglichen werden. Außerdem lässt sich bei Abweichungen direkt feststellen, in welchem Schritt es passiert ist, warum es gescheitert ist und, aufgrund der Richtigkeit von GSIZE_LIST, welcher Wert eigentlich in GSIZE_BOOL enthalten sein müsste. Durch dieses Vorgehen konnte das Verhalten von GSIZE_LIST auch in Grenzfällen passen analysiert und nachgebildet werden.

Leider benötigt die neue Version GSIZE_BOOL aber erheblich mehr Rechenaufwand, da sowohl beim Speichern eines neuen Wertes als auch beim Lesen eines Wertes die ursprüngliche Zahl berechnet werden muss. Durch diesen Ansatz konnte der Speicherverbrauch im Vergleich zu der vorherigen Implementierung als Vektor reduziert werden. Aufgrund der benötigten Umrechnung der Werte erhöhte sich jedoch die Laufzeit gegenüber dem vorherigen Ansatz deutlich. Die Ergebnisse von Laufzeit- und Speicherverbrauchsmessungen auf Datensätzen verschiedener Größe lässt sich in 6.79 sehen. Beide Varianten wurden drei mal ausgeführt, um Schwankungen auszugleichen. Die gemessenen Werte beziehen sich dabei auf die Ausführungsdauer des Algorithmus und beziehen nicht die Initialisierungsphase oder das Einlesen und Vorbereiten der Texte mit

ein. Als Grundlage diente der Text *english* des *Pizza&Chili Corpus* [21]. Wegen dieser starken Erhöhung der Laufzeit bei gleichzeitig nur relativ moderater Reduktion des Speicherverbrauchs ist der untersuchte Ansatz als keine sinnvolle Optimierung von GSACA anzusehen. Jedoch hat sich der beschriebene Ansatz zum Vergleichen mehrerer alternativen Implementierungen als praktisch erwiesen. Durch die parallele Ausführung mehrerer Varianten konnten schnell Abweichungen und Fehler gefunden werden. Zusätzlich hatten die beiden Implementierungen von GSIZE nach jeder Operation den gleichen Zustand, so dass sich Unterschiede schnell untersuchen lassen konnten und der für die Ungleichheit verantwortlich Schritt reproduzierbar verglichen werden konnte.

GSACA mit GSIZE_LIST

	Zeit in Millisekunden			Speicherpeak in Byte
	1. Messung	2. Messung	3. Messung	
1K	0,07	0,05	0,05	17.296
10K	0,65	0,51	0,64	165.168
100K	7,84	6,10	5,51	1.639.888
1M	176,87	163,13	174,05	16.778.944

GSACA mit GSIZE_BOOL

	Zeit in Millisekunden			Speicherpeak in Byte
	1. Messung	2. Messung	3. Messung	
1K	0,16	0,14	0,15	15.246
10K	11,26	8,34	8,56	144.686
100K	645,56	645,18	644,93	1.435.086
1M	70.417,64	70.432,93	70.391,62	14.681.790

Tabelle 6.79: Messergebnisse des Vergleichs der Laufzeit und des Speicherbedarfs zwischen GSACA mit GSIZE-Liste aus Integer (oben) und GSIZE-Liste aus boolsche Werte (unten) auf Texten der Größe 1K, 10K, 100K und 1M.

Kapitel 7

Messungen und Evaluation

In diesem Kapitel evaluieren wir die Performance der implementierten *Algorithmen* (Abschnitt 7.1), indem wir sie mit verschiedenen *Testdaten* (Abschnitt 7.2) und unterschiedlichen *Konfigurationen* (Abschnitt 7.3) auf einem gemeinsamen *Messsystem* (Abschnitt 7.4) ausführen. Die so erhaltenen Messwerte sind in Anhang B aufgeführt. Wir führen auf ihnen eine Reihe von Evaluationen durch:

In Abschnitt 7.5 betrachten wir zunächst alle sequentiellen Algorithmen, beginnend damit ob die Berechnung der Suffix-Arrays erfolgreich war (Abschnitt 7.5.1). Anschließend vergleichen wir alle eigenen Implementierungen bei Verarbeitung einer größeren Eingabe aus den *Large*-Testdaten in Abschnitt 7.5.2. Dasselbe tun wir für alle Referenzimplementierungen in Abschnitt 7.5.3.

In Abschnitt 7.5.4 gehen wir näher auf die einzelnen Implementierungen ein. Hierfür teilen wir die Ergebnisse in Gruppen von direkt miteinander verwandten Algorithmen ein und evaluieren für jede Gruppe das Verhalten in Hinblick auf verschiedenen Eingabegrößen.

Abschnitt 7.6 ist ähnlich zu Abschnitt 7.5 strukturiert, aber behandelt die parallelen Algorithmen. Wir überprüfen zunächst wieder in Abschnitt 7.6.1, ob die Suffix-Arrays erfolgreich berechnet werden konnten. Anschließend vergleichen wir zunächst in Abschnitt 7.6.2 das Verhalten aller Algorithmen zusammen und gehen in Abschnitt 7.6.3 wieder auf Details von einzelnen Gruppen von Algorithmen ein.

Zum Schluss betrachten wir noch in Abschnitt 7.6.4 das Verhalten von GPU-Algorithmen im Vergleich zu parallelen.

Testdatei (200MiB)	$\|\Sigma\|$	Beschreibung
`pc_dblp.xml`	96	Pizza&Chili Korpus [21].
`pc_dna`	16	
`pc_english`	225	
`pc_proteins`	25	
`pc_sources`	230	
`pcr_cere`	5	Pizza&Chili Repetitive Korpus [21]. Echte Texte.
`pcr_para`	5	
`pcr_einstein.en.txt`	124	
`pcr_kernel`	160	
`tagme_wiki-disamb30`	205	Wiki-Disamb30 aus dem TAGME Dateset [52].
`wiki_all_vital.txt`	204	Klartextauszug der *Most-Vital* Wikipedia Artikel [57].
`cc_commoncrawl.ascii`	114	Zufälliger Klartextauszug von ASCII Seiten aus Commoncrawl [13].

Testdatei (100GiB)	$\|\Sigma\|$	Beschreibung
`commoncrawl.txt`	242	Größerer zufälliger Klartextauszug von ASCII Seiten aus Commoncrawl [13].
`wiki.txt`	213	XML Datendump von Wikipedia.
`dna.txt`	4	DNA Daten aus dem 1000 Genomes Projekt ohne Fastq-Daten.

Tabelle 7.1: Alle für die Messung benutzten Testdaten.

7.1 Algorithmen

Die Messung erfolgt mit zwei verschiedenen Arten von Algorithmen. Zum einen mit den von der Projektgruppe vorgenommenen Implementierungen, und zum anderen mit den Referenzimplementierungen der originalen Paper, falls diese vorhanden sind. Letztere sind mit dem Namenszusatz *ref* gekennzeichnet.

Wir unterscheiden des weiteren zwischen sequentiellen und parallelen Algorithmen, wobei letztere durch Variationen von *par*allel im Namen gekennzeichnet sind.

7.2 Testdaten

Die Testdaten stammen aus zwei verschiedenen Quellen und sind in Tabelle 7.1 beschrieben. Sie bestehen zum einen aus einer Auswahl an verschiedenen *Small-*

Texten mit eine Größe von 200MiB, die durch das `make datasets` Target des Buildsystems bereitgestellt werden. Zum anderen bestehen sie aus aus einer geringen Auswahl an *Large*-Texten mit einer Größe von 100GiB, von denen Prefixe für die Scaling-Experimente benutzt werden.

7.3 Messverfahren

Sei *input* eine Eingabedatei, *algorithm* ein Algorithmus, k eine Größe der Eingabe, t eine Anzahl der zu benutzenden Threads, r eine Anzahl von Messwiederholungen und b eine Anzahl von Bits für den `sa_index` Typ. Wir führen Messungen mit 4 verschiedenen Konfigurationen durch, wobei immer $r = 3$ gilt. Es gilt außerdem $b = 32$ für $k < 2\text{GiB}$ und $b = 64$ sonst:

Konfiguration	*input*	*algorithm*	k	t
Small-Sequential	Small	Alle Sequentiellen	200MiB	N.A.
Large-Sequential-Input-Scaling	Large	Alle Sequentiellen	$t * 200\text{MiB}$	$\in [1, 2, 4, 8, 12, 16, 20]$
Large-Parallel-Weak-Scaling	Large	Alle Parallelen	$t * 200\text{MiB}$	$\in [1, 2, 4, 8, 12, 16, 20]$
Large-Parallel-Strong-Scaling	Large	Alle Parallelen	200MiB	$\in [1, 2, 4, 8, 12, 16, 20]$

Für jede Konfiguration wird der Befehl `taskset -c 0,1,...,`t `sacabench batch` *input* mit den Optionen `--whitelist` *algorithm*, `--check`, `--prefix` k, `--repetitions` r, `-b` b und `--benchmark` $outfile$ aufgerufen. Hierbei ist $outfile$ jeweils ein eindeutiger Dateipfad.

Dieses Verfahren wird in der Framework Implementierung durch den Python Skript `zbmessung/lido3_launcher.py` automatisiert.

7.4 Messsystem

Die Messung erfolgt auf dem Lido3-Clusters der TU Dortmund [20]. Alle Knoten haben die selbe Spezifikation, die in Tabelle 7.2 beschrieben ist. Die Daten wurden durch die Linux Befehle `uname`, `lscpu` und `free` ermittelt.

Betriebssystem	GNU/Linux
Kernel	Linux
Kernel Release	3.10.0-862.14.4.el7.x86_64
Kernel Version	#1 SMP Wed Sep 26 15:12:11 UTC 2018
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	20
On-line CPU(s) list:	0-19
Thread(s) per core:	1
Core(s) per socket:	10
Socket(s):	2
NUMA node(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	79
Model name:	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
Stepping:	1
CPU MHz:	2599.951
CPU max MHz:	3400.0000
CPU min MHz:	1200.0000
BogoMIPS:	4789.01
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	25600K
NUMA node0 CPU(s):	0-9
NUMA node1 CPU(s):	10-19
Speichergröße	64GiB

Tabelle 7.2: Technische Spezifikation eines Knotens des Lido3-Clusters.

7.5 Ergebnisse Sequentielle SA Konstruktion

7.5.1 Suffix-Array Korrektheit

Wir überprüfen zunächst, ob alle Testdaten von allen Implementierungen korrekt verarbeitet werden konnten. Die Messergebnisse enthalten hierfür die von `--check` erzeugte Informationen und können in Anhang B.1.2 (Tabelle B.4) eingesehen werden. Dabei wurde die Korrektheit aller sequentieller Algorithmen mit verschiedenen Testdaten der Größe 200 MiB festgehalten. Dabei fällt positiv auf, dass sowohl die Implementierungen der Projektgruppe, als auch die Referenzimplementierungen der originalen Paper bei den meisten Testdaten der Größe 200 MiB das Suffix-Array korrekt bestimmen. Lediglich bei den Texten `pcr_cere`, `pcr_kernel` und `pcr_para` überschreitet vereinzelt die Berechnungsdauer der Implementierungen das Zeitlimit des Messsystems. Das ist der Fall, weil diese Texte repetitiv sind und damit einige Algorithmen eine längere Berechnungszeit benötigen. Aber kein Algorithmus hat das Suffix-Array falsch berechnet oder ist abgestürzt.

7.5.2 Vergleich der eigenen Implementierungen

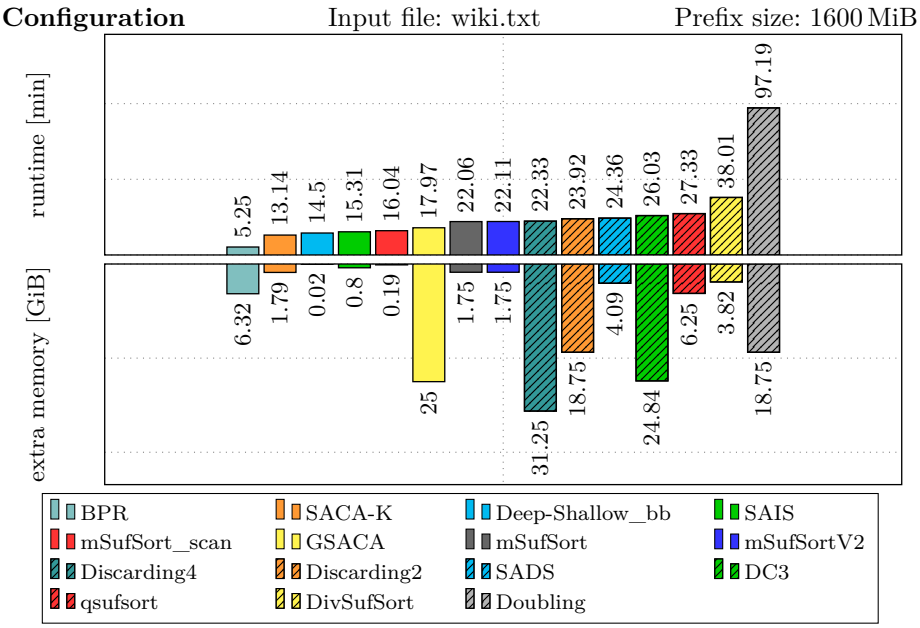


Abbildung 7.1: Vergleich der sequentiellen Implementierungen mit Laufzeit und Speicherbedarf auf wiki.txt

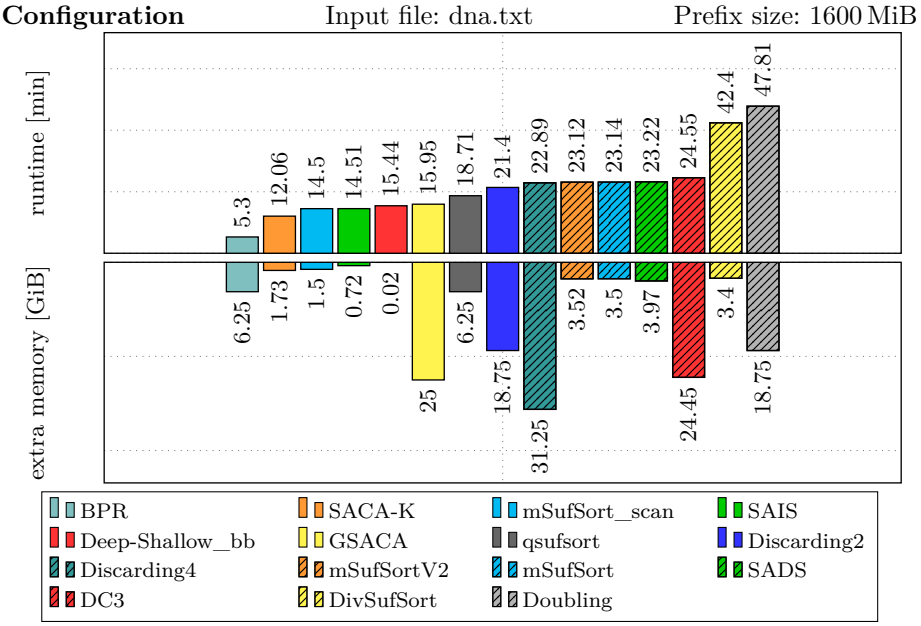


Abbildung 7.2: Vergleich der sequentiellen Implementierungen mit Laufzeit und Speicherbedarf auf dna.txt

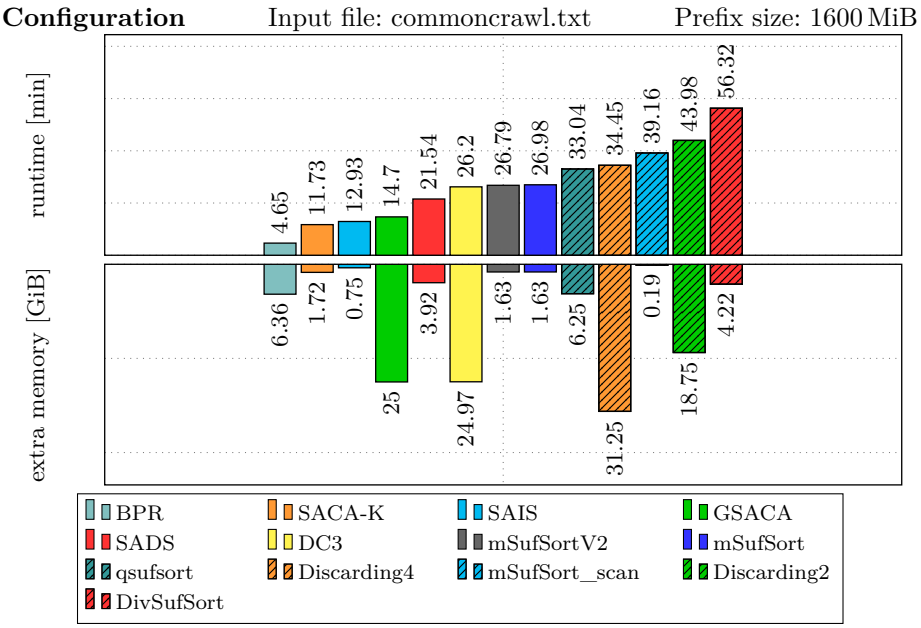


Abbildung 7.3: Vergleich der sequentiellen Implementierungen mit Laufzeit und Speicherbedarf auf commoncrawl.txt

In den Abbildungen 7.1 bis 7.3 sind die Laufzeiten und der jeweilige zusätzliche Speicherbedarf der Implementierungen der Projektgruppe festgehalten. Dabei wurden die Algorithmen auf den drei Eingabetexten `wiki.txt`, `dna.txt` und `commoncrawl.txt` mit jeweils einer Größe von 1600 MiB ausgewertet.

Die Laufzeitmessung zeigt, dass der BPR die beste Laufzeit aller von der Projektgruppe umgesetzten Implementierungen auf den drei Eingabetexten aufweist - dicht gefolgt von SACA-K, der jeweils den zweiten Platz belegt. Demgegenüber stehen der DivSufSort und der Doubling-Algorithmus, die jeweils den vorletzten bzw. letzten Platz belegen. Der mSufSort_scan belegt zwar bei den Testdaten `wiki.txt` und `dna.txt` den dritten Platz bezüglich der Laufzeit, allerdings belegt dieser Algorithmus den drittletzten Platz bei dem Eingabetext `commoncrawl.txt`. Generell ändert sich bei diesem Text die Reihenfolge einiger Implementierungen. Neben dem mSufSort_scan verbessert sich der DC3 von dem 12. Platz auf den sechsten Platz. Auch ein Grund dafür ist, dass die Implementierungen des Deep-Shallow, Deep-Shallow_bb und des Doubling-Algorithmus in der Abbildung fehlen. Diese drei Implementierungen überschreiten bei dem Eingabetext `commoncrawl.txt` das Zeitlimit des Messsystems, sodass sie daher nicht in die Abbildung aufgenommen werden konnten.

Die Speichermessung zeigt, dass die beiden Deep-Shallow Varianten die Implementierungen sind, die den geringsten zusätzlichen Speicherplatz benötigen. Diese beiden Algorithmen verbrauchen bei einer Eingabegröße von 1600 MiB nur knapp 20 MiB zusätzlichen Speicherplatz. Die von der Projektgruppe umgesetzten Algorithmen Osipov_sequential_wp, Discarding4, GSACA und DC3 belegen die letzten Plätze.

Genauere Informationen zu den Auswertungen können dem Anhang B.1.1 entnommen werden.

In Tabelle B.2 sind die zu Tabelle B.1 gehörigen Laufzeiten der Algorithmen. Die Implementierungen, dessen Berechnungszeit das Zeitlimit des Systems überschritten haben, weisen ein '-' in der jeweiligen Spalte auf. Zusätzlich sind je Eingabetext die drei besten Implementierungen grün und die drei schlechtesten Implementierungen in rot markiert.

Zusätzlich liegt im Anhang B.1.2 eine weitere Auswertung vor. In dieser Versuchsreihe werden die Algorithmen auch auf größeren Eingabetexten miteinander verglichen. Dabei sind die jeweiligen sequentiellen Algorithmen auf den Eingabetexten `commoncrawl.txt`, `dna.txt` und `wiki.txt` getestet worden. Die Größe der Texte variiert dabei. Die Tabelle B.4 ist also so zu lesen, dass der jeweilige Text in der ersten Spalte eine Größe von 200 MiB aufweist, in der zweiten Spalte $2 \cdot 200$ MiB = 400 MiB, in der dritten Spalte $4 \cdot 200$ MiB = 800 MiB usw..

In der Tabelle B.4 fällt auf, dass die meisten Algorithmen bei einer Eingabegröße von $12 \cdot 200$ MiB = 2400 MiB abbrechen. Dies ist der Fall, weil das Messsystem eine Speichergröße von 64 GiB hat, die Algorithmen aber mehr

Speicherplatz benötigen, um das Suffix-Array zu berechnen. Zusätzlich überschreiten manche Algorithmen wie der Deep-Shallow, Deep-Shallow_bb und NzSufSort bei bereits unter 1000 MiB das Zeitlimit des Systems.

7.5.3 Vergleich der Referenzimplementierungen

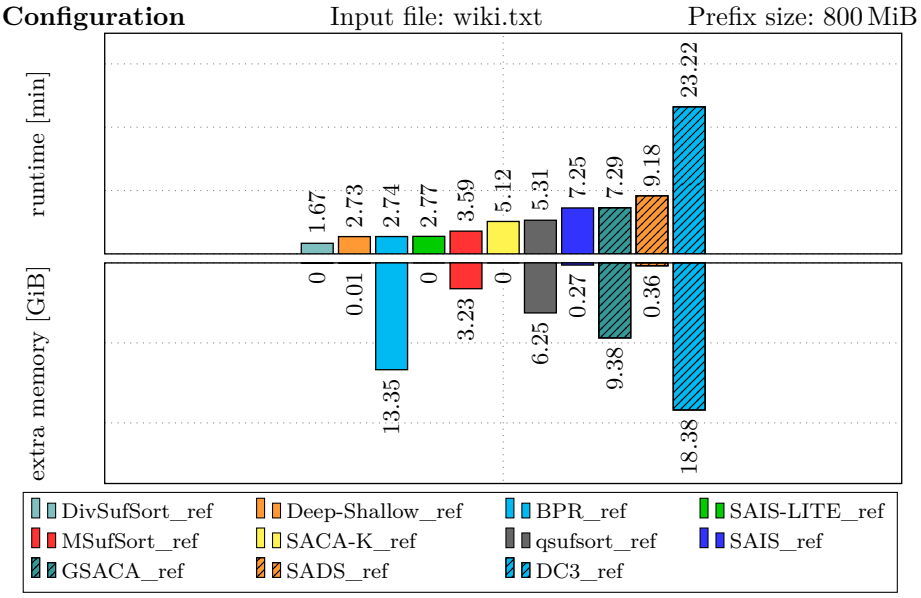


Abbildung 7.4: Vergleich der sequentiellen Referenzimplementierungen mit Laufzeit und Speicherbedarf auf wiki.txt

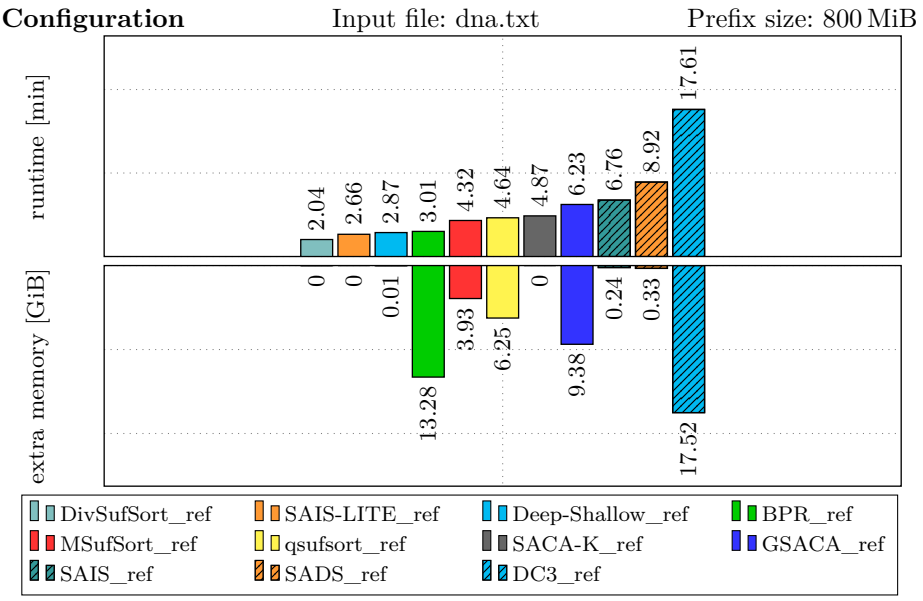


Abbildung 7.5: Vergleich der sequentiellen Referenzimplementierungen mit Laufzeit und Speicherbedarf auf dna.txt

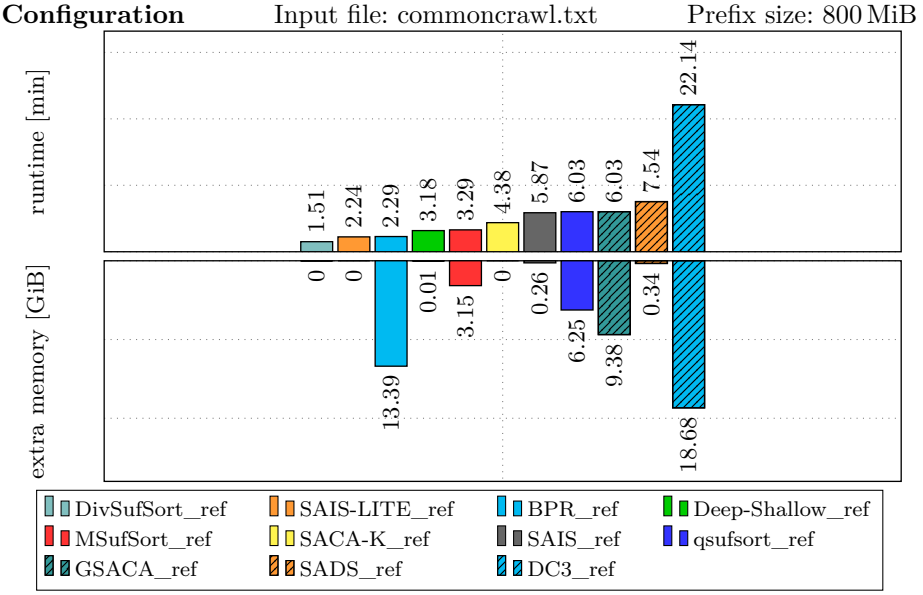


Abbildung 7.6: Vergleich der sequentiellen Referenzimplementierungen mit Laufzeit und Speicherbedarf auf commoncrawl.txt

In diesem Kapitel werden die sequentiellen Referenzimplementierungen miteinander verglichen. Wie im vorherigen Abschnitt sind die Implementierungen auf demselben Messsystem mit denselben Eingabetexten getestet worden. Die Präfixgröße wurde jedoch von 1600 MiB auf 800 MiB heruntergesetzt, da nicht alle Referenzalgorithmen für derart große Eingaben lauffähig sind.

Die Laufzeitmessung auf den Testdaten `wiki.txt` und `commoncrawl.txt` zeigen, dass die Implementierungen des DivSufSort_ref und SAIS-LITE_ref die besten Laufzeiten aufweisen. Auf dem repetitiven Text `dna.txt` liegt der Deep-Shallow_ref jedoch auf dem zweiten Platz. Die schlechtesten Algorithmen in dieser Auswertung sind der DC3_ref und der SADS_ref.

Die Speichermessung der Referenzimplementierungen legen dar, dass sie sich bezüglich des zusätzlichen Speicherverbrauches in zwei Gruppen aufteilen. Eine Gruppe benötigt überwiegend keinen Extra-Speicher und die andere verbraucht relativ viel Extra-Speicher. Zu der ersten Gruppe gehören die Implementierungen DivSufSort_ref, Deep-Shallow_ref, SACA-K_ref, SAIS_ref und SADS_ref. Zu der zweiten Gruppe zählen DC3_ref, BPR_ref, qSufSort_ref und GSACA_ref. Diese Reihenfolge ist unabhängig von den Testdaten. So benötigt der DC3_ref bei einer Eingabegröße von 800 MiB je nach Datei rund 18 GiB zusätzlichen Speicher. Demgegenüber berechnen der DivSufSort_ref und SACA-K_ref das Suffix-Array ohne zusätzlichen Speicher. Aber auch der SAIS_ref nimmt nur etwa 250 MiB zusätzlichen Speicher in Anspruch.

Auch zu den Referenzimplementierungen können weitere Informationen dem Anhang B.1.1 entnommen werden. Dort werden die sequentiellen Algorithmen der Projektgruppe, als auch die der Referenzimplementierungen bezüglich der Laufzeit und des zusätzlichen Speicherverbrauchs tabellarisch gegenübergestellt. Die jeweils besten und schlechtesten Implementierungen sind dabei in grün, beziehungsweise rot markiert.

Der Vergleich der Referenzimplementierungen mit den Implementierungen der Projektgruppe zeigt, dass DivSufSort_ref auf allen Eingaben der schnellste Algorithmus ist - dicht gefolgt von BPR der Projektgruppe und der Referenzimplementierung SAIS-LITE_ref. Die langsamsten Algorithmen sind Doubling, NzSufSort und DC3-Lite.

Die Algorithmen mit dem geringsten zusätzlichen Speicherverbrauch sind DivSufSort_ref, SACA-K_ref und der naive Algorithmus. Aber auch die beiden Deep-Shallow Varianten der Projektgruppe und der NzSufSort benötigen nur minimalen Extra-Speicher.

Lediglich ab der Eingabegröße von 2400 MiB wechselt die Reihenfolge, da unter anderem die Referenzimplementierungen DivSufSort_ref und SAIS-LITE_ref ab dieser Eingabegröße abbrechen. Ist die Eingabe größer als 2400 MiB, ist BPR der Projektgruppe der schnellste Algorithmus und mSufSort_scan verbraucht nach SACA-K_ref den geringsten Extra-Speicher.

7.5.4 Skalierbarkeit

In den folgenden Kapiteln untersuchen wir Gruppen sequentieller Implementierungen in Bezug auf ihre Skalierbarkeit. Dabei wird die Präfixgröße kontinuierlich bis maximal 4GB Eingabegröße erhöht und dabei die Laufzeit und der Speicherverbrauch in Abhängigkeit davon gemessen.

BPR

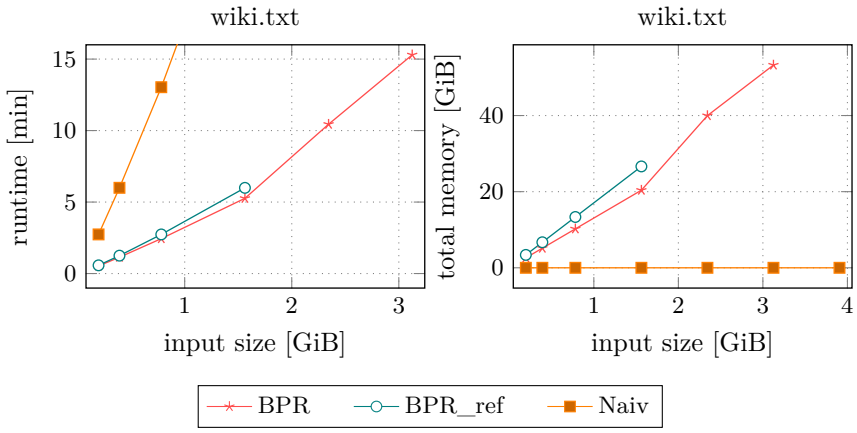


Abbildung 7.7: BPR und Vergleichsalgorithmen auf wiki.txt

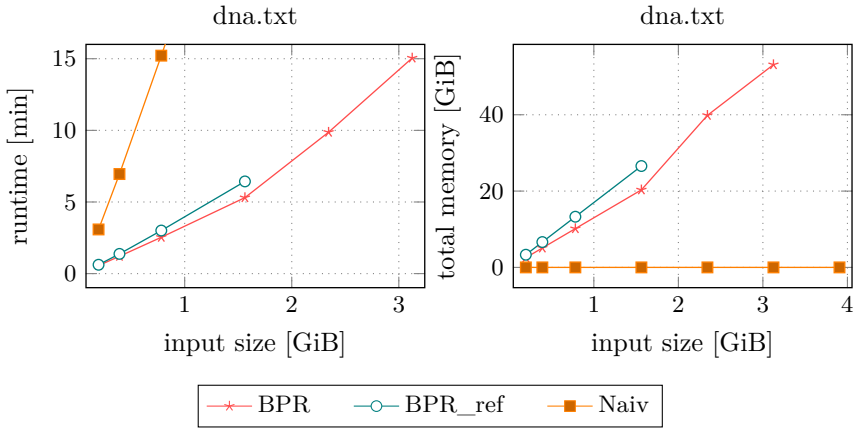


Abbildung 7.8: BPR und Vergleichsalgorithmen auf dna.txt

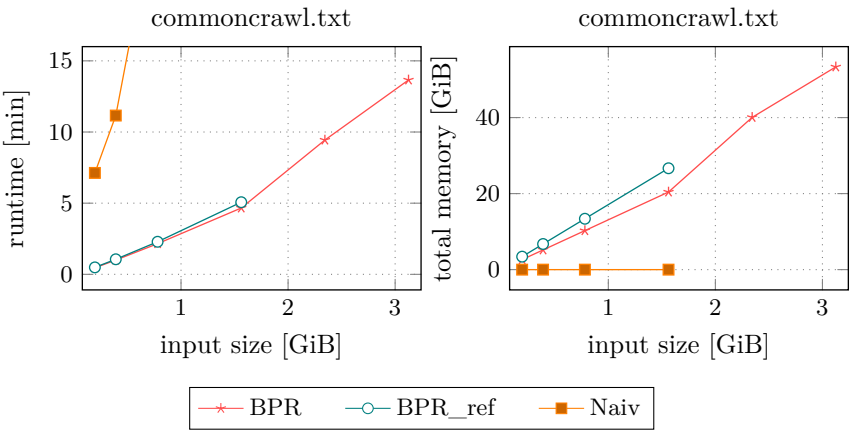


Abbildung 7.9: BPR und Vergleichsalgorithmen auf commoncrawl.txt

Für die Auswertung der Implementierung von *Bucket-Pointer Refinement* wurde die von uns implementierte Version der Referenzimplementierung auf verschiedenen Dateien gegenübergestellt.

Die Speichermessung lässt anhand der oben stehenden Diagramme (Abbildungen 7.7 bis 7.9) sofort erkennen, dass die Beschränkung der Ressourcen im Bezug auf Hauptspeicher eine große Rolle bei der Messung gespielt hat: Die auf dem Rechenknoten verfügbaren 64 GB Hauptspeicher erlauben auf allen getesteten Eingabedateien im Falle der Referenzimplementierung eine maximale Größe von 1.6 GB und im Falle der eigenen Implementierung maximal 3.2 GB. Die größte getestete Eingabe mit einer Dateigröße von 4 GB konnte durch die Beschränkung der Ressourcen und den hohen Bedarf an Hauptspeicher von keinem der beiden vorliegenden Algorithmen verarbeitet werden. Aus Gründen der Unterscheidbarkeit von BPR und Vergleichsalgorithmen wurden in den Diagrammen die Messpunkte des naiven Algorithmus zum Vergleich ausgelassen. Die entsprechenden Werte sind den Tabellen B.5 und B.6 zu entnehmen.

Der Vergleich beider Algorithmen zeigt die aus der Theorie herleitbare und daher zu erwartende Struktur: Während der Speicherbedarf beider Algorithmen nahezu unabhängig von der Art der Eingabedatei ist, liegt der Bedarf der eigenen Implementierung für Dateien kleiner als 2 GB etwas unter der der Referenzimplementierung, da letztere für jede Eingabegröße 64 Bit Indizes für das Suffix-Array verwendet und somit auch für kleinere Eingaben große Suffix-Arrays verwendet. Zu beachten ist, dass an dieser Stelle im Gegensatz zu allen anderen Plots der Gesamtspeicherbedarf statt des Zusatzspeichers zum Vergleich herangezogen wurde. Der Grund dafür ist der Quellcode der Referenzimplementierung, welcher sich nicht in ausreichender Tiefe in unser Framework integrieren lässt, um die Allokation verschiedener Speicherstrukturen nähergehend zu unterscheiden.

Die Laufzeitmessung zeigt, dass die Optimierungen erfolgreich waren, wenngleich die Laufzeitersparnis je nach Eingabe nur 7.5% bis 16.7% beträgt. Die Messung bestätigt insgesamt aber die Erwartungen an den Zuwachs an Performanz.

Deep-Shallow

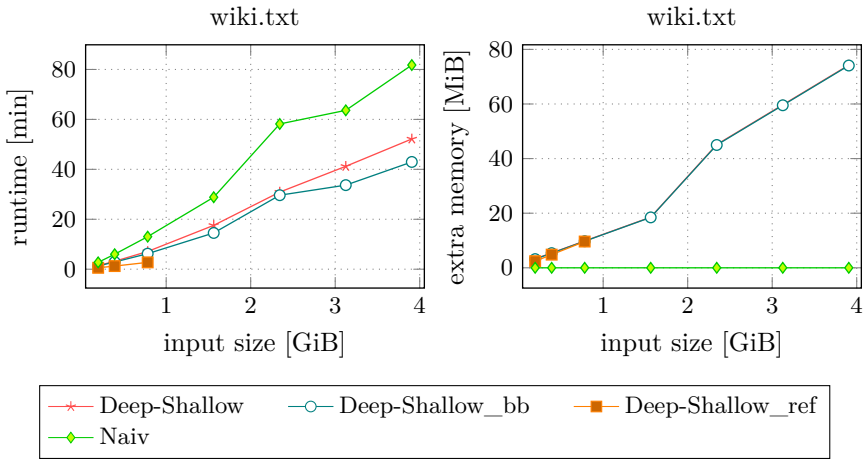


Abbildung 7.10: Deep-Shallow, Deep-Shallow_bb und Deep-Shallow_ref auf wiki.txt

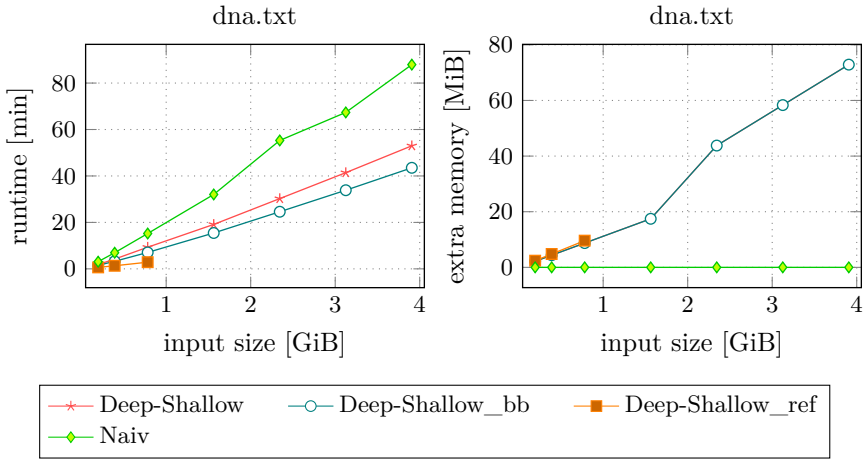


Abbildung 7.11: Deep-Shallow, Deep-Shallow_bb und Deep-Shallow_ref auf dna.txt

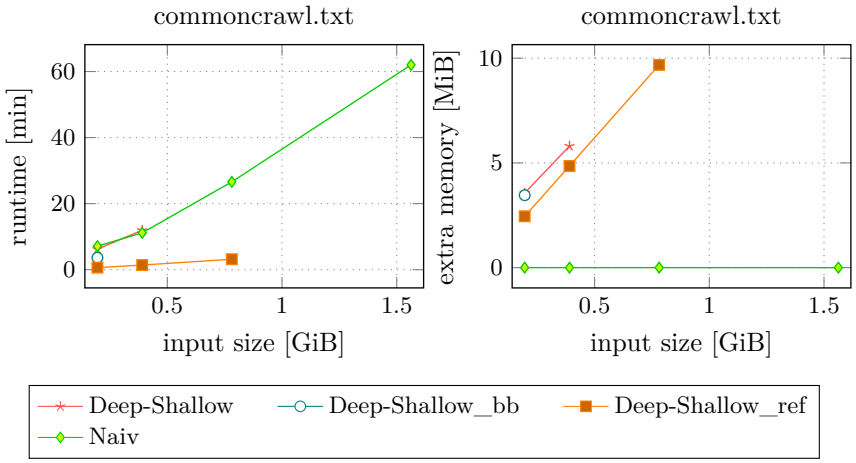


Abbildung 7.12: Deep-Shallow, Deep-Shallow_bb und Deep-Shallow_ref auf commoncrawl.txt

Die Speichermessung in den Abbildungen zu den drei Texten `wiki`, `dna` und `commoncrawl` lässt erkennen, dass die beiden Deep-Shallow Varianten (Deep-Shallow und Deep-Shallow_bb) nur geringen Speicherverbrauch haben, da sie etwa 1% des Eingabetextes als Hilfsspeicher benötigen. Dieser Speicherverbrauch ist unabhängig vom verwendeten Text. Der Referenz-SACA (Deep-Shallow_ref) verwendet noch ein bisschen weniger Speicher und der naive SACA verwendet keinen Extraspeicher.

Die Laufzeitmessung zeigt in den ersten beiden Abbildungen ein ähnliches Verhalten: Der schnellste Algorithmus zwischen 100 und 800 MiB ist jeweils die Referenzimplementierung. Ab 1600 MiB ist ihr jedoch nicht mehr möglich das SA zu berechnen. Dies liegt daran, dass sie innerhalb des Zeitlimits von zwei Stunden nicht terminiert ist. Der Algorithmus verwendet zwar 32-Bit-Integer und verwendet außerdem eins dieser Bits als Tag, dies schränkt den adressierbaren Text jedoch nur auf $2^{31} \approx 2\,\text{GiB}$ ein. Daher ist zu vermuten, dass diese (relativ alte) Implementierung andere Annahmen trifft, die die Textgröße weiter einschränken. Unsere Implementierungen schaffen es, das SA bis zu 4 GiB zu berechnen. Dabei liegt ihre Laufzeit immer zwischen dem naiven SACA und der Referenz. Es zeigt sich außerdem, dass die Variante mit Big-Buckets (Abschnitt 6.5.4) in allen Fällen eine leicht bessere Laufzeit als die andere Variante aufweist. Der naive SACA ist in allen Fällen schlechter.

In der dritten Abbildung zum Text `commoncrawl` zeigt sich leider die schlechte Worst-Case-Laufzeit unserer Implementierung von Deep-Shallow. Unsere Deep-Shallow-Implementierung ohne Big-Buckets (siehe Abschnitt 6.5.4) schafft es hier zwar, bis 400MB das SA zu berechnen, allerdings reicht die Zeit (2 Stun-

den) nicht aus, um das SA für 800MB Text zu berechnen. Die Deep-Shallow-Big-Buckets-Implementierung schafft es nicht, das 200MB-SA zu berechnen.

Vergleichend lässt sich sagen, dass unsere Implementierungen zwar den naiven SACA auf zwei Texten schlagen, die optimiertere Referenzimplementierung allerdings in fast allen Fällen besser ist.

Doubling und Discarding

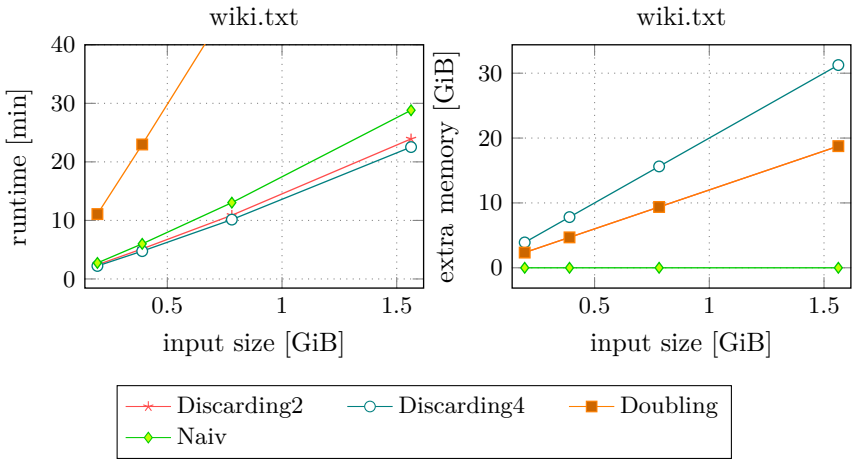


Abbildung 7.13: Discarding2, Discarding4 und Doubling auf wiki.txt

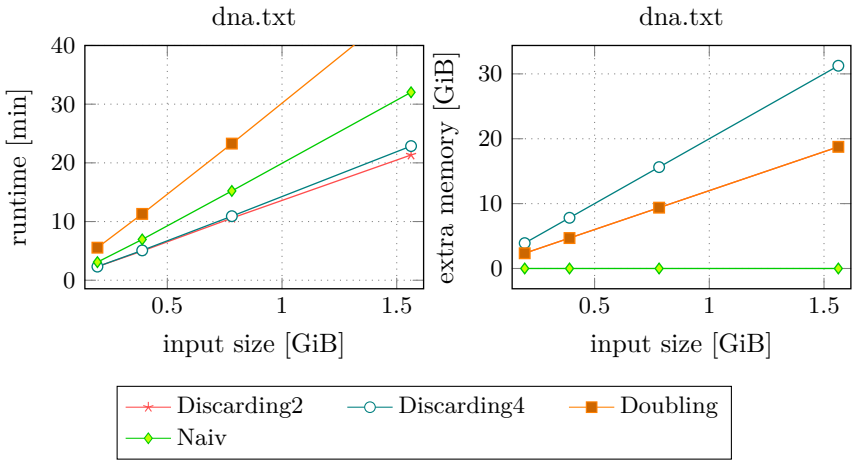


Abbildung 7.14: Discarding2, Discarding4 und Doubling auf dna.txt

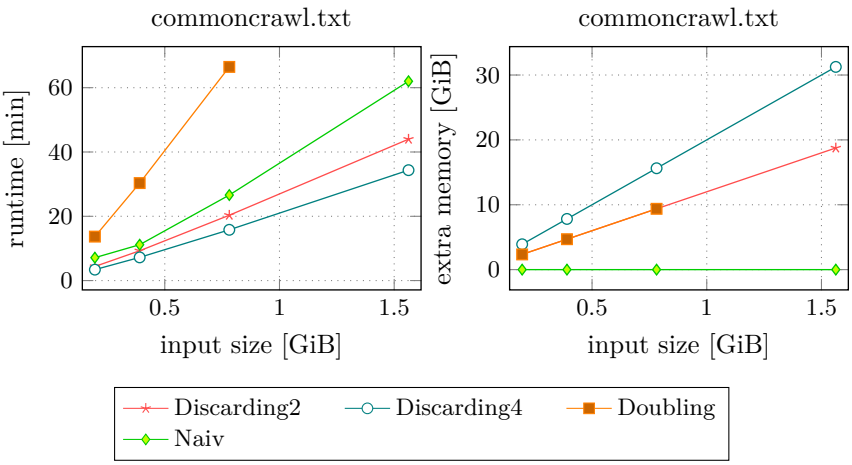


Abbildung 7.15: Discarding2, Discarding4 und Doubling auf commoncrawl.txt

Wir werten nun das Verhalten des *Doubling*-Algorithmus, sowie zwei seiner Varianten mit *Discarding* und a-Tupling für $a = 2$ und $a = 4$ (siehe Abschnitt 6.3), bei skalierender Eingabegröße aus.

Die Speichermessung zeigt, dass der Speicherverbrauch der Algorithmen rein von der Länge der Eingabe und der Größe von `sa_index` abhängt. Dies ist aus den Diagrammen und Tabellen B.3 und B.6 ersichtlich.

Dies stimmt mit ihren erwartete Verhalten basierend auf der Implementierung (Abschnitt 6.3.9) überein, gemäß der alle Varianten des Algorithmus nur auf einem Array von $|\mathsf{T}|$ Elementen arbeiten. So liegt zum Beispiel der theoretische Speicherverbrauch des Discardings mit a=4 und der Arrayüberlagerungsoptimierung für eine 200 MiB Eingabe bei 200 [MiB]$*(a+1)*$`sizeof(sa_index)` = 4000 [MiB] = 3.907 [GiB], was fast genau den gemessenen Betrag entspricht.

An Tabelle B.4 erkennt man jedoch auch, das der Speicherverbrauch relativ hoch ist, da ab einer Eingabelänge von über 1600 MiB das Speicherlimit des Systems erreicht wird. Der Algorithmus ist somit nicht für Speicher-limitierte Systeme geeignet.

Die Laufzeitmessung in Tabelle B.2 und Tabelle B.5 zeigt, dass das reine *Doubling* einer der langsamsten Algorithmen ist. Dies ist bei seiner theoretischen Laufzeit von $\mathcal{O}(\text{sort}(n)\lceil \log \text{maxlcp} \rceil)$ (Abschnitt 6.3.4) plausibel, da Suffix-Indexe die eindeutig feststehen ggf. wiederholt neu bestimmt werden.

Der optimierte Algorithmus – *Discarding* mit a-Tupling, Pipelining, Arrayüberlagerung und Wordpacking – behandelt stattdessen jeden Suffix-Index nur solange bis er eindeutig feststeht. Dies lässt sich aus den Diagrammen und Tabellen ablesen, bei denen der Algorithmus für beide a-Werte in der Laufzeit besser als der Naive Algorithmus skaliert.

In Tabelle B.2 sieht man jedoch auch, dass für kleine Eingaben der Unterschied zum Naiven Algorithmus teilweise nicht signifikant ist, der Laufzeitvorteil also nur für größere Eingaben zu trage kommt.

Laut der theoretischen Betrachtung sollte der Algorithmus mit $a = 4$ schneller sein als mit $a = 2$. Dies ist in der Messung für die meisten Eingaben der Fall, es gibt aber auch Ausnahmen bei denen der Unterschied gering oder sogar umgekehrt ausfällt. Am größten tritt dieser Effekt bei `pc_dna` und `dna.txt` auf, was darauf schließen lässt das bei kleinen Alphabetgrößen $a = 2$ effizienter zu sein scheint, bzw. die gemeinsamen Prefixe bei DNA kurz genug ausfallen das bei $a = 4$ die Iterationsschritte zu grob-granular sind und eindeutige Suffix-Index zu lange beibehalten werden.

Im Vergleich zum Naiven Algorithmus fällt jedoch auch auf, das es keinen signifikanten Unterschied in der Laufzeit bei verschiedenen a Werten gibt, weshalb man in Anbetracht des wesentlich größeren Speicherbedarfs bei $a = 4$ für In-Memory Implementierungen vermutlich besser beim normalen verdoppeln, also $a = 2$ bleiben sollte.

mSufSort

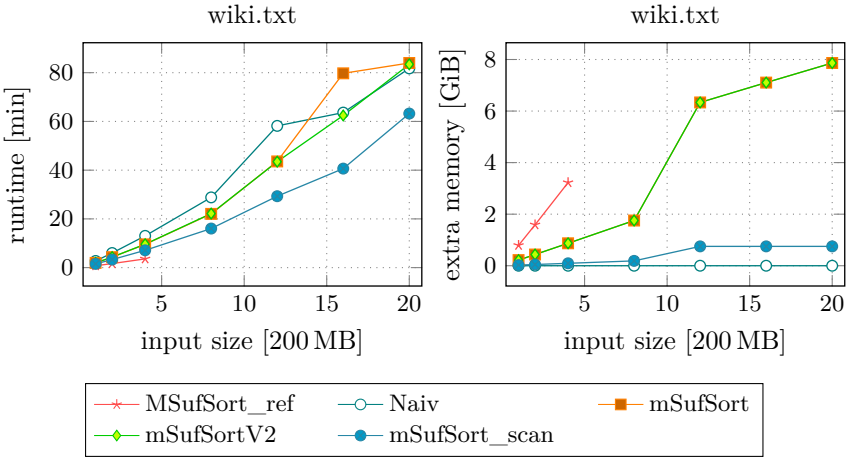


Abbildung 7.16: mSufSort Varianten auf wiki.txt

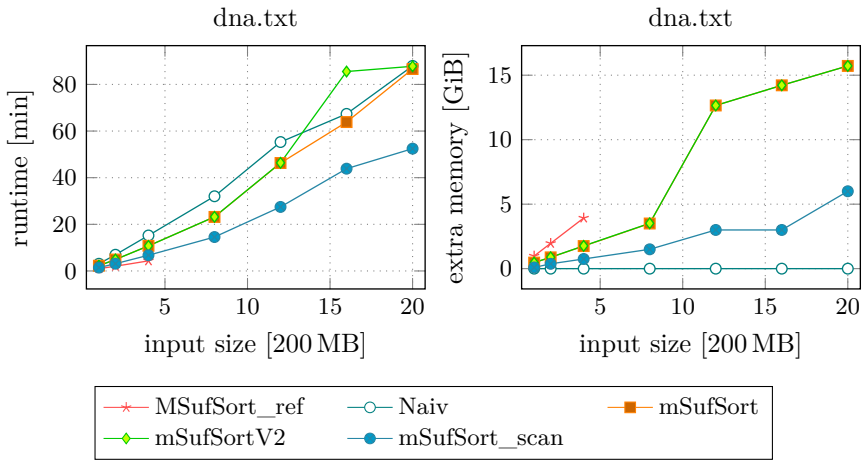


Abbildung 7.17: mSufSort Varianten auf dna.txt

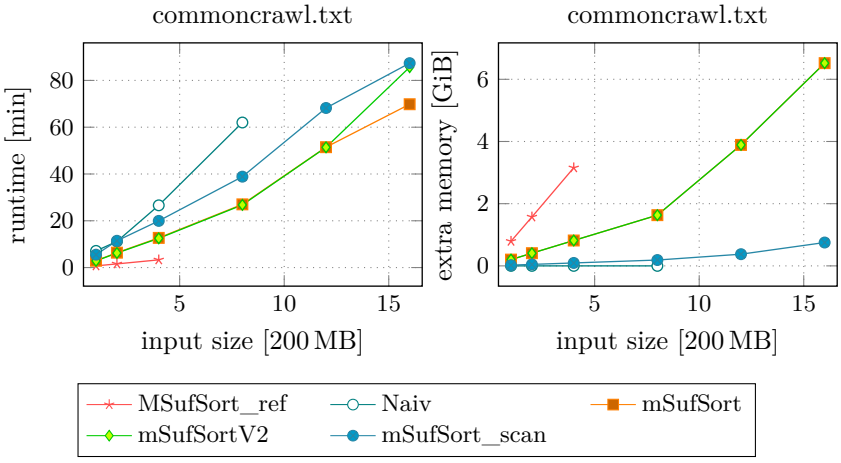


Abbildung 7.18: mSufSort Varianten auf commoncrawl.txt

Die Laufzeitmessung bei den Experimenten 7.16, 7.17 und 7.18 zeigt, dass jeweils der naive Algorithmus langsamer als mSufSort_scan und die Referenzimplementierung ist. Die beiden anderen Varianten, mSufSort und mSufSortV2 sind teilweise ähnlich schlecht wie der naive SACA und unterscheiden sich nur geringfügig voneinander. Auf allen drei Texten ist jedoch die Referenzimplementierung jeweils deutlich am schnellsten; diese funktioniert aber nur für bis zu 800 Megabyte Eingabegröße. In 7.16 und 7.17 ist von den drei eigenen mSufSort-Varianten die Scan-Variante am schnellsten. In 7.18 liegt die Scan-Variante etwa parallel nach oben versetzt zu den anderen Varianten, skaliert aber zumindest nicht schlechter und liegt näher an den normalen mSufSort Varianten als am naiven Sortierer.

Die Speichermessung ergibt bei allen drei Texten, dass die Scan-Variante von mSufSort den geringsten Speicherverbrauch hat. Die beiden anderen Varianten unterscheiden sich auch im Speicherverbrauch kaum. Die Referenzimplementierung hat auf allen Eingaben den größten Speicherverbrauch, insbesondere bei Commoncrawl im Vergleich zur Scan-Variante ist der Unterschied groß.

qSufSort

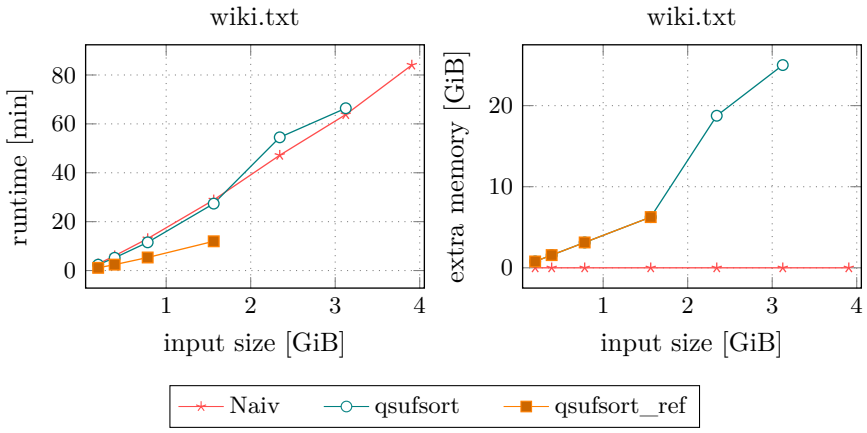


Abbildung 7.19: qSufSort auf wiki.txt

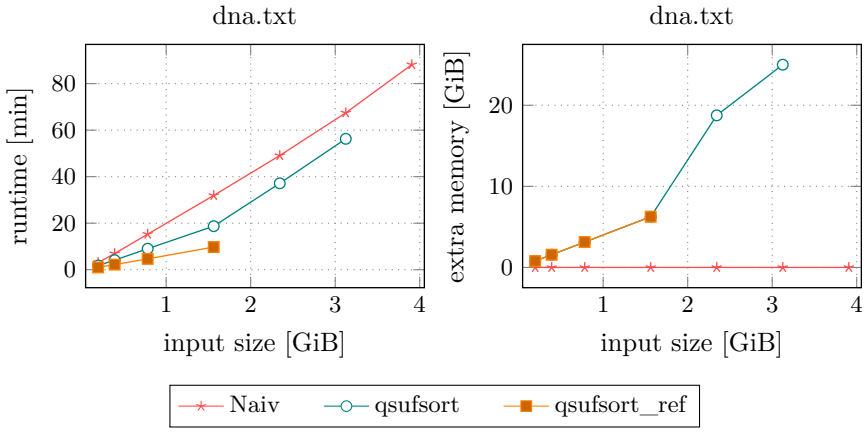


Abbildung 7.20: qSufSort auf dna.txt

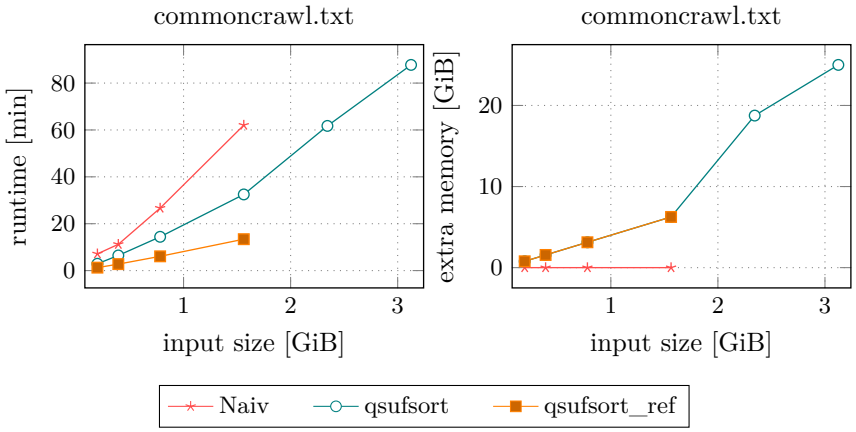


Abbildung 7.21: qSufSort auf commoncrawl.txt

Die Abbildungen 7.19, 7.20 und 7.21 zeigen jeweils die Vergleiche zwischen unserer Implementierung des qSufSort, der Implementierung der Autoren Larsson und Sadakane und des naiven SACAs.

Auf allen drei Texten zeichnet sich sowohl für das Laufzeitverhalten, als auch für den Speicherverbrauch ein ähnliches Bild ab. Da die Referenzimplementierung des qSufSort lediglich 32-Bit-Typen verwendet, können Eingaben ab 1,6GB nicht mehr verarbeitet werden.

Die Laufzeitmessung zeigt jedoch, dass bis zu diesem Punkt, die Referenzimplementierung den anderen beiden Algorithmen bezüglich der Laufzeit überlegen ist. Bei beiden Varianten des Prefix-Doublers lässt sich darüber hinaus beobachten, dass die benötigte Laufzeit nahezu linear mit der Eingabegröße steigt, also gut mit dieser skaliert.

Die Speichermessung zeigt, dass beide Varianten bis zu einer Dateigröße von 1,6GB den selben Speicherverbrauch haben. Das liegt daran, dass beide lediglich ein zusätzliches Hilfsarray nutzen, dass ausschließlich von der Größe der Eingabe abhängt. Gut zu erkennen ist zudem der deutlich erhöhte Speicherverbrauch unserer Implementierung durch die Nutzung von 64-Bit Datentypen für Eingabegrößen größer als 1,6GB.

Unsere Implementierung kann somit sowohl in Hinblick auf Speicher und Laufzeit mit der Referenzimplementierung konkurrieren. Zudem kann diese 64-Bit-Datentypen verwenden, wodurch auch größere Eingaben verarbeitet werden können. Dennoch ist die Referenzimplementierung, vor allem wegen ihrer guten Laufzeit, von Relevanz. Ein mögliche Verbesserung dieser bestehe darin, sie auf 64-Bit-Typen zu erweitern, um diese auch für Eingabegrößen über 1,6GB vergleichen zu können.

SAIS/SADS/SACA-K

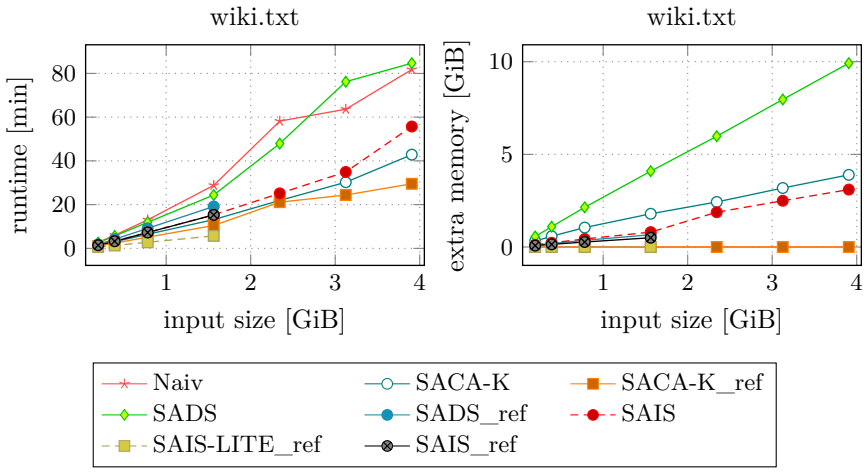


Abbildung 7.22: SAIS, SADS, SAIS-LITE und SACA-K auf wiki.txt

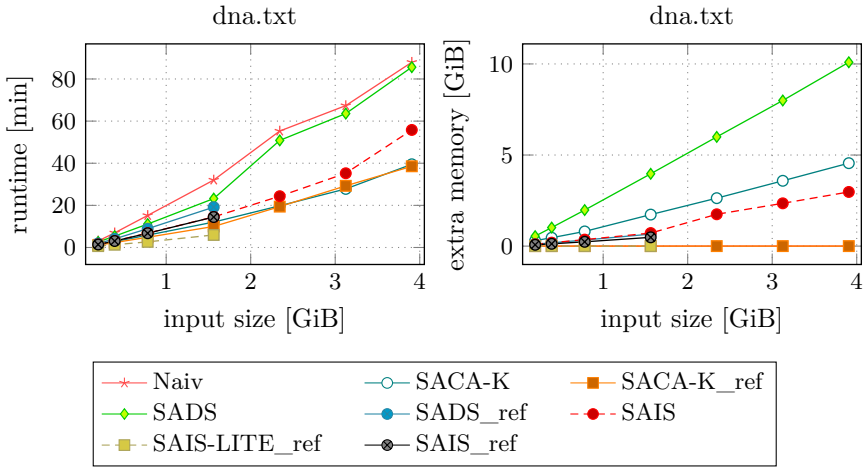


Abbildung 7.23: SAIS, SADS, SAIS-LITE und SACA-K auf dna.txt

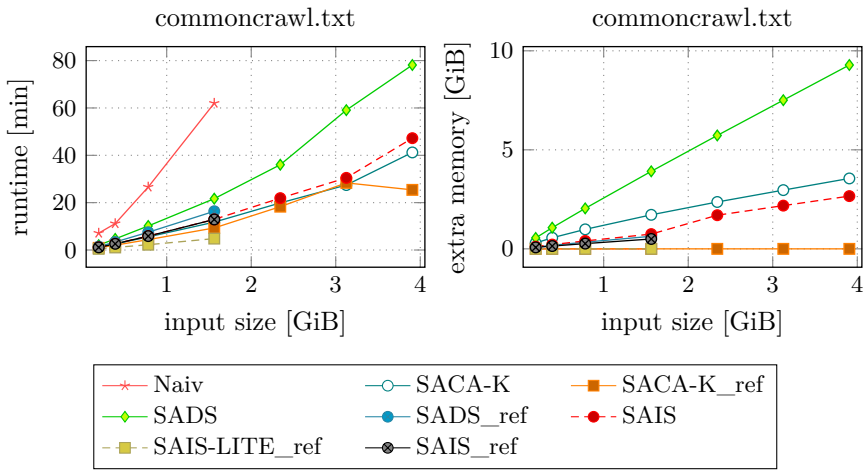


Abbildung 7.24: SAIS, SADS, SAIS-LITE und SACA-K auf commoncrawl.txt

Die Speichermessung zeigt die direkte Abhängigkeit zur Textgröße mit Ausnahme der Referenzimplementierung des SACA-K, da in den meisten Algorithmen der Suffixtyp jedes Zeichens im Eingabetext in ein eigenes Bit-Array gespeichert wird. Die Messungen zeigen deutlich den Trade-Off des SAIS-LITE_ref zwischen Laufzeit und Speicherplatz. Der Grund für die Ausnahme beim SACA-K ist, dass weder in den Haupt- noch den Rekursionsinstanzen des Algorithmus Extraspeicher abhängig von der Textgröße allokiert und immer wieder ausgelesen werden muss. Viele der Berechnungen finden *on-the-fly* statt und Pointer werden sehr schlau im Speicher des SAs zwischengespeichert. Leider ist genau diese letztere Methode noch nicht effizient in unserer Implementierung des SACA-K vorhanden, weshalb die Laufzeit zwar trotzdem besser ist als die der SAIS-Implementierung, aber beim Speicherverbrauch noch viel Optimierung nötig ist, da genau dies eine der wichtigsten Features des SACA-K ist.

Die Zeitmessung macht deutlich, dass das Laufzeitverhalten des SAIS-LITE_ref mit Abstand am besten ist. Grund dafür sind die Optimierungstechniken auf Bit-Ebene, die Yuta Mori bei seiner Implementierung verwendet. Gleichzeitig ist der naive Algorithmus auf allen Texten schlechter als die Referenz- und Eigenimplementierungen. Außerdem bestätigt sich die Beobachtung der Autoren, dass der SADS grundsätzlich schlechter als der SAIS ist. Da die Referenzalgorithmen auf 32 Bit-Typen basieren, schaffen diese nicht das Berechnen der Suffix-Arrays für eine Dateigröße bis zu 4 GB. Die Referenzimplementierung und unsere Implementierung des SACA-K schneiden bei den Laufzeitmessungen nach dem SAIS-LITE_ref am besten ab.

Difference-Cover

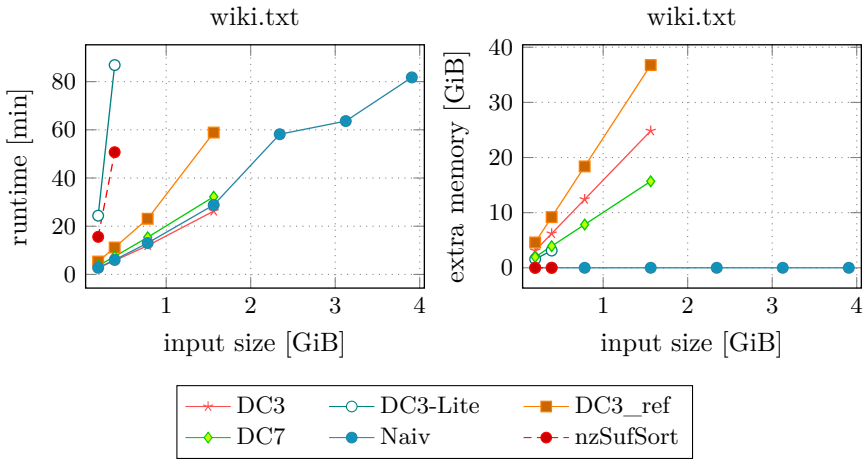


Abbildung 7.25: Algorithmen, die auf Difference Cover basieren, auf wiki.txt

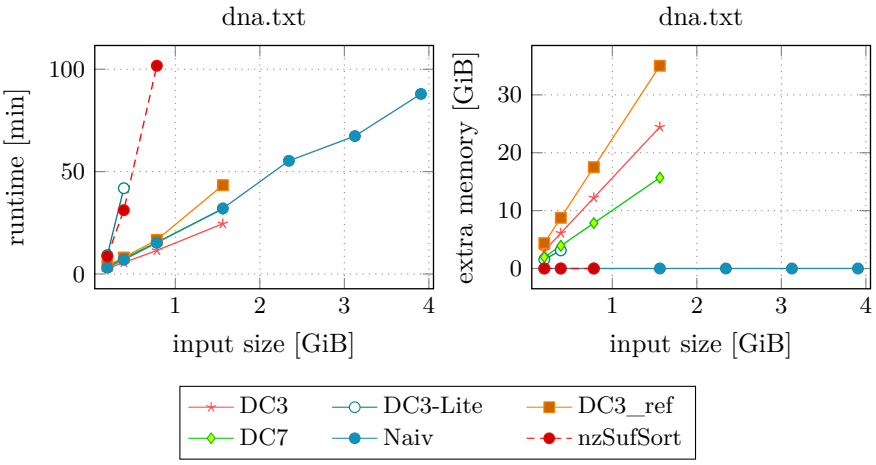


Abbildung 7.26: Algorithmen, die auf Difference Cover basieren, auf dna.txt

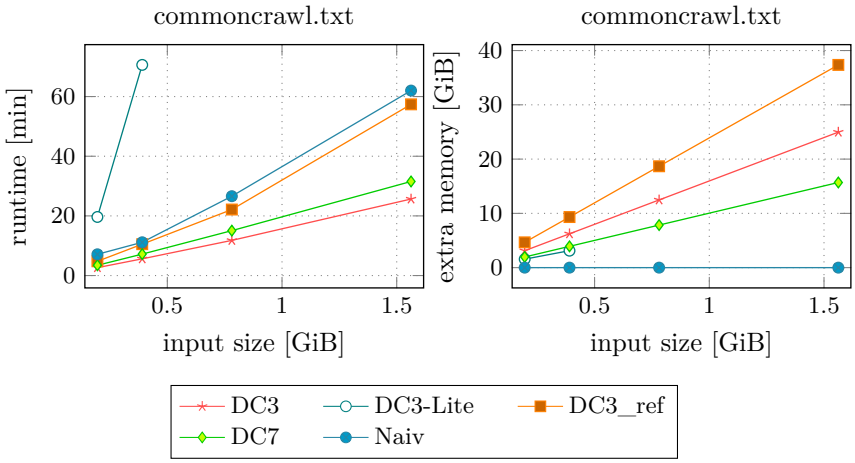


Abbildung 7.27: Algorithmen, die auf Difference Cover basieren, auf commoncrawl.txt

In den Abbildungen 7.25 bis 7.27 sind die Implementierungen, die in dem SACABench Framework umgesetzt worden sind, miteinander verglichen worden, die auf dem *Difference Cover* basieren.

Die Laufzeitmessungen zeigen, dass die Implementierung des DC3 der Projektgruppe als einziger Algorithmus den naiven Algorithmus bezüglich der Laufzeit auf allen Testdaten schlägt. Die Referenzimplementierungen des DC3 und des DC7 sind jedoch nur auf dem `commoncrawl.txt` besser als der Naive. Auf allen Eingabetexten belegen der nzSufSort und der DC3-Lite die letzten Plätze bezüglich der Laufzeit. Außerdem sieht man, dass der DC7 langsamer als der DC3 ist, da in der dritten Phase mehr Vergleiche stattfinden um die Mengen zu vereinigen.

Die Speichermessungen zeigen dafür, dass die Implementierungen des nzSufSort und DC3-Lite nur sehr geringen zusätzlichen Speicher zur Berechnung des Suffix-Arrays benötigen. Daher belegen die beiden Algorithmen nach dem naiven Algorithmus den zweiten und dritten Platz. Allerdings sind die Laufzeiten der beiden Implementierungen so schlecht, dass sie bereits vor 1000 MiB Eingabegröße das Zeitlimit des Messsystems überschreiten. Die Algorithmen DC3 und DC7 benötigen ab einer Größe von knapp 1600 MiB mehr Speicherplatz als auf dem Messsystem vorhanden, sodass sie ab dieser Eingabegröße ebenfalls abbrechen. Die Messungen zeigen ebenfalls, dass der Speicherverbrauch der Implementierung des DC3 der Projektgruppe geringer ist als der der Referenzimplementierung. Der DC7 benötigt sogar noch weniger Extra-Speicher. Dies liegt daran, dass die Rekursionstiefe des DC7 geringer ist als die des DC3.

DivSufSort

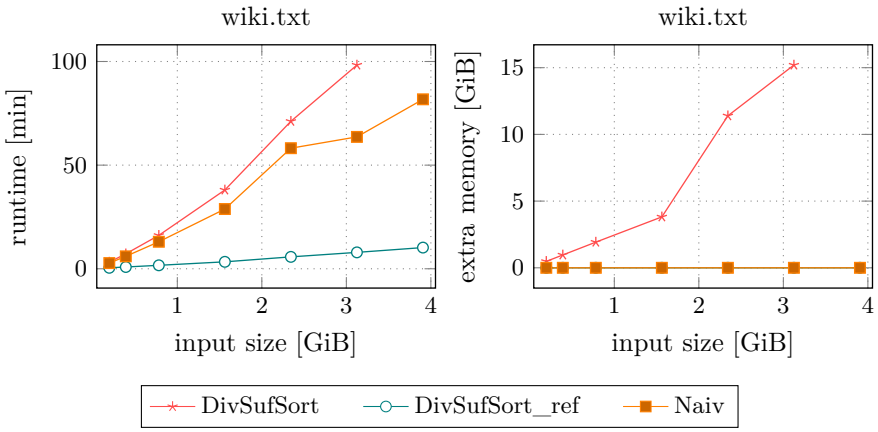


Abbildung 7.28: DivSufSort und DivSufSort_ref auf wiki.txt

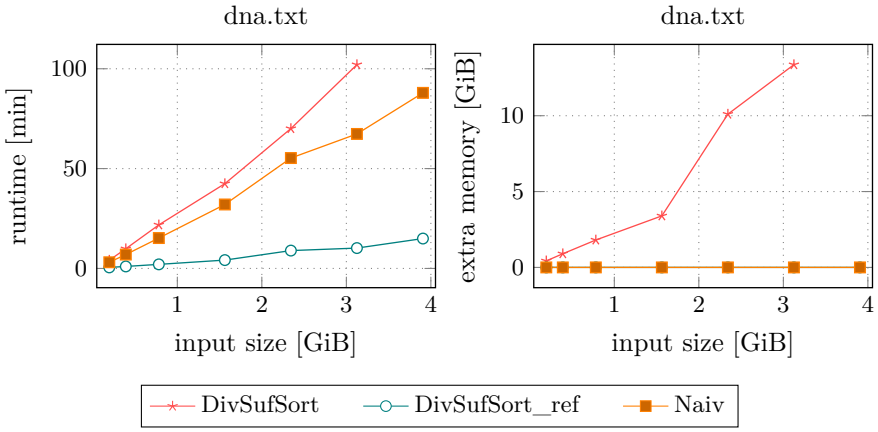


Abbildung 7.29: DivSufSort und DivSufSort_ref auf dna.txt

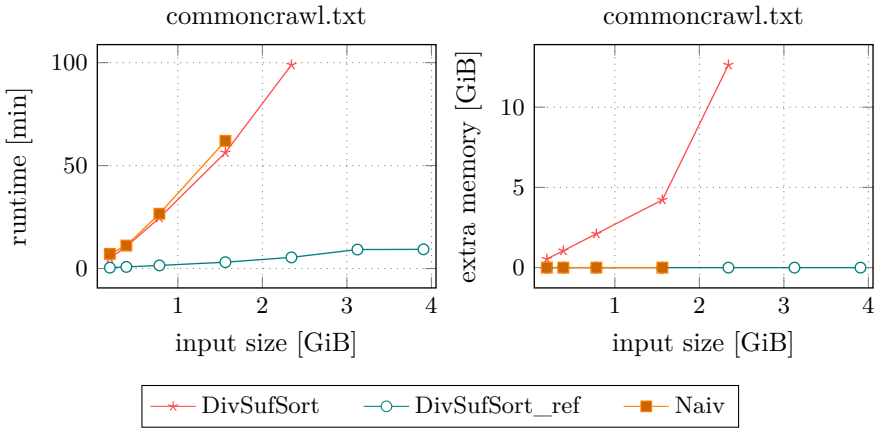


Abbildung 7.30: DivSufSort und DivSufSort_ref auf commoncrawl.txt

Für alle drei Eingabedateien werden bei der Referenzimplementierung des DivSufSort von Mori [39, 23] durch die 32-Bit Suffix-Index Typen ab einer Eingabegröße von 1,8 GiB keine Berechnungen durchführt. Bis 1,6 GiB hingegen ist er deutlich schneller als unsere Implementierung und der naive SACA. Bei allen drei Eingabedateien befindet sich die Berechnungsdauer im selben Minutenbereich. Extraspeicher braucht dieser nur in Abhängigkeit vom Alphabet, weshalb er in den Grafiken gleichauf mit dem Naiven ist. Die Laufzeit erhöht sich sowohl für den naiven SACA als auch für unsere Referenzimplementierung für die Eingabe `commoncrawl.txt`. So benötigen beide Algorithmen für 1,6 GiB mehr als 50 Min, während unser DivSufSort für die anderen beiden Eingaben etwas über 40 Min und der Naive etwa 30 Min brauchte. Unsere Implementierung ist nicht signifikant schneller als der naive Algorithmus und bricht bei einer Eingabegröße $> 2,4$ GiB ab.

Unsere Implementierung weist zwei deutliche Nachteile auf: Zum einen wächst die Berechnungsdauer proportional zur Referenzimplementierung, wobei die Dauer unserer Implementierung ab einer Eingabegröße von 2 GiB schneller ansteigt als bei vorigen Größen, bis die Berechnung aufgrund eines $2h$-Timeouts beendet wird. Zum anderen wächst der Speicher in Abhängigkeit von der Eingabegröße, was bei der Referenzimplementierung nicht der Fall ist. Der Sprung im Speicherbedarf ab einer Größe von 2,4 GiB deutet darauf hin, dass die Anzahl der RMS-Substrings überproportional ansteigt. Beide Ursachen lassen sich durch die verwendete Sortierung der RMS-Substrings sowie durch die Sortierung der RMS-Suffixe erklären. Bei der Sortierung der Substrings wurde auf die Kombination aus Multikey-Quicksort und Introsort verzichtet und stattdessen der Introsort mit einem zeichenweisen Vergleich gewählt. Für diesen Vergleich wurden dafür zunächst die jeweiligen Teilstrings erzeugt und in die Vergleichsfunktion integriert. Der Speicherverbrauch kann reduziert werden, indem die Eigenschaft ausgenutzt wird, dass zu diesem Zeitpunkt die Suffix-Indizes in Textreihenfolge

in $\mathsf{SA}[n-m, n)$ vorliegen. Damit können die RMS-Teilstrings durch eine geeignete Vergleichsfunktion simuliert werden, ohne zusätzlichen Speicher zu benötigen. Die Sortierung der RMS-Suffixe wurde naiv umgesetzt, indem erst die Suffixe nach ihren Rängen mittels Quicksort sortiert werden und daraufhin die Ränge neuberechnet werden. Dies wird solange wiederholt, bis alle RMS-Suffixe eindeutig sortiert wurden. Bei einer größeren Anzahl von RMS-Suffixen steigt somit auch die Laufzeit des Algorithmus. Ein Durchlauf der RMS-Suffix Sortierung speichert alle partiellen ISA-Werte zwischen, um Abhängigkeiten durch die naive Neuzuweisung der Ränge aufzulösen. Dies ermöglicht, die Ränge für alle RMS-Suffixe im naiven Ansatz korrekt zu berechnen. Eine nähere Implementierung an der Referenz würde diese Abhängigkeiten auflösen und sowohl den Speicherbedarf als auch die Laufzeit, ebenfalls ohne Repetition-Detection, reduzieren.

GSACA

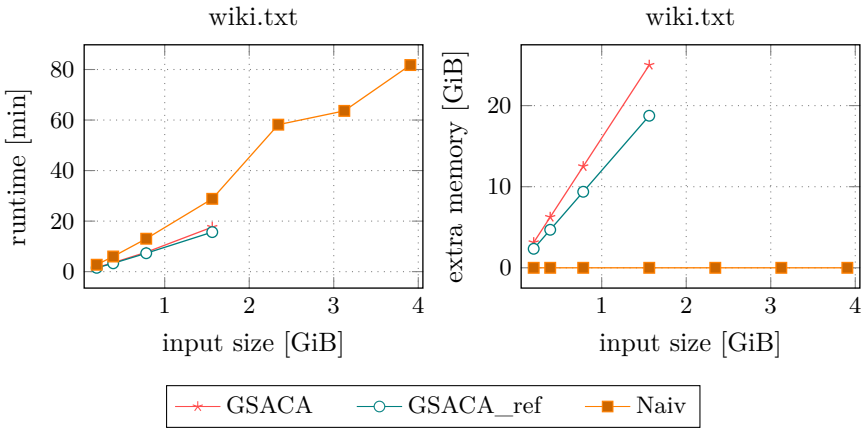


Abbildung 7.31: GSACA und GSACA_ref auf wiki.txt

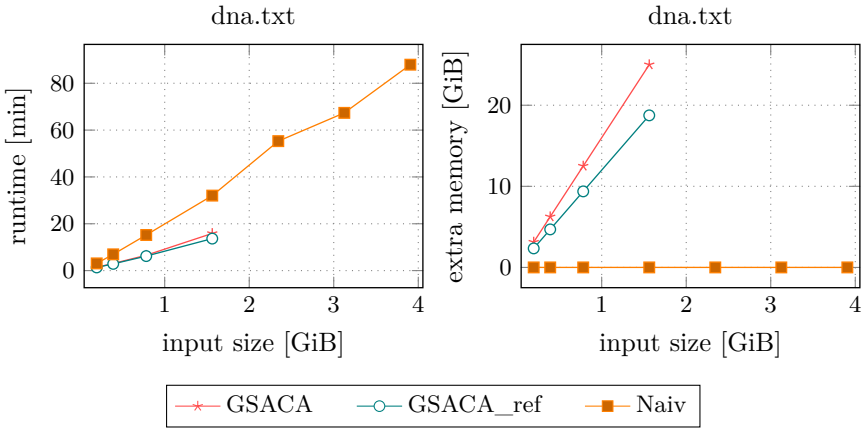


Abbildung 7.32: GSACA und GSACA_ref auf dna.txt

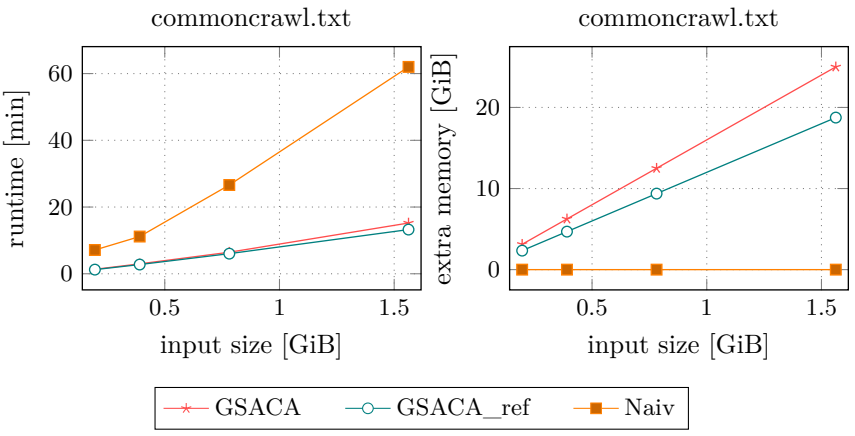


Abbildung 7.33: GSACA und GSACA_ref auf commoncrawl.txt

Die Abbildungen 7.31, 7.32 und 7.33 stellen den Suffix-Array-Konstruktionsalgorithmus GSACA der Referenzimplementierung und dem naiven SACA gegenüber.

Die Laufzeitmessungen zeigen, dass GSACA und die Referenzimplementierung von Uwe Baier bei allen drei Eingabetexten schneller als der naive Algorihtmus sind. Dabei zeigt sich der größte Unterschied auf dem Text `commoncrawl.txt`, bei dem sich die Laufzeiten bei einer Eingabegröße von 1,6 GiB um etwa 40 Minuten zu dem naiven Algorithmus unterscheiden. Sowohl bei dem Text `wiki.txt` als auch bei `dna.txt` liegt die Laufzeit der drei SACAs näher beieinander.

Die Speichermessungen lassen erkennen, dass GSACA und die Referenzimplementierung einen höheren Speicherbedarf haben, als der naive Algorithmus. Dieser ist bei GSACA wiederum größer als bei der Referenzimplementierung. Auf allen drei Eingabetexten brechen sowohl GSACA als auch GSACA_ref ab einer Eingabegröße von 1,6 GiB ab. Wie in Tabelle B.4 ersichtlich ist, liegt dies dadran, dass GSACA das Speicherlimit des Systems überschreitet.

Die alternative Variante von GSACA, welche in Abschnitt 6.14.7 beschrieben wurde, ist hingegen in keinem der Diagrammen aufgeführt. Dies liegt an der verlängerten Laufzeit, welche durch den zusätzlichen Aufwand in der Berechnung der Werte von GSIZE entsteht. Hierdurch schaffte es diese Variante nicht, in der vorgegebenen maximalen Zeit das Suffix-Array zu berechnen.

7.6 Ergebnisse Parallele SA Konstruktion

7.6.1 Suffix-Array Korrektheit

Wie auch bei den sequentiellen Experimenten wurden alle Ergebnisse der parallelen Algorithmen mittels SA-Checker auf Korrektheit überprüft. Hierbei wurden die Experimente mit Weak Scaling durchgeführt. Beim Weak Scaling werden dabei die Anzahl der Threads um 1 und die Eingabegröße um 200 MiB gleichzeitig erhöht. Dabei kann man erkennen, welche Algorithmen gut skalieren, da sich bei diesen bei Verwendung von mehr Threads auf einer größeren Eingabe die Laufzeit nur geringfügig verändert. Die Ergebnisse dieser Tests sind in Anhang B.2.1 (Tabelle B.7) gelistet.

Man sieht in der Tabelle, dass viele unserer parallelen Implementierungen das korrekte SA berechnen. Dennoch brechen einige Algorithmen bei bestimmten Eingaben ab, wenn ein Algorithmus zu viel Speicher benötigt, das Zeitlimit des Systems überschreitet oder wenn ein Laufzeitfehler auftritt.

Speicherfehler treten unter anderem beim DC3-Parallel-V1 und Discarding2-Parallel ab einer Eingabegröße von 2400 MiB auf. Der Osipov_parallel_wp benötigt sogar ab 1600 MiB zu viel Speicher.

Das Zeitlimit des Systems wird beim PARALLEL_SAIS auf `wiki.txt` ab 3200 MiB überschritten.

Laufzeitfehler treten beim PARALLEL_SAIS auf allen drei Eingabedateien ab einer Größe von 2400 MiB auf und der Deep-Shallow_par bricht auf der Datei `commoncrawl.txt` ab einer Größe von 1600 MiB ab.

7.6.2 Vergleich der parallelen CPU-Implementierungen

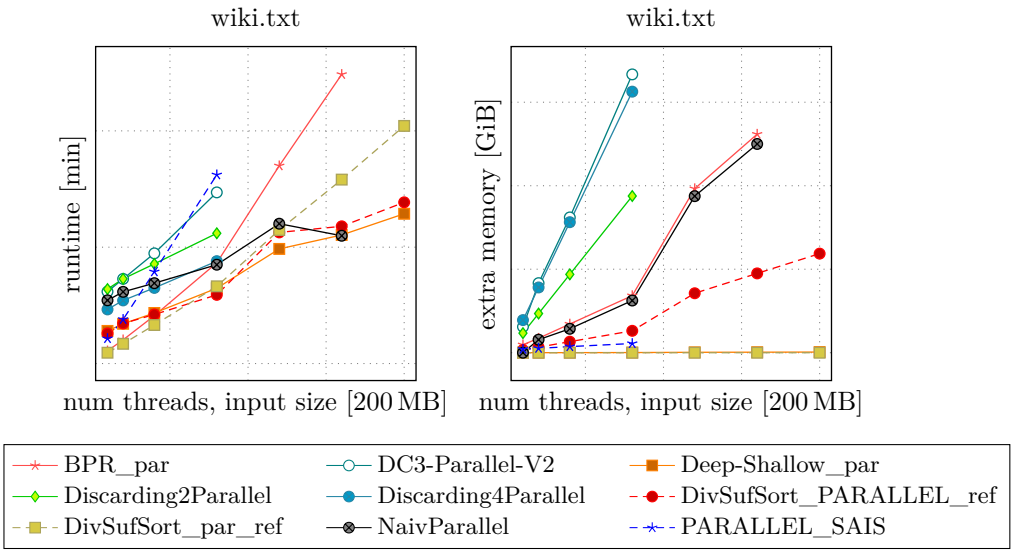


Abbildung 7.34: Vergleich der parallelen Implementierungen mit Laufzeit und Speicherbedarf auf wiki.txt

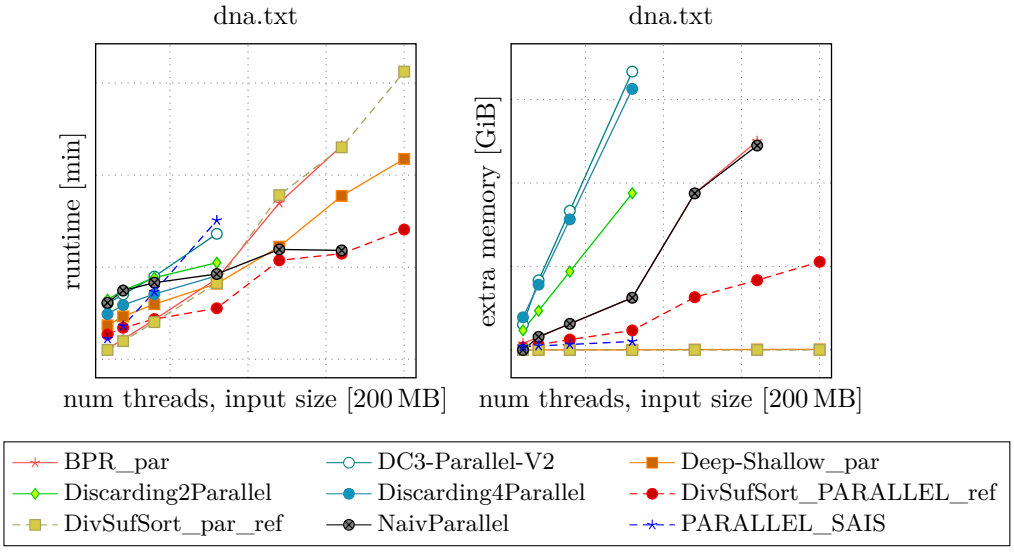


Abbildung 7.35: Vergleich der parallelen Implementierungen mit Laufzeit und Speicherbedarf auf dna.txt

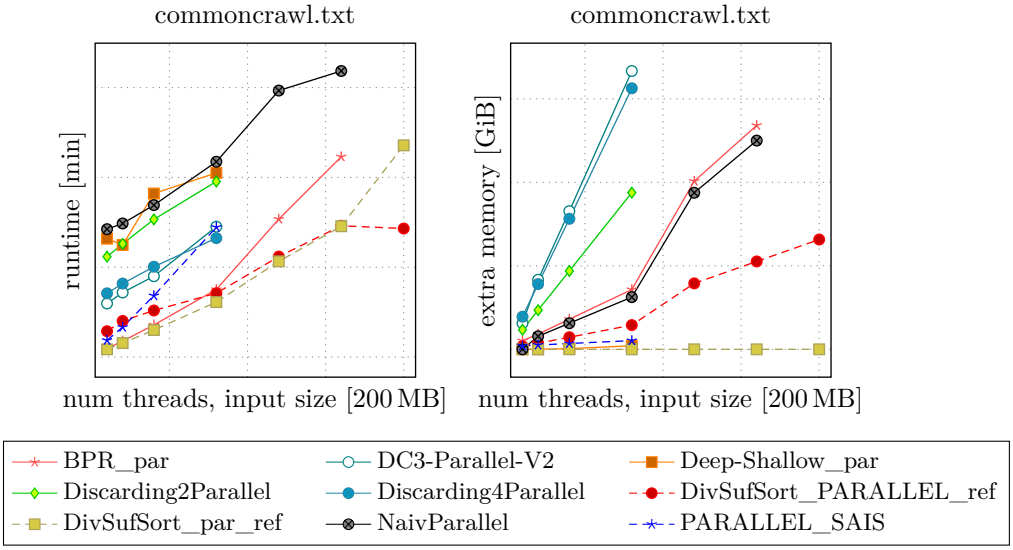


Abbildung 7.36: Vergleich der parallelen Implementierungen mit Laufzeit und Speicherbedarf auf commoncrawl.txt

In den Abbildungen 7.34 bis 7.36 sieht man die Ergebnisse des parallelen Vergleichs mit Weak Scaling auf den Eingabedateien `wiki.txt`, `dna.txt` und `commoncrawl.txt`. In diesem Abschnitt werden die parallelen Implementierungen untereinander verglichen, während im darauffolgenden Abschnitt die Ergebnisse für die einzelnen Implementierungen erläutert werden.

Die Laufzeitmessungen zeigen, dass auf der Eingabe `wiki.txt` der Deep-Shallow_par, der DivSufSort_par_ref und der DivSufSort_PARALLEL_ref am besten skalieren. Am schlechtesten skalieren der PARALLEL_SAIS, der BPR_par und der DC3-Parallel-V2.

Auf der Eingabe `dna.txt` kann man erkennen, dass hier der Discarding2Parallel, der Discarding4Parallel und der NaivParallel am besten skalieren. Nicht gut skalieren der DC3-Parallel-V2, der PARALLEL_SAIS, der BPR_par und der Deep-Shallow_par.

Auf `commoncrawl.txt` lässt sich beobachten, dass der BPR_par, der DivSufSort_par_ref und der DivSufSort_PARALLEL_ref am besten skalieren, während der Deep-Shallow_par, der NaivParallel und der Discarding2Parallel am schlechtesten skalieren.

Die Speichermessungen zeigen, dass sich die Algorithmen auf allen drei Dateien bezüglich des Speichers ähnlich verhalten. Die drei Algorithmen, die am wenigsten Speicher benötigen, sind der Deep-Shallow_par, der PARALLEL_SAIS und der DivSufSort_par_ref und die drei, die am meisten Speicher benötigen, sind der Discarding4Parallel, der DC3-Parallel-V2 und der Discarding4Parallel. Auffällig ist hierbei, dass der BPR_par und der NaivParallel ab einer Eingabe von 1600 MiB und 8 Threads auf einmal deutlich mehr Speicher benötigen.

7.6.3 Weak Scaling

BPR

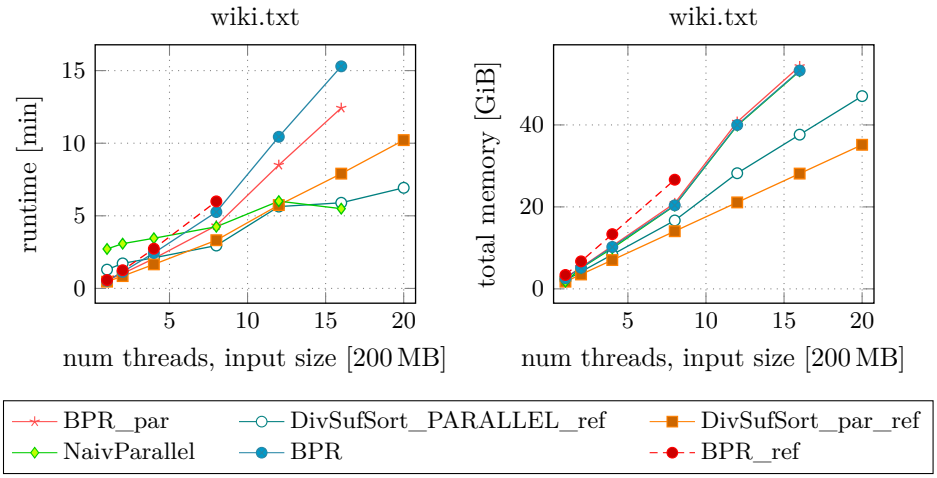


Abbildung 7.37: BPR_par und Vergleichsalgorithmen auf wiki.txt

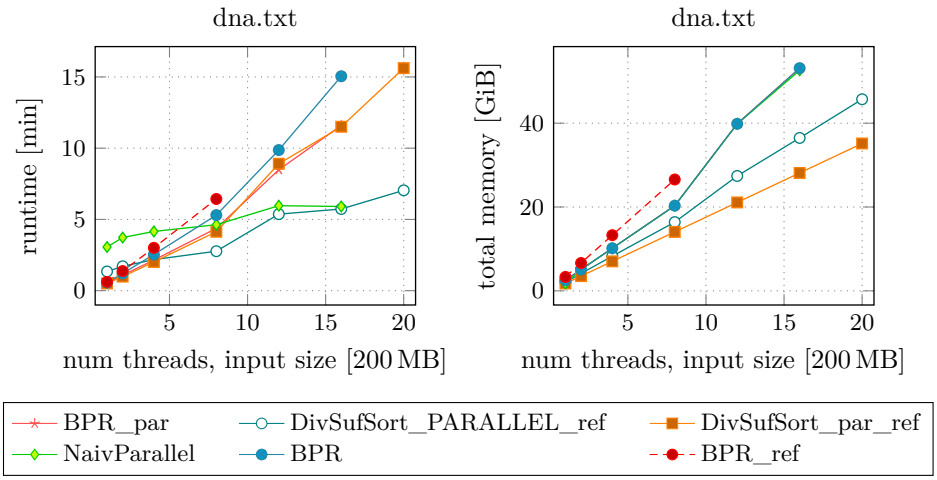


Abbildung 7.38: BPR_par und Vergleichsalgorithmen auf dna.txt

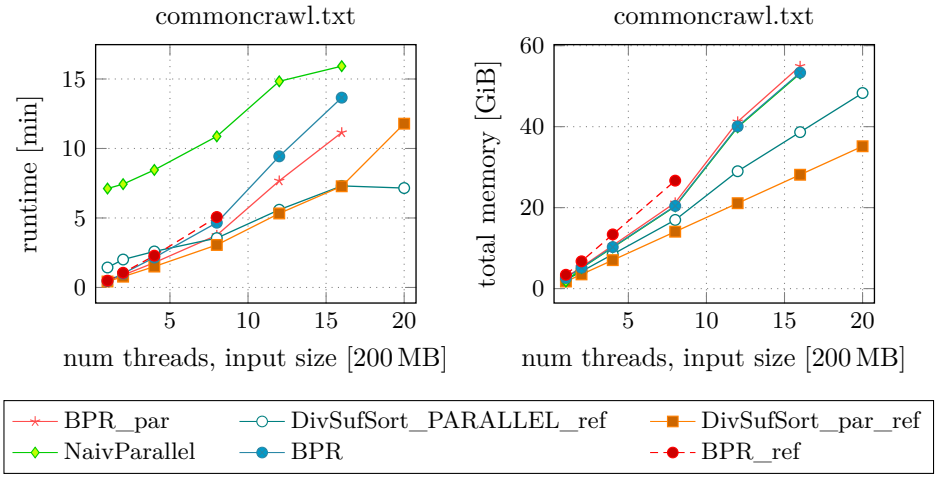


Abbildung 7.39: BPR_par und Vergleichsalgorithmen auf commoncrawl.txt

Die Speichermessung erfolgt analog zur Auswertung des sequentiellen *Bucket-Pointer Refinement* auch beim Vergleich der parallelen Implementierungen in den Abbildungen 7.37 bis 7.39 bezüglich des gesamten Speicherbedarfs inklusive Eingabe- und Ausgabe-Array eingeganben, um einen Vergleich zur sequentiellen Referenzimplementierung zu ermöglichen, welche die Darstellung differenzierter Speicherwerte nicht zulässt.

Wie zu erwarten war, ist in allen Fällen nur ein geringer Unterschied zwischen der sequentiellen und der parallelen Implementierung des BPR zu erkennen. Da der Algorithmus, wie in Abschnitt 6.6.4 näher erläutert, zu großen Teilen nicht parallelisiert werden konnte, ergibt sich durch die getroffenen Maßnahmen nur ein geringer Vorteil gegenüber der sequentiellen Implementierung. Da der in den Abbildungen 7.37 und 7.39 erkennbare leicht gesteigerte Speicherbedarf von der Anzahl der Kerne sowie der Größe des Eingabealphabets abhängig ist, ist außerdem zu erwarten, dass es im Worst-Case bei besonders großen Eingabealphabeten zu einem deutlich erhöhten Speicherbedarf kommt.

Die Laufzeitmessung zeigt, dass die Laufzeit des Algorithmus auf `wiki.txt` und `dna.txt` (Abbildungen 7.37 und 7.38) schon bei Eingaben mittlerer Anzahl von Threads die des naiven parallelen Algorithmus übersteigt, was außerdem belegt, dass *Bucket-Pointer Refinement* parallel nicht gut mit der Anzahl der Kerne skaliert. Ferner ist zu erkennen, dass die parallele Version des BPR in Bezug auf die Laufzeit keinen eindeutigen Vorteil gegenüber der sequentiellen Version erzielen konnte.

DC3

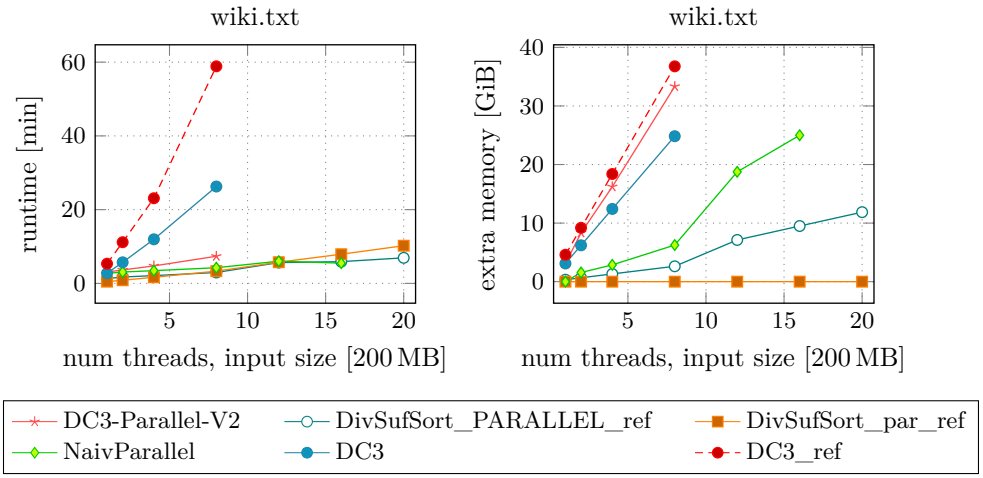


Abbildung 7.40: Paralleler DC3 und Vergleichsalgorithmen auf wiki.txt.

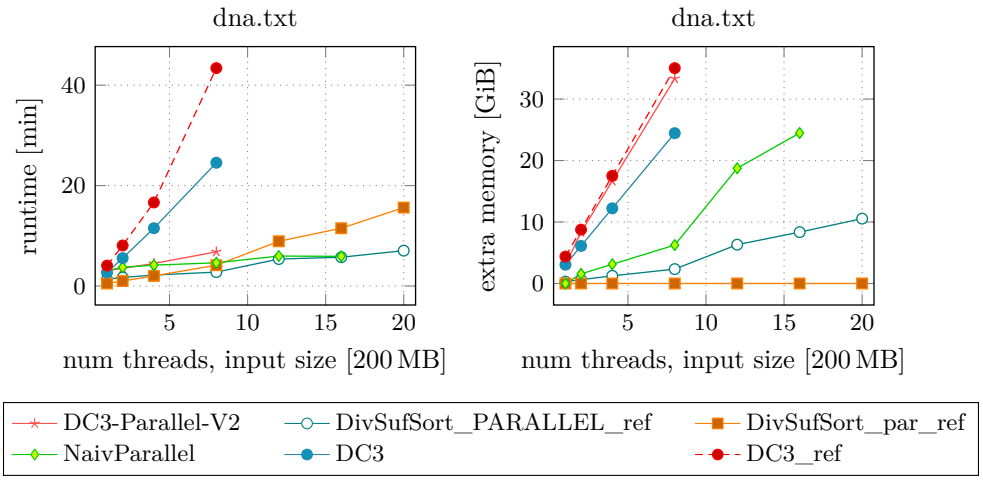


Abbildung 7.41: Paralleler DC3 und Vergleichsalgorithmen auf dna.txt

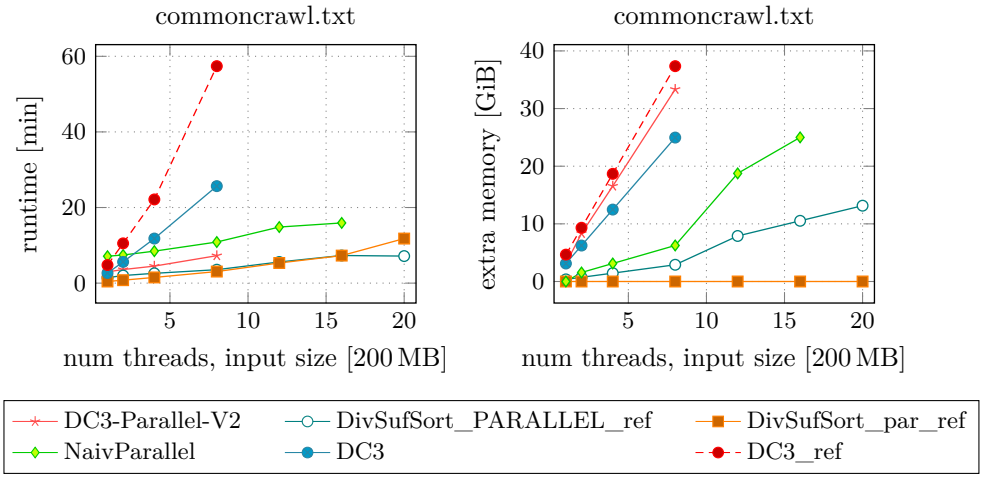


Abbildung 7.42: Paralleler DC3 und Vergleichsalgorithmen auf commoncrawl.txt

In den Abbildungen 7.40,7.41 und 7.42 sind die Implementierungen, die in dem SACABench Framework umgesetzt worden sind, miteinander verglichen worden, die auf dem *Difference Cover* basieren. Dabei ist der DC3-Parallel-V2 die Variante des parallelen DC3 mit naiver Parallelisierung der ersten beiden Phasen und die Implementierung des siebten Theorems, das in Abschnitt 6.4.4 näher erläutert worden ist, für die dritte Phase.

Die Laufzeitmessungen zeigen, dass die Laufzeiten des DC3-Parallel-V2 auf allen getesteten Eingaben größer als die der beiden parallelen DivSufSort-Varianten sind. Auf dem Testdatensatz `commoncrawl.txt` ist der DC3-Parallel-V2 aber schneller als der naive parallele Sortierer und belegt den dritten Platz vor dem naiv parallelen Algorithmus, dem DC3 und dem DC3_ref. Es ist zu sehen, dass der DC3-Parallel-V2 bezüglich der Laufzeit gut skaliert.

Die Speichermessungen zeigen, dass der zusätzliche Speicherverbrauch des DC3-Parallel-V2 zwar größer als bei dem sequentiellen DC3 ist, aber besser als der Speicherverbrauch der sequentiellen Referenzimplementierung des DC3. Dass die Algorithmen DC3, DC3_ref und DC3-Parallel-V2 ab der Eingabegröße von 2400 MiB abbrechen, liegt daran, dass das Speicherlimit des Messsystems mit diesen Algorithmen überschritten wird.

Deep-Shallow

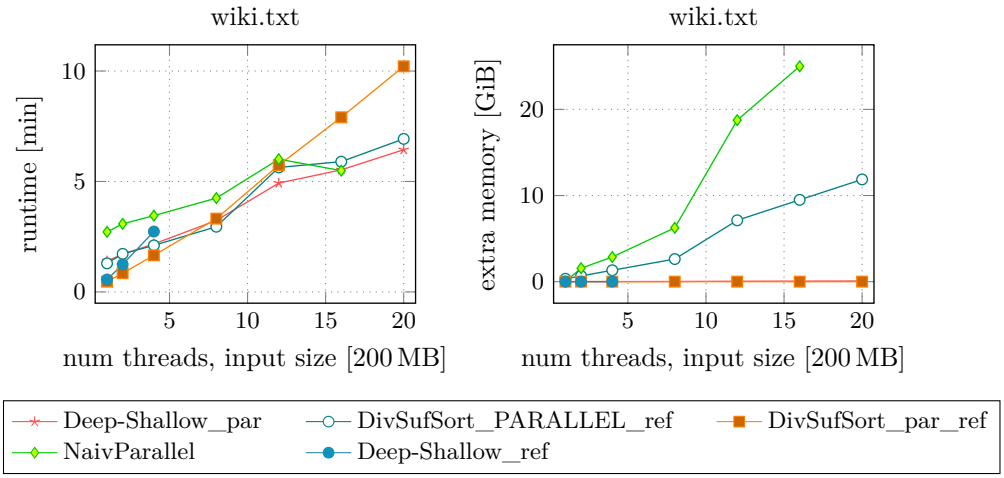


Abbildung 7.43: Deep-Shallow_par, Deep-Shallow_bb und Deep-Shallow_ref auf wiki.txt

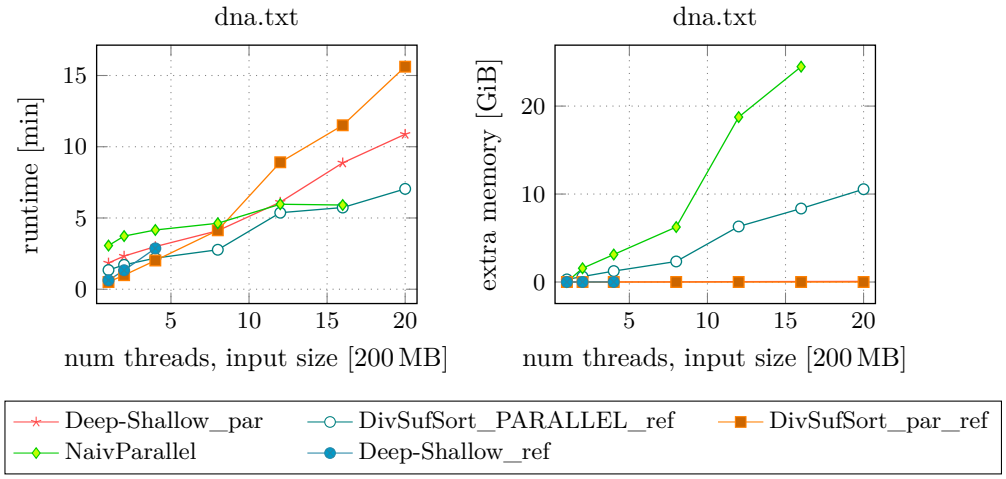


Abbildung 7.44: Deep-Shallow_par, Deep-Shallow_bb und Deep-Shallow_ref auf dna.txt

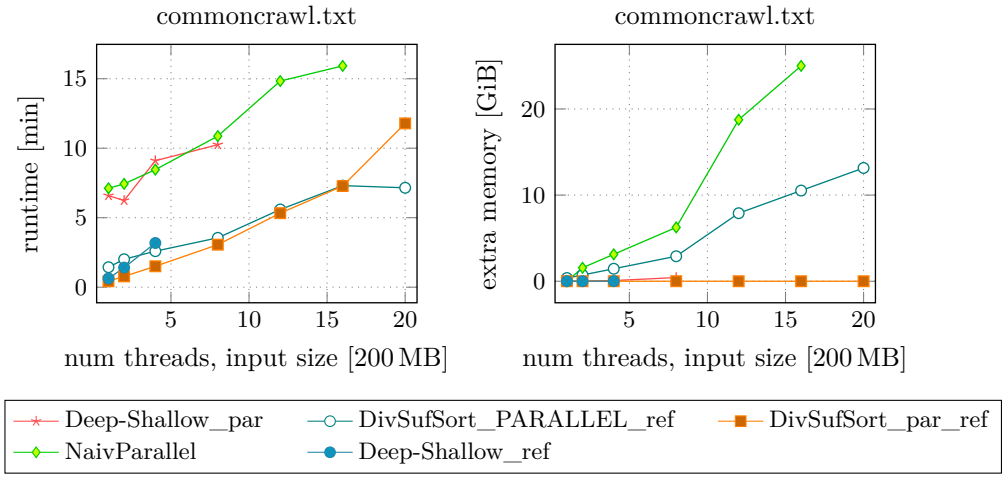


Abbildung 7.45: Deep-Shallow_par, Deep-Shallow_bb und Deep-Shallow_ref auf commoncrawl.txt

Die Speichermessung zeigt , dass unser paralleler Deep-Shallow keinen Extraspeicher verwendet. Daher ist er bezügliches des Speichers immer besser als die anderen verglichenen Algorithmen.

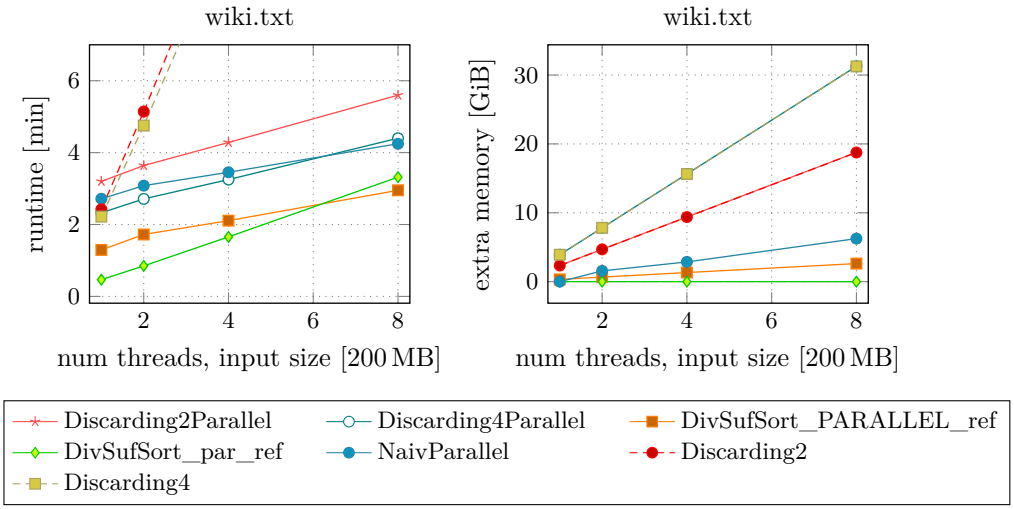
Die Laufzeitmessung ist hierbei interessanter. Auf dem ersten Bild bezüglich *wiki.txt* lässt sich erkennen, dass der parallele Deep-Shallow zwar erst von der Referenz und beiden parallelen DivSufSorts geschlagen wird, ab einer bestimmten Kernanzahl (12) der parallele Deep-Shallow alle Vergleichsalgorithmen dominiert.

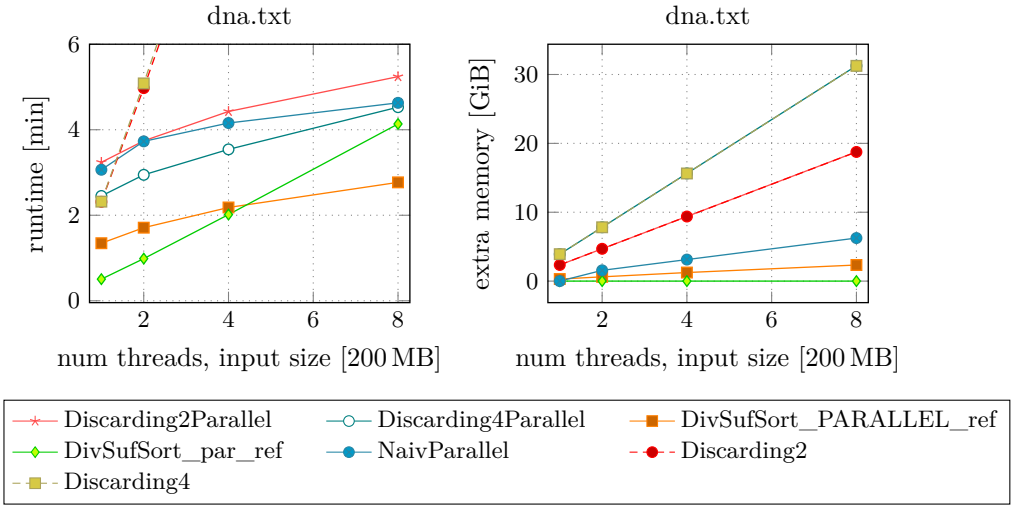
In der zweiten Abbildung (*dna.txt*) zeichnet sich ein anderes Bild. Aufgrund der geringen Alphabetgröße können nur wenige Kerne benutzt werden, wodurch die im Weak-Scaling-Experiment erhöhte Textgröße schwer wiegt und die Laufzeit signifikant erhöht. Dort wird unser Algorithmus von beiden parallelen DivSufSorts und mit höherer Kernzahl auch vom naiv-parallelen geschlagen.

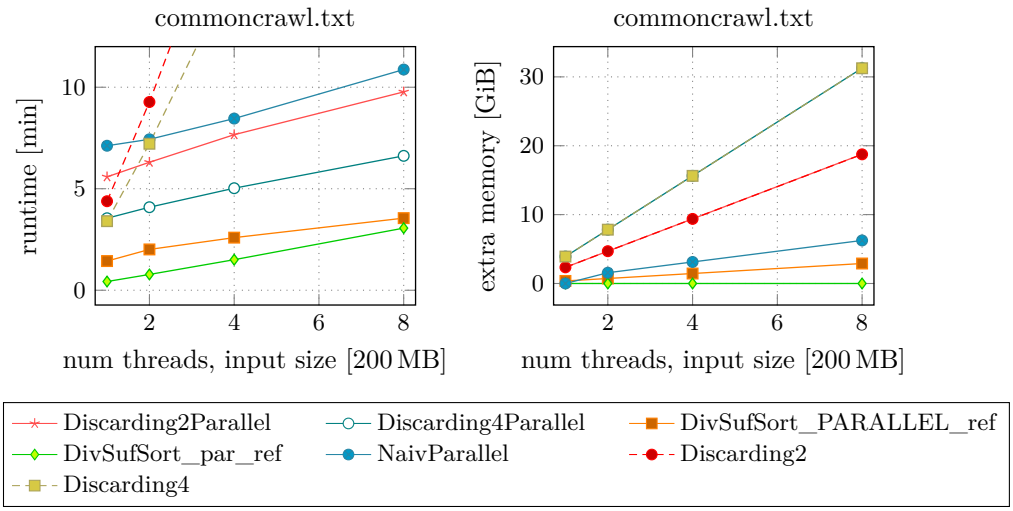
Auf dem dritten Bild (*commoncrawl.txt*) ist erkennbar, dass sich zu Beginn der naive SACA und unser Algorthmus prinzipiell in ihrer Performance sehr ähnlich sind, paralleles Deep-Shallow allerdings ab 1600 MB das SA aufgrund von Segfaults nicht mehr konstruieren kann. Leider war es uns im Zeitrahmen der Projektgruppe jedoch nicht mehr möglich diesen Fehler noch zu beheben.

Insgesamt lässt sich daher sagen, dass die Skalierung auf `wiki.txt` zwar adäquat ist, die Unzuverlässigkeit jedoch inakzeptabel. Die Performance auf zwei der drei Texten ist nicht zufriedenstellend.

Discarding


wiki.txt
runtime [min]
num threads, input size [200 MB]
wiki.txt
extra memory [GiB]
num threads, input size [200 MB]
Discarding2Parallel
Discarding4Parallel
DivSufSort_PARALLEL_ref
DivSufSort_par_ref
NaivParallel
Discarding2
Discarding4


dna.txt
runtime [min]
num threads, input size [200 MB]
dna.txt
extra memory [GiB]
num threads, input size [200 MB]
Discarding2Parallel
Discarding4Parallel
DivSufSort_PARALLEL_ref
DivSufSort_par_ref
NaivParallel
Discarding2
Discarding4



Wir werten nun das Verhalten der zwei parallelen *Discarding* Varianten des *Doubling*-Algorithmus mit a-Tupling für $a = 2$ und $a = 4$ (siehe Abschnitt 6.3.12) bei skalierender Eingabegröße und Thread-Anzahl aus. Allgemein sind die Ergebnisse ähnlich zu denen der sequentiellen Messung aus Abschnitt 7.5.4, da die einzige Änderung am Algorithmus die Nutzung eines parallelen Sortieralgorithmus ist, und sich dies auf alle Varianten des Algorithmus gleich auswirkt.

Die Speichermessung zeigt also ebenfalls, dass der Speicherverbrauch der Algorithmen rein von der Länge der Eingabe und der Größe von `sa_index` abhängt. Dies ist erneut aus den Diagrammen und Tabellen B.9 und B.12 ersichtlich.

Es gilt also ebenfalls, dass das Algorithmusverhalten mit der Implementierung (Abschnitt 6.3.9) übereinstimmt. Ebenso hat der Algorithmus wieder bei zu großer Eingabe einen zu hohen Speicherverbrauch, wie man in Tabelle B.9 erkennt.

Die Laufzeitmessung zeigt im Gegensatz zum sequentiellen Fall (siehe Abschnitt 7.5.4) einen deutlichen Unterschied zwischen a-Tupling mit $a = 2$ und $a = 4$, wobei letzteres schneller ist. Dies deutet darauf hin, dass der parallele IPs^{4}o Sortierer stärker auf die dadurch verursachten Unterschiede in der Anzahl und Größe der zu sortierenden Elemente reagiert als der sequentielle.

Dies führt jedoch auch dazu, dass *Discarding* mit $a = 2$ je nach Eingabe nah am parallelen Naiven Algorithmus liegt und teilweise sogar langsamer ist. Im Gegensatz zum sequentiellen Fall wird das SA also mit $a = 4$ wesentlich schneller berechnet.

Verglichen zum parallelen *DivSufSort* fällt auf, dass der Algorithmus ihn zwar in den erfolgreich gemessenen Bereich bis 8 Kerne nicht schlägt, aber eine

Extrapolation der Laufzeit andeutet dass dies bei höheren Kernanzahlen geschehen kann. *Discarding* scheint somit besser zu Skalieren.

pSAIS

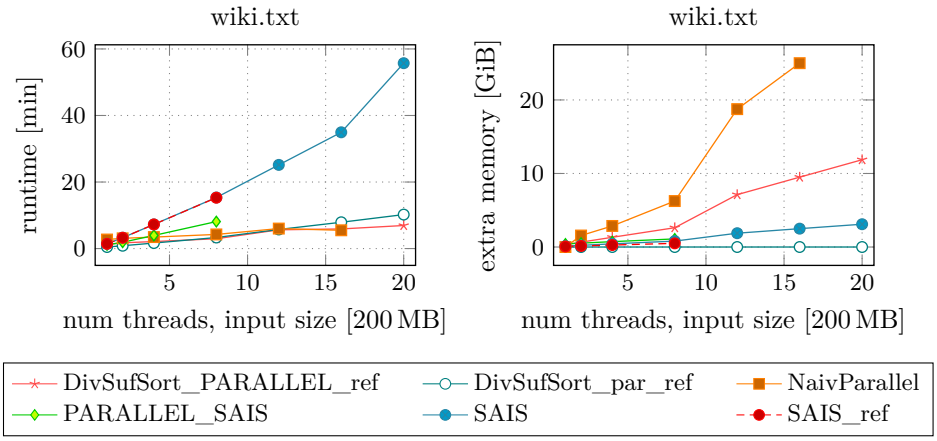


Abbildung 7.46: pSAIS verglichen mit anderen parallelen und nicht-parallelen Algorithmen auf wiki.txt

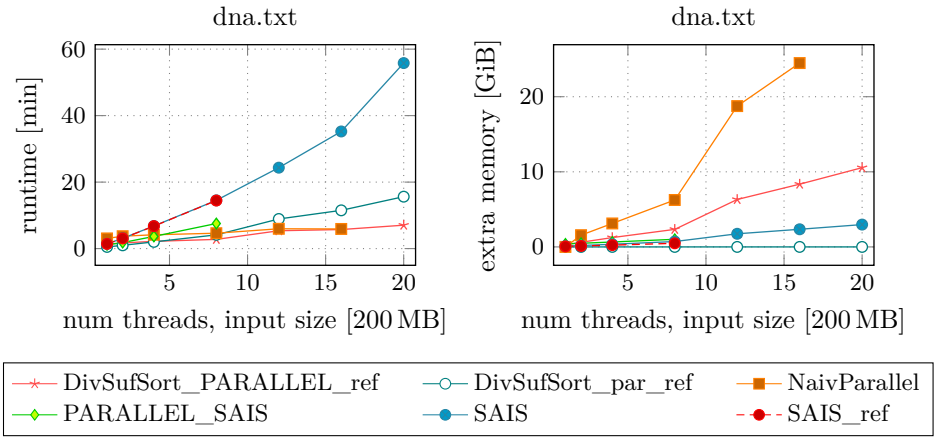


Abbildung 7.47: pSAIS verglichen mit anderen parallelen und nicht-parallelen Algorithmen auf dna.txt

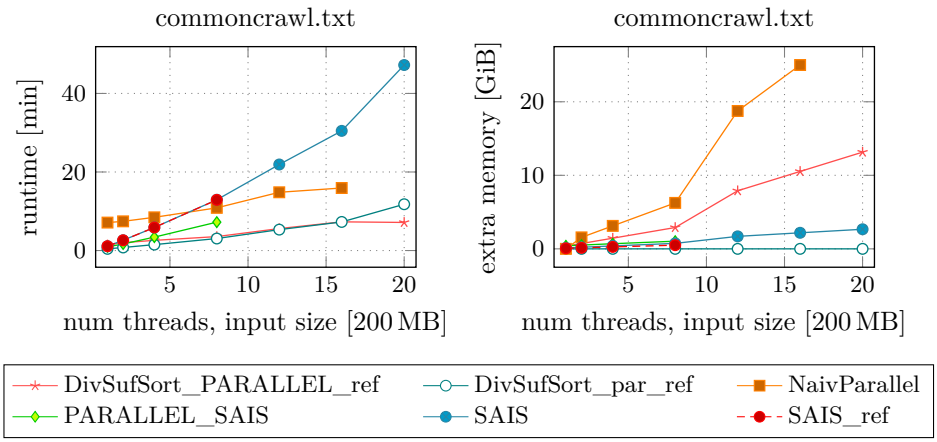


Abbildung 7.48: pSAIS verglichen mit anderen parallelen und nicht-parallelen Algorithmen auf commoncrawl.txt

Die Speichermessung ergibt, dass der benötigte Speicher des pSAIS (PARALLEL_SAIS) nicht von der Eingabegröße abhängig ist. Dies liegt daran, dass der zusätzlich benötigte Speicher nur von der Blockgröße β abhängig ist, welche auf einen konstanten Wert gesetzt ist. Er ist damit bezüglich des Speicherplatzverbrauchs deutlich besser als der naive Algorithmus.

Die Zeitmessung zeigt, dass der pSAIS immer schlechter als der DivSufSort_par_ref ist, aber immer besser als der naive Algorithmus, der dafür aber deutlich besser skaliert. Aus Zeitgründen konnte das Verhalten des pSAIS bei der Ausführung auf mehr als acht Kernen nicht weiter analysiert werden. Dort kommt es zu Fehlern und der Algorithmus schafft es nicht, das Suffix-Array zu berechnen. Auch anzumerken ist, dass der parallele Algorithmus immer besser als sein sequentielles Gegenstück SAIS ist. Das bedeutet, dass die vorgenommenen Parallelisierungen erfolgreich waren, was auch der guten Skalierung zu entnehmen ist.

7.6.4 GPU-Implementierungen

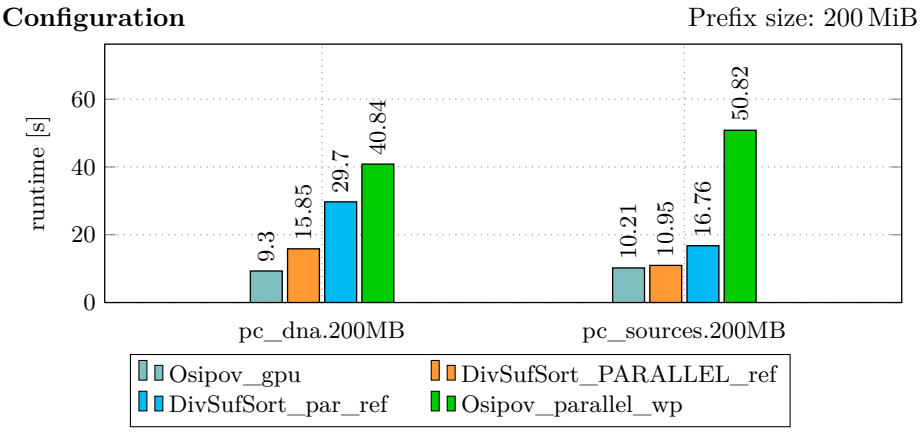


Abbildung 7.49: CPU- und GPU-Version des Osipov-Algorithmus im Vergleich mit DivSufSort auf pc_dna.xml.200MB und pc_sources.200MB

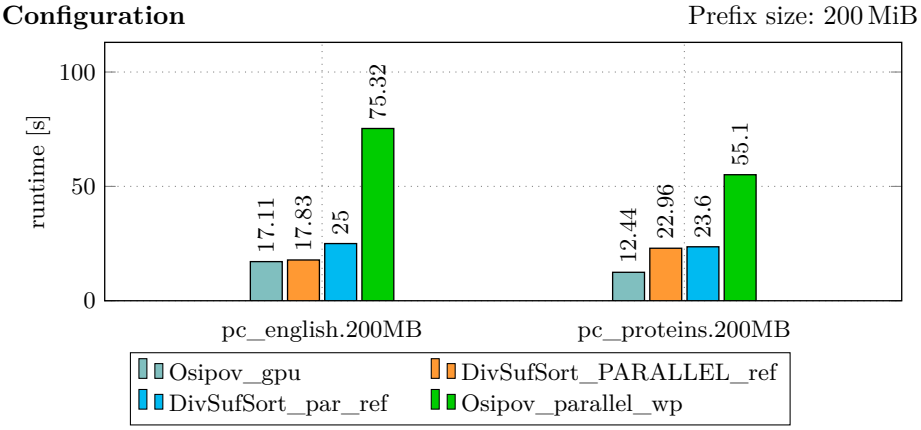


Abbildung 7.50: CPU- und GPU-Version des Osipov-Algorithmus im Vergleich mit DivSufSort auf pc_proteins.200MB und pc_english.200MB

Die Abbildungen 7.49 und 7.50 zeigen die Laufzeitvergleiche zwischen der parallelen Version des DivSufSort von Mori sowie der Implementierung von Shun et al. und den von uns implementierten Varianten des Prefix-Doublers nach Osipov, sowohl die parallele Variante für die CPU als auch für die GPU. Die Optimierung von Shun et al. stellt dabei den schnellsten von uns gemessenen Algorithmus auf der CPU dar.

Die drei CPU-Algorithmen wurden jeweils, wie zuvor, auf dem Intel® Xeon® Chipsatz mit 20 Kernen ausführt, wohingegen beim GPU-Algorithmus als Koprozessor eine NVIDIA® Tesla K40 Grafikkarte mit 2880 CUDA Kernen zum Einsatz kommt. Dieser stehen 12GB GDDR5 zur Verfügung. Als Programmierplattform nutzen wir dabei CUDA 10. Als Eingabetexte nutzen wir auch hier eine Auswahl von Texten aus dem Pizza&Chilli Korpus, alle sind dabei jeweils 200MB groß.

Die Laufzeitmessungen zeigen, dass auch hier die stark parallelisierte Variante des DivSufSort auf allen vier Eingabetexten den anderen Implementierungen auf der CPU laufzeittechnisch zum Teil deutlich überlegen ist. Anders sieht es hingegen im Vergleich zur GPU-Implementierung aus: auf allen vier Eingabetexten erreicht diese bessere Laufzeiten als alle drei CPU-Algorithmen, zum Teil nur mit minimalen Unterschieden. Durchschnittlich braucht diese in unseren Benchmarks 12,26 Sekunden, die Parallelisierung von Shun et al. 16,9 Sekunden, der DivSufSort nach Mori 23,77 Sekunden und das CPU-Pendant des Prefix-Doublers sogar 55,52 Sekunden. Zudem lässt sich beobachten, dass sowohl der GPU-Osipov mit 3,02 Sekunden, der DivSufSort nach Shun et al. mit 4,3 Sekunden, als auch der parallele DivSufSort nach Mori mit 4,63 Sekunden, geringe Standardabweichungen aufweisen und dementsprechend nicht unverhältnismäßig verschieden auf unterschiedliche Eingabetexte reagieren. Anderes gilt dabei für die CPU-Variante des Osipov-Algorithmus, der für die vier Texte gleicher Größe eine Standardabweichung von 12,55 Sekunden und damit vergleichsweise große Schwankungen in der Laufzeit aufweist.

Aus diesen Grafiken allerdings nicht ersichtlich ist der hohe Speicherverbrauch des Osipov-Algorithmus. Wie bereits in der Algorithmenbeschreibung in Kapitel 6.2 erwähnt, wird neben einem zusätzlichen Hilfsarray `aux` darüber hinaus noch eine Liste aus Tripeln benötigt, die je nach Eingabe bis zu n Elemente groß sein kann. Damit summiert sich der Speicherverbrauch der Osipov-Implementierungen zu $28n$ ($20n$ für unsere Implementierung, ca. $8n$ für den Radixsort) und ist damit deutlich höher als der des DivSufSort, welcher Zusatzspeicher abhängig von der Alphabetgröße $|\Sigma|$ benötigt.

Die Werte zeigen, dass über die GPU beschleunigte Algorithmen einen Geschwindigkeitsvorteil aufweisen können. Bei unserem GPU-SACA handelt es sich um eine unoptimierte Version des Osipov-Algorithmus, welcher optimierte Versionen der Präfixsumme und des Radixsorts aus dem externen CUB-Framework verwendet. Diese unoptimierte Variante konnte bereits den schnellsten CPU-SACA unseres Frameworks schlagen, auch wenn diese bei den Texten `pc_sources.200MB` sowie `pc_english.200MB` etwa gleichauf waren. Dies deutet

darauf hin, dass die Verwendung der GPU zur Berechnung des Suffix-Arrays lohnenswert sein kann und daher näher untersucht werden sollte. Wie bereits in Abschnitt 6.2 zum Osipov-Algorithmus erwähnt wurde, gibt es einige wenige SACAs, welche sogar noch schnellere Ergebnisse erzielen können. Leider ist es uns nicht gelungen, eine (lauffähige) Version dieser Algorithmen zu erhalten, weshalb ein noch größerer Augenmerk auf die Entwicklung von GPU-SACAs gelegt werden sollte. Ein besonders wichtiges Ziel könnte hierbei die Reduktion des Speicherbedarfs darstellen, da zum aktuellen Stand der Grafikspeicher (einer einzelnen Grafikkarte) deutlich geringer ist als der Arbeitsspeicher eines Testsystems und beispielsweise unsere Implementierung des Osipov-SACAs $28n$ Zusatzspeicher benötigt. Zusätzlich wäre die Nutzung mehrerer Grafikkarten für die Berechnung eines Suffix-Arrays ebenfalls eine Option. Dies stellt durch häufige Datenabhängigkeiten, wie durch das inverse Suffix-Array, jedoch eine besondere Herausforderung dar.

Kapitel 8

Fazit

8.1 Zusammenfassung

Wir haben das Tool SACABench vorgestellt, welches viele verschiedene SACAs enthält. Durch dieses Commandline Tool können alle Algorithmen inklusive bestehender Referenzimplementationen einfach ausgeführt und in Laufzeit und Speicherverbrauch auf Texten mit verschiedenen Eigenschaften gegeneinander getestet werden.

Es wurden gemeinsame Komponenten wie beispielsweise verschiedene Sortierverfahren ausgelagert und optimiert, um doppelten Code zu verhindern und bessere Implementierungen dieser Algorithmen zu erreichen.

Zusammenfassend sieht es so aus, dass unsere Implementierung teils einen besseren Speicherverbrauch vorweist, dafür aber meistens eine schlechtere Laufzeit als die Referenz erzielt. Insgesamt wurde das Ziel, die angegebenen Algorithmen zu implementieren und durch ein Benchmark-Framework miteinander zu vergleichen, erreicht. Es zeigt sich, dass DivSufSort immernoch auf den meisten Texten der schnellste SACA ist.

Im zweiten Semester haben wir eine Auswahl von Algorithmen auf GPU und CPU parallelisiert: zuerst wurde dabei eine naive Parallelisierung versucht, indem die Sortieralgorithmen in den SACAs durch parallele Varianten ersetzt wurden, beispielsweise den parallelen IPs^{4}o (Abschnitt 4.1.6) oder den stabilen Sortierer aus der GNU Standardbibliothek. Dabei dominieren die Sortierer aus der Standardbibliothek klar. Zuletzt wurden komplexere Parallelisierungen der SACAs verfolgt, wie beispielsweise für SAIS oder Deep-Shallow: Diese zeigen jedoch noch Optimierungspotential.

8.2 Ausblick

Da einige unserer Implementierungen sowie unsere parallelen Implementierungen noch wenig optimiert sind, wäre es in Zukunft interessant, diese weiter zu verbessern. Wünschenswert wäre es, alle Algorithmen laufzeit- bzw. speichereffizienter als die Referenzimplementierung zu implementieren.

Es wäre auch denkbar, das Framework durch eine graphische Benutzeroberfläche attraktiver zu gestalten sowieo die Benutzung zu vereinfachen. Dadurch wäre es noch einfacher, für eine bestimmte Menge SACAs Laufzeit- und Speichermessungen durchzuführen und direkt Plots in der GUI zu generieren. Dies ist allerdings im Rahmen der Projektgruppe aus Zeitgründen nicht mehr möglich.

Anhang A

Manual

A.1 sacabench

SACABENCH(1) User Commands SACABENCH(1)

NAME

sacabench − manual page for sacabench 1.0

SYNOPSIS

sacabench [*OPTIONS*] *SUBCOMMAND*

DESCRIPTION

CLI for SACABench.

OPTIONS

−h,−−help
Print this help message and exit

Subcommands:

list List all implemented algorithms.

construct
Construct a SA.

demo Run all algorithms on an example string.

batch Measure runtime and memory usage for all algorithms.

plot Plot measurements.

SEE ALSO

sacabench list(1), **sacabench construct**(1), **sacabench demo**(1), **sacabench batch**(1), **sacabench plot**(1)

sacabench 1.0 February 2019 1

A.2 sacabench list

SACABENCH LIST(1) User Commands SACABENCH LIST(1)

NAME

sacabench list − manual page for sacabench list 1.0

SYNOPSIS

sacabench *list* [*OPTIONS*]

DESCRIPTION

List all implemented algorithms.

OPTIONS

−h,−−help
Print this help message and exit

−n,−−no−description
Don't show a description for each algorithm.

−j,−−json
Output list as an json array

SEE ALSO

sacabench(1)

sacabench list 1.0 February 2019 1

A.3 sacabench demo

SACABENCH DEMO(1) User Commands SACABENCH DEMO(1)

NAME
sacabench demo − manual page for sacabench demo 1.0

SYNOPSIS
sacabench *demo* [*OPTIONS*]

DESCRIPTION
Run all algorithms on an example string.

OPTIONS
−h,−−help
Print this help message and exit

SEE ALSO
sacabench(1)

sacabench demo 1.0 February 2019 1

A.4 sacabench construct

SACABENCH CONSTRUCT(1) User Commands SACABENCH CONSTRUCT(1)

NAME

sacabench construct − manual page for sacabench construct 1.0

SYNOPSIS

sacabench *construct* [*OPTIONS*] *algorithm input*

DESCRIPTION

Construct a SA.

Positionals:

algorithm TEXT REQUIRED
: Which algorithm to run.

input TEXT REQUIRED
: Path to input file, or − for STDIN.

OPTIONS

−h,−−help
: Print this help message and exit

−−**config** TEXT
: Read an config file for CLI args

−c,−−check
: Check the constructed SA.

−q,−−fastcheck
: Check the constructed SA with a faster, parallel algorithm.

−b,−−benchmark TEXT
: Record benchmark and output as JSON. Takes path to output file, or − for STDOUT

−J,−−json TEXT
: Output SA as JSON array. Takes path to output file, or − for STDOUT.

−B,−−binary TEXT
: Output SA as binary array of unsigned integers, with a 1 Byte header describing the number of bits used for each integer. Takes path to output file, or − for STDOUT.

−F,−−fixed UINT Needs: −−**binary**
: Elide the header, and output a fixed number of bits per SA entry

−p,−−prefix TEXT
: Calculate SA of prefix of size TEXT.

−f,−−force
: Overwrite existing files instead of raising an error.

−m,−−minimum_sa_bits UINT=32
: The lower bound of bits to use per SA entry during construction

−r,−−repetitions UINT=1
: The value indicates the number of times the SACA(s) will run. A larger number will possibly yield more accurate results

−z,−−rplot Needs: −−**benchmark**
: Plots measurements with R.

−−**latexplot** Needs: −−**benchmark**
: Plots measurements with LaTex and SqlPlotTools.

−s,−−sysinfo Needs: −−**benchmark**
: Add system information to benchmark output.

sacabench construct 1.0 February 2019 1

SEE ALSO

sacabench(1)

A.5 sacabench batch

SACABENCH BATCH(1) User Commands SACABENCH BATCH(1)

NAME

sacabench batch − manual page for sacabench batch 1.0

SYNOPSIS

sacabench *batch* [*OPTIONS*] *input*

DESCRIPTION

Measure runtime and memory usage for all algorithms.

Positionals:

input TEXT REQUIRED
Path to input file, or − for STDIN.

OPTIONS

−h,−−help
Print this help message and exit

−−**config** TEXT
Read an config file for CLI args

−c,−−check
Check the constructed SA.

−q,−−fastcheck
Check the constructed SA with a faster, parallel algorithm.

−b,−−benchmark TEXT
Record benchmark and output as JSON. Takes path to output file, or − for STDOUT

−f,−−force
Overwrite existing files instead of raising an error.

−m,−−minimum_sa_bits UINT=32
The lower bound of bits to use per SA entry during construction

−p,−−prefix TEXT
calculate SA of prefix of input.

−r,−−repetitions UINT=1
The value indicates the number of times the SACA(s) will run. A larger number will possibly yield more accurate results

−−**whitelist** TEXT ... Excludes: −−**blacklist**
Execute only specific algorithms

−−**blacklist** TEXT ... Excludes: −−**whitelist**
Blacklist algorithms from execution

−z,−−rplot Needs: −−**benchmark**
Plots measurements with R.

−−**latexplot** Needs: −−**benchmark**
Plots measurements with LaTex and SqlPlotTools.

−s,−−sysinfo Needs: −−**benchmark**
Add system information to benchmark output.

SEE ALSO

sacabench(1)

sacabench batch 1.0 February 2019 1

A.6 sacabench plot

SACABENCH PLOT(1) User Commands SACABENCH PLOT(1)

NAME

sacabench plot − manual page for sacabench plot 1.0

SYNOPSIS

sacabench *plot* [*OPTIONS*] *benchmark_file*

DESCRIPTION

Plot measurements.

Positionals:

benchmark_file TEXT REQUIRED

Path to benchmark json file.

OPTIONS

−h,−−help

Print this help message and exit

SEE ALSO

sacabench(1)

sacabench plot 1.0 February 2019 1

Anhang B

Messwerte

B.1 Sequentielle Algorithmen

B.1.1 Kleine Dateien

	cc_commoncrawl.ascii	pc_dblp.xml	pc_dna	pc_english	pc_proteins	pc_sources	pcr_cere	pcr_einstein.en.txt	pcr_kernel	pcr_para	tagme_wiki-disamb30	wiki_all_vital.txt
BPR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
BPR_{ref}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DC3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DC3-Lite	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$DC3_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DC7	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Deep-Shallow	✓	✓	✓	✓	✓	✓	◷	✓	◷	◷	✓	✓
Deep-Shallow_bb	✓	✓	✓	✓	✓	✓	◷	✓	◷	◷	✓	✓
$Deep\text{-}Shallow_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Discarding2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Discarding4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DivSufSort	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$DivSufSort_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Doubling	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
GSACA	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
GSACA_Opt	◷	◷	◷	◷	◷	◷	◷	◷	◷	◷	◷	◷
$GSACA_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$MSufSort_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Naiv	✓	✓	✓	✓	✓	✓	✓	✓	◷	✓	✓	✓
NaivIps4o	◷	◷	◷	◷	◷	◷	◷	◷	◷	◷	◷	◷
Osipov_sequential	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Osipov_sequential_wp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SACA-K	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$SACA\text{-}K_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SADS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$SADS_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SAIS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$SAIS\text{-}LITE_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SAIS_WPI	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$SAIS_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MSufSort	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MSufSortV2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MSufSort_scan	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NzSufSort	✓	✓	✓	✓	✓	✓	◷	✓	✓	◷	✓	✓
Qsufsort	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$Qsufsort_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tabelle B.1: SA Korrektheit Small Sequential

✓ SA wurde korrekt berechnet.
✗ SA wurde falsch berechnet.
◷ Berechnung hat Zeitlimit des Systems erreicht.
Berechnung hat Speicherlimit des Systems erreicht.
Berechnung brach mit einem Laufzeitfehler ab.
⊘ Nicht durch Implementierung unterstützt.

	cc_commoncrawl.ascii	pc_dblp.xml	pc_dna	pc_english	pc_proteins	pc_sources	pcr_cere	pcr_einstein.en.txt	pcr_kernel	pcr_para	tagme_wiki-disamb30	wiki_all_vital.txt
BPR	0.49	0.53	0.57	0.67	0.52	0.45	0.59	0.72	0.59	0.60	0.55	0.55
BPR_{ref}	0.51	0.53	0.68	0.63	0.53	0.50	0.70	0.72	0.58	0.72	0.57	0.60
DC3	2.76	2.68	2.83	2.84	2.78	2.76	2.88	3.01	2.89	2.90	2.71	2.79
DC3-Lite	24.27	27.68	8.68	18.97	25.62	30.83	7.60	10.13	9.88	7.88	23.19	24.35
$DC3_{ref}$	4.79	4.26	4.66	5.51	5.47	4.75	3.90	3.92	4.24	4.10	5.23	5.45
DC7	3.58	3.64	3.78	3.72	3.68	3.46	3.95	4.16	3.98	3.97	3.79	3.68
Deep-Shallow	5.33	1.71	1.84	13.22	3.45	2.18	–	42.00	–	–	1.67	1.45
Deep-Shallow_bb	2.98	1.20	1.86	7.83	2.10	1.77	–	20.82	–	–	1.72	1.34
$Deep\text{-}Shallow_{ref}$	0.74	0.52	0.64	0.88	0.79	0.53	1.05	2.25	1.51	1.09	0.62	0.63
Discarding2	4.80	3.04	2.07	5.21	3.56	3.22	8.28	11.06	13.53	8.30	2.90	2.26
Discarding4	3.74	2.70	2.14	3.98	2.93	2.73	6.29	8.04	9.42	6.27	2.53	2.15
DivSufSort	5.45	4.34	4.20	5.11	4.43	3.35	8.51	24.72	14.94	8.98	3.45	3.52
$DivSufSort_{ref}$	0.29	0.30	0.48	0.46	0.48	0.33	0.40	0.45	0.30	0.49	0.35	0.50
Doubling	13.70	7.11	12.54	15.54	12.07	14.46	13.24	14.10	17.18	12.60	7.63	10.21
GSACA	1.40	1.47	1.45	1.76	1.64	1.38	1.34	1.56	1.41	1.31	1.73	1.67
GSACA_Opt	–	–	–	–	–	–	–	–	–	–	–	–
$GSACA_{ref}$	1.40	1.39	1.38	1.57	1.55	1.28	1.20	1.50	1.32	1.23	1.53	1.69
$MSufSort_{ref}$	1.11	0.74	1.14	0.89	0.79	0.66	1.52	1.01	1.05	1.20	0.90	0.98
Naiv	4.72	3.19	3.02	6.41	3.09	3.05	7.12	69.65	–	5.29	2.78	2.78
NaivIps4o	–	–	–	–	–	–	–	–	–	–	–	–
Osipov_sequential	4.16	3.21	2.54	4.95	3.65	3.25	6.46	8.68	9.27	6.47	3.29	2.80
Osipov_sequential_wp	3.82	2.91	2.32	4.75	3.29	2.89	6.19	8.21	9.21	6.22	2.91	2.42
SACA-K	1.57	1.11	1.41	1.57	2.22	1.04	4.38	1.40	1.06	1.82	1.77	1.53
$SACA\text{-}K_{ref}$	1.08	0.88	1.15	1.25	1.27	0.82	1.02	1.40	0.87	1.25	1.19	1.22
SADS	2.05	2.04	2.73	2.85	2.98	2.59	2.27	2.48	2.77	3.23	3.86	2.84
$SADS_{ref}$	1.43	1.39	1.99	2.05	2.94	1.32	1.65	1.79	1.87	2.39	1.98	2.79
SAIS	1.11	1.09	1.48	2.16	1.67	1.01	1.28	2.02	1.46	1.32	1.52	1.56
$SAIS\text{-}LITE_{ref}$	0.61	0.39	0.97	0.95	0.84	0.58	0.39	0.35	0.45	0.55	0.62	0.69
SAIS_WPI	0.80	0.79	0.97	1.08	1.15	0.72	1.15	0.93	0.71	0.91	1.32	1.11
$SAIS_{ref}$	1.11	1.07	1.46	2.15	1.81	1.38	1.27	2.03	1.05	1.32	1.51	1.55
MSufSort	5.47	2.12	2.93	2.69	2.14	2.08	13.63	3.43	4.25	5.03	2.72	1.99
MSufSortV2	5.26	2.88	2.24	3.38	2.82	2.09	13.69	3.46	3.53	5.04	2.05	2.67
MSufSort_scan	8.97	1.61	1.55	3.89	2.02	2.08	3.58	2.81	4.40	2.20	1.78	1.50
NzSufSort	14.28	12.84	11.12	12.38	17.06	17.98	–	14.44	7.31	–	13.94	12.70
Qsufsort	2.87	2.50	2.72	3.79	2.46	2.70	3.76	4.80	5.09	3.94	2.42	2.52
$Qsufsort_{ref}$	1.33	1.14	1.25	1.83	1.13	1.10	1.74	2.22	2.42	1.72	1.22	1.18

Tabelle B.2: Laufzeit in Minuten Small Sequential

Grün Die besten drei Werte.

Rot Die schlechtesten drei Werte.

	cc_commoncrawl.ascii	pc_dblp.xml	pc_dna	pc_english	pc_proteins	pc_sources	pcr_cere	pcr_einstein.en.txt	pcr_kernel	pcr_para	tagme_wiki-disamb30	wiki_all_vital.txt
BPR	0.793	0.788	0.792	0.867	0.785	0.873	0.784	0.796	0.812	0.784	0.846	0.845
BPR_{ref}	3.331	3.327	3.328	3.405	3.323	3.411	3.321	3.335	3.351	3.321	3.385	3.384
DC3	3.119	3.079	3.116	3.121	3.111	3.119	3.116	3.121	3.122	3.116	3.079	3.105
DC3-Lite	1.563	1.563	1.563	1.563	1.563	1.563	1.563	1.563	1.563	1.563	1.563	1.563
$DC3_{ref}$	4.660	4.482	4.647	4.669	4.627	4.660	4.647	4.669	4.675	4.647	4.482	4.596
DC7	1.961	1.961	1.961	1.961	1.961	1.961	1.961	1.961	1.961	1.961	1.961	1.961
Deep-Shallow	0.002	0.002	0.002	0.003	0.002	0.003	–	0.005	–	–	0.003	0.003
Deep-Shallow_bb	0.002	0.002	0.002	0.003	0.002	0.003	–	0.005	–	–	0.003	0.003
$Deep\text{-}Shallow_{ref}$	0.002	0.002	0.002	0.002	0.002	0.003	0.002	0.005	0.002	0.002	0.002	0.002
Discarding2	2.346	2.346	2.346	2.346	2.346	2.346	2.346	2.346	2.346	2.346	2.346	2.346
Discarding4	3.908	3.908	3.908	3.908	3.908	3.908	3.908	3.908	3.908	3.908	3.908	3.908
DivSufSort	0.477	0.449	0.436	0.490	0.499	0.448	0.407	0.470	0.451	0.422	0.479	0.494
$DivSufSort_{ref}$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Doubling	2.346	2.346	2.346	2.346	2.346	2.346	2.346	2.346	2.346	2.346	2.346	2.346
GSACA	3.125	3.125	3.125	3.125	3.125	3.125	3.125	3.125	3.125	3.125	3.125	3.125
GSACA_Opt	–	–	–	–	–	–	–	–	–	–	–	–
$GSACA_{ref}$	2.344	2.344	2.344	2.344	2.344	2.344	2.344	2.344	2.344	2.344	2.344	2.344
$MSufSort_{ref}$	0.792	0.806	0.982	0.811	0.807	0.792	1.183	0.811	0.795	1.183	0.807	0.807
Naiv	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	–	0.000	0.000	0.000
NaivIps4o	–	–	–	–	–	–	–	–	–	–	–	–
Osipov_sequential	5.466	5.469	5.469	5.469	5.469	5.469	5.469	5.469	5.469	5.469	5.469	5.469
Osipov_sequential_wp	5.466	5.469	5.469	5.469	5.469	5.469	5.469	5.469	5.469	5.469	5.469	5.469
SACA-K	0.227	0.126	0.333	0.291	0.390	0.241	0.034	0.001	0.016	0.046	0.303	0.317
$SACA\text{-}K_{ref}$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SADS	0.489	0.457	0.576	0.533	0.549	0.516	0.334	0.305	0.318	0.344	0.533	0.565
$SADS_{ref}$	0.079	0.064	0.116	0.100	0.100	0.084	0.043	0.038	0.039	0.046	0.095	0.104
SAIS	0.092	0.083	0.108	0.110	0.125	0.100	0.043	0.036	0.039	0.046	0.109	0.119
$SAIS\text{-}LITE_{ref}$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SAIS_WPI	0.087	0.079	0.130	0.104	0.122	0.093	0.162	0.036	0.033	0.162	0.104	0.116
$SAIS_{ref}$	0.063	0.051	0.075	0.071	0.083	0.064	0.037	0.036	0.036	0.039	0.072	0.074
MSufSort	0.219	0.119	0.483	0.381	0.188	0.239	0.474	0.227	0.241	0.481	0.239	0.248
MSufSortV2	0.219	0.119	0.483	0.381	0.188	0.239	0.474	0.227	0.241	0.481	0.239	0.248
MSufSort_scan	0.024	0.047	0.094	0.047	0.012	0.024	0.094	0.047	0.024	0.094	0.047	0.047
NzSufSort	0.196	0.000	0.000	0.001	0.195	0.196	–	0.196	0.196	–	0.196	0.000
Qsufsort	0.783	0.783	0.783	0.783	0.783	0.783	0.783	0.783	0.783	0.783	0.783	0.783
$Qsufsort_{ref}$	0.783	0.783	0.783	0.783	0.783	0.783	0.783	0.783	0.783	0.783	0.783	0.783

Tabelle B.3: Extra-Speicher in GiB Small Sequential

Grün Die besten drei Werte.

Rot Die schlechtesten drei Werte.

B.1.2 Große Dateien mit Input Scaling

				commoncrawl.txt							dna.txt							wiki.txt			
Threads:	1	2	4	8	12	16	20	1	2	4	8	12	16	20	1	2	4	8	12	16	20
BPR	✓	✓	✓	✓	✓	✓	💾	✓	✓	✓	✓	✓	✓	💾	✓	✓	✓	✓	✓	✓	💾
BPR_{ref}	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
DC3	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
DC3-Lite	✓	✓	🕘	🕘	🕘	💾	💾	✓	✓	🕘	🕘	🕘	💾	💾	✓	✓	🕘	🕘	🕘	💾	💾
$DC3_{ref}$	✓	✓	✓	✓	⊘	⊘	⊘	✓	✓	✓	✓	⊘	⊘	⊘	✓	✓	✓	✓	⊘	⊘	⊘
DC7	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
Deep-Shallow	✓	✓	🕘	🕘	🕘	🕘	🕘	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Deep-Shallow_bb	✓	🕘	🕘	🕘	🕘	🕘	🕘	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$Deep\text{-}Shallow_{ref}$	✓	✓	✓	🕘	⊘	⊘	⊘	✓	✓	✓	✗	⊘	⊘	⊘	✓	✓	✓	🕘	⊘	⊘	⊘
Discarding2	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
Discarding4	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
DivSufSort	✓	✓	✓	✓	✓	🕘	🕘	✓	✓	✓	✓	✓	✓	🕘	✓	✓	✓	✓	✓	✓	🕘
$DivSufSort_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Doubling	✓	✓	✓	🕘	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
GSACA	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
GSACA_Opt	🕘	🕘	🕘	🕘	💾	💾	💾	🕘	🕘	🕘	🕘	💾	💾	💾	🕘	🕘	🕘	🕘	💾	💾	💾
$GSACA_{ref}$	✓	✓	✓	✓	⊘	⊘	⊘	✓	✓	✓	✓	⊘	⊘	⊘	✓	✓	✓	✓	⊘	⊘	⊘
$MSufSort_{ref}$	✓	✓	✓	ϟ	⊘	⊘	⊘	✓	✓	✓	ϟ	⊘	⊘	⊘	✓	✓	✓	ϟ	⊘	⊘	⊘
Naiv	✓	✓	✓	✓	🕘	🕘	🕘	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NaivIps4o	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘	🕘
Osipov_sequential	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
Osipov_sequential_wp	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
SACA-K	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$SACA\text{-}K_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SADS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$SADS_{ref}$	✓	✓	✓	✓	⊘	⊘	⊘	✓	✓	✓	✓	⊘	⊘	⊘	✓	✓	✓	✓	⊘	⊘	⊘
SAIS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$SAIS\text{-}LITE_{ref}$	✓	✓	✓	✓	⊘	⊘	⊘	✓	✓	✓	✓	⊘	⊘	⊘	✓	✓	✓	✓	⊘	⊘	⊘
SAIS_WPI	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$SAIS_{ref}$	✓	✓	✓	✓	⊘	⊘	⊘	✓	✓	✓	✓	⊘	⊘	⊘	✓	✓	✓	✓	⊘	⊘	⊘
MSufSort	✓	✓	✓	✓	✓	✓	🕘	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MSufSortV2	✓	✓	✓	✓	✓	✓	🕘	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MSufSort_scan	✓	✓	✓	✓	✓	✓	🕘	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NzSufSort	✓	✓	🕘	🕘	🕘	🕘	🕘	✓	✓	✓	🕘	🕘	🕘	🕘	✓	✓	🕘	🕘	🕘	🕘	🕘
Qsufsort	✓	✓	✓	✓	✓	✓	💾	✓	✓	✓	✓	✓	✓	💾	✓	✓	✓	✓	✓	✓	💾
$Qsufsort_{ref}$	✓	✓	✓	✓	⊘	⊘	⊘	✓	✓	✓	✓	⊘	⊘	⊘	✓	✓	✓	✓	⊘	⊘	⊘

Tabelle B.4: SA Korrektheit Large Sequential Input-Scaling

- ✓ SA wurde korrekt berechnet.
- ✗ SA wurde falsch berechnet.
- 🕘 Berechnung hat Zeitlimit des Systems erreicht.
- 💾 Berechnung hat Speicherlimit des Systems erreicht.
- ϟ Berechnung brach mit einem Laufzeitfehler ab.
- ⊘ Nicht durch Implementierung unterstützt.

	commoncrawl.txt							dna.txt							wiki.txt						
Threads:	1	2	4	8	12	16	20	1	2	4	8	12	16	20	1	2	4	8	12	16	20
BPR	0.46	1.00	2.15	4.66	9.43	13.66	–	0.56	1.21	2.54	5.30	9.87	15.05	–	0.53	1.14	2.44	5.26	10.44	15.30	–
BPR_{ref}	0.49	1.06	2.29	5.07	–	–	–	0.62	1.38	3.01	6.44	–	–	–	0.57	1.25	2.74	5.99	–	–	–
DC3	2.66	5.61	11.79	25.66	–	–	–	2.69	5.56	11.51	24.55	–	–	–	2.75	5.71	11.95	26.27	–	–	–
DC3-Lite	19.64	70.62	–	–	–	–	–	9.19	41.89	–	–	–	–	–	24.37	86.88	–	–	–	–	–
$DC3_{ref}$	4.78	10.52	22.14	57.39	–	–	–	4.07	8.07	16.65	43.40	–	–	–	5.34	11.16	23.09	58.87	–	–	–
DC7	3.45	7.19	15.06	31.54	–	–	–	3.62	7.52	15.49	31.80	–	–	–	3.59	7.43	15.40	32.25	–	–	–
Deep-Shallow	6.23	11.96	–	–	–	–	–	1.86	4.20	9.32	19.13	30.25	41.42	52.98	1.42	3.17	7.09	17.56	30.89	41.20	52.15
Deep-Shallow_bb	3.64	–	–	–	–	–	–	1.52	3.25	7.07	15.44	24.59	33.85	43.51	1.32	2.88	6.22	14.51	29.63	33.67	42.95
$Deep\text{-}Shallow_{ref}$	0.63	1.41	3.18	–	–	–	–	0.63	1.33	2.87	–	–	–	–	0.56	1.25	2.73	–	–	–	–
Discarding2	4.38	9.27	20.33	43.98	–	–	–	2.31	4.97	10.59	21.34	–	–	–	2.42	5.14	10.88	23.92	–	–	–
Discarding4	3.40	7.22	15.79	34.35	–	–	–	2.32	5.08	10.91	22.86	–	–	–	2.22	4.75	10.14	22.53	–	–	–
DivSufSort	5.02	10.69	24.62	56.32	98.90	–	–	4.12	9.86	21.80	42.45	70.04	102.05	–	3.20	7.20	16.13	38.01	71.15	98.28	–
$DivSufSort_{ref}$	0.43	0.78	1.51	3.08	5.42	9.22	9.32	0.51	1.00	2.03	4.17	8.94	10.19	14.97	0.47	0.85	1.67	3.35	5.74	7.91	10.22
Doubling	13.68	30.33	66.47	–	–	–	–	5.55	11.30	23.28	47.88	–	–	–	11.08	22.97	47.47	97.19	–	–	–
GSACA	1.34	2.97	6.45	15.21	–	–	–	1.44	3.05	6.56	15.88	–	–	–	1.62	3.53	7.69	17.62	–	–	–
GSACA_Opt	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
$GSACA_{ref}$	1.24	2.76	6.03	13.24	–	–	–	1.36	2.88	6.20	13.67	–	–	–	1.52	3.30	7.29	15.65	–	–	–
$MSufSort_{ref}$	0.73	1.57	3.29	–	–	–	–	0.98	2.03	4.32	–	–	–	–	0.77	1.66	3.59	–	–	–	–
Naiv	7.13	11.16	26.60	62.00	–	–	–	3.08	6.95	15.21	32.03	55.28	67.36	87.93	2.74	6.00	13.03	28.82	58.17	63.63	81.76
NaivIps4o	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Osipov_sequential	3.94	8.40	18.03	40.69	–	–	–	2.78	5.92	12.51	27.05	–	–	–	2.86	6.06	12.79	29.74	–	–	–
Osipov_sequential_wp	3.58	7.67	16.52	37.87	–	–	–	2.50	5.36	11.37	24.71	–	–	–	2.49	5.32	11.31	26.79	–	–	–
SACA-K	1.15	2.57	5.53	11.78	19.89	27.44	41.21	1.36	2.81	5.86	12.06	19.80	27.81	39.50	1.41	3.04	6.45	13.14	21.92	30.24	42.85
$SACA\text{-}K_{ref}$	0.91	2.04	4.38	9.32	18.30	28.36	25.40	1.11	2.33	4.86	9.97	19.41	29.27	38.55	1.11	2.42	5.12	10.46	21.12	24.40	29.52
SADS	2.05	4.70	10.15	21.68	36.03	59.08	78.14	2.64	5.37	11.13	23.22	50.83	63.57	85.60	2.60	5.65	11.92	24.39	47.92	76.19	84.67
$SADS_{ref}$	1.43	3.38	7.50	16.36	–	–	–	1.93	4.10	8.92	19.15	–	–	–	1.83	4.16	9.16	19.14	–	–	–
SAIS	1.11	2.62	5.87	12.93	21.91	30.47	47.24	1.42	3.09	6.80	14.52	24.34	35.23	55.80	1.43	3.26	7.25	15.35	25.15	34.95	55.72
$SAIS\text{-}LITE_{ref}$	0.50	1.08	2.23	4.82	–	–	–	0.66	1.31	2.67	5.96	–	–	–	0.62	1.32	2.77	5.71	–	–	–
SAIS_WPI	1.46	2.60	4.72	9.13	47.84	61.62	32.52	1.95	3.02	5.18	9.43	17.98	24.19	33.42	1.86	3.12	5.58	10.28	46.34	54.16	36.93
$SAIS_{ref}$	1.10	2.61	5.87	12.90	–	–	–	1.41	3.07	6.76	14.45	–	–	–	1.42	3.25	7.25	15.29	–	–	–
MSufSort	2.91	6.36	12.69	27.02	51.46	69.81	–	2.24	4.91	10.78	23.17	46.29	63.85	86.61	1.94	4.29	9.56	22.06	43.58	79.74	83.98
MSufSortV2	2.87	6.20	12.49	26.79	51.40	85.65	–	2.26	4.93	10.74	23.22	46.36	85.53	87.77	1.95	4.32	9.61	22.19	43.48	62.44	83.43
MSufSort_scan	5.50	11.44	19.95	38.86	68.21	87.38	–	1.44	3.11	6.74	14.55	27.46	43.86	52.45	1.54	3.30	7.11	16.04	29.31	40.64	63.22
NzSufSort	14.49	46.75	–	–	–	–	–	8.59	31.18	101.70	–	–	–	–	15.58	50.71	–	–	–	–	–
Qsufsort	2.91	6.45	14.40	32.48	61.95	72.54	–	1.79	4.10	9.01	18.71	36.61	55.64	–	2.40	5.20	11.56	27.43	54.42	84.58	–
$Qsufsort_{ref}$	1.25	2.75	6.03	13.31	–	–	–	0.97	2.17	4.63	9.68	–	–	–	1.11	2.40	5.31	11.90	–	–	–

Tabelle B.5: Laufzeit in Minuten Large Sequential Input-Scaling

Grün Die besten drei Werte.

Rot Die schlechtesten drei Werte.

	commoncrawl.txt							dna.txt							wiki.txt						
Threads:	1	2	4	8	12	16	20	1	2	4	8	12	16	20	1	2	4	8	12	16	20
BPR	0.879	1.667	3.232	6.357	18.964	25.214	–	0.784	1.565	3.127	6.252	18.753	25.003	–	0.849	1.631	3.194	6.319	18.888	25.138	–
BPR_{ref}	3.417	6.744	13.387	26.668	–	–	–	3.320	6.641	13.281	26.563	–	–	–	3.387	6.708	13.349	26.631	–	–	–
DC3	3.119	6.238	12.476	24.968	–	–	–	3.056	6.113	12.226	24.451	–	–	–	3.105	6.209	12.419	24.837	–	–	–
DC3-Lite	1.563	3.125	–	–	–	–	–	1.563	3.125	–	–	–	–	–	1.563	3.125	–	–	–	–	–
$DC3_{ref}$	4.660	9.321	18.678	37.355	–	–	–	4.379	8.758	17.515	35.031	–	–	–	4.596	9.192	18.384	36.768	–	–	–
DC7	1.961	3.922	7.844	15.689	–	–	–	1.961	3.922	7.844	15.689	–	–	–	1.961	3.922	7.844	15.689	–	–	–
Deep-Shallow	0.003	0.006	–	–	–	–	–	0.002	0.004	0.009	0.017	0.043	0.057	0.071	0.003	0.005	0.010	0.018	0.044	0.058	0.073
Deep-Shallow_bb	0.003	–	–	–	–	–	–	0.002	0.004	0.009	0.017	0.043	0.057	0.071	0.003	0.005	0.010	0.018	0.044	0.058	0.072
$Deep\text{-}Shallow_{ref}$	0.002	0.005	0.009	–	–	–	–	0.002	0.005	0.009	–	–	–	–	0.002	0.005	0.009	–	–	–	–
Discarding2	2.346	4.689	9.377	18.752	–	–	–	2.346	4.689	9.377	18.752	–	–	–	2.346	4.689	9.377	18.752	–	–	–
Discarding4	3.908	7.815	15.627	31.252	–	–	–	3.908	7.815	15.627	31.252	–	–	–	3.908	7.815	15.627	31.252	–	–	–
DivSufSort	0.536	1.054	2.107	4.219	12.634	–	–	0.431	0.901	1.807	3.397	10.102	13.365	–	0.481	0.963	1.924	3.815	11.393	15.197	–
$DivSufSort_{ref}$	0.000	0.000	0.000	0.000	0.001	0.001	0.001	0.000	0.000	0.000	0.000	0.001	0.001	0.001	0.000	0.000	0.000	0.000	0.001	0.001	0.001
Doubling	2.346	4.689	9.377	–	–	–	–	2.346	4.689	9.377	18.752	–	–	–	2.346	4.689	9.377	18.752	–	–	–
GSACA	3.125	6.250	12.500	25.000	–	–	–	3.125	6.250	12.500	25.000	–	–	–	3.125	6.250	12.500	25.000	–	–	–
GSACA_Opt	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
$GSACA_{ref}$	2.344	4.688	9.375	18.750	–	–	–	2.344	4.688	9.375	18.750	–	–	–	2.344	4.688	9.375	18.750	–	–	–
$MSufSort_{ref}$	0.792	1.573	3.155	–	–	–	–	0.982	1.963	3.926	–	–	–	–	0.791	1.588	3.226	–	–	–	–
Naiv	0.000	0.000	0.000	0.000	–	–	–	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
NaivIps4o	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Osipov_sequential	5.468	10.935	21.872	43.746	–	–	–	5.469	10.938	21.875	43.750	–	–	–	5.467	10.934	21.871	43.746	–	–	–
Osipov_sequential_wp	5.468	10.935	21.872	43.746	–	–	–	5.469	10.938	21.875	43.750	–	–	–	5.467	10.934	21.871	43.746	–	–	–
SACA-K	0.297	0.559	0.984	1.718	2.367	2.966	3.556	0.301	0.459	0.806	1.730	2.632	3.585	4.553	0.320	0.582	1.049	1.793	2.431	3.184	3.893
$SACA\text{-}K_{ref}$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
SADS	0.552	1.070	2.044	3.915	5.726	7.508	9.284	0.547	1.018	1.979	3.970	5.995	7.994	10.092	0.565	1.099	2.140	4.092	5.976	7.961	9.921
$SADS_{ref}$	0.093	0.180	0.338	0.634	–	–	–	0.107	0.180	0.335	0.658	–	–	–	0.099	0.190	0.360	0.662	–	–	–
SAIS	0.113	0.216	0.401	0.745	1.699	2.183	2.664	0.101	0.185	0.356	0.717	1.747	2.349	2.973	0.119	0.227	0.434	0.804	1.878	2.497	3.099
$SAIS\text{-}LITE_{ref}$	0.000	0.000	0.000	0.000	–	–	–	0.000	0.000	0.000	0.000	–	–	–	0.000	0.000	0.000	0.000	–	–	–
SAIS_WPI	0.113	0.220	0.404	0.745	1.704	2.200	2.668	0.101	0.162	0.216	0.580	4.273	4.309	4.576	0.119	0.228	0.438	0.808	1.894	2.511	3.076
$SAIS_{ref}$	0.073	0.141	0.265	0.499	–	–	–	0.071	0.125	0.235	0.481	–	–	–	0.075	0.142	0.269	0.499	–	–	–
MSufSort	0.202	0.408	0.816	1.631	3.890	6.519	–	0.470	0.875	1.750	3.500	12.652	14.215	15.719	0.218	0.435	0.870	1.754	6.334	7.104	7.864
MSufSortV2	0.202	0.408	0.816	1.631	3.890	6.519	–	0.470	0.883	1.766	3.516	12.652	14.215	15.719	0.218	0.435	0.870	1.754	6.334	7.104	7.864
MSufSort_scan	0.024	0.047	0.094	0.188	0.376	0.751	–	0.094	0.375	0.750	1.500	3.000	3.000	6.000	0.024	0.047	0.094	0.188	0.751	0.751	0.751
NzSufSort	0.196	0.392	–	–	–	–	–	0.000	0.000	0.000	–	–	–	–	0.000	0.001	–	–	–	–	–
Qsufsort	0.783	1.564	3.127	6.252	18.752	25.002	–	0.783	1.564	3.127	6.252	18.752	25.002	–	0.783	1.564	3.127	6.252	18.752	25.002	–
$Qsufsort_{ref}$	0.783	1.564	3.127	6.252	–	–	–	0.783	1.564	3.127	6.252	–	–	–	0.783	1.564	3.127	6.252	–	–	–

Tabelle B.6: Extra-Speicher in GiB Large Sequential Input-Scaling

Grün Die besten drei Werte.

Rot Die schlechtesten drei Werte.

B.2 Parallele Algorithmen

B.2.1 Weak Scaling

				commoncrawl.txt				dna.txt							wiki.txt						
Threads:	1	2	4	8	12	16	20	1	2	4	8	12	16	20	1	2	4	8	12	16	20
BPR_par	✓	✓	✓	✓	✓	✓	💾	✓	✓	✓	✓	✓	✓	💾	✓	✓	✓	✓	✓	✓	💾
DC3-Parallel-V1	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
DC3-Parallel-V2	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
Deep-Shallow_par	✓	✓	✓	✓	ϟ	ϟ	ϟ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Discarding2Parallel	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
Discarding4Parallel	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾	✓	✓	✓	✓	💾	💾	💾
DivSufSort_PARALLEL$_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DivSufSort_par$_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NaivIps4oParallel	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
NaivParallel	✓	✓	✓	✓	✓	✓	💾	✓	✓	✓	✓	✓	✓	💾	✓	✓	✓	✓	✓	✓	💾
Osipov_parallel_wp	✓	✓	✓	💾	💾	💾	💾	✓	✓	✓	💾	💾	💾	💾	✓	✓	✓	💾	💾	💾	💾
PARALLEL_SAIS	✓	✓	✓	✓	ϟ	ϟ	ϟ	✓	✓	✓	✓	ϟ	ϟ	ϟ	✓	✓	✓	✓	ϟ	🕒	🕒

Tabelle B.7: SA Korrektheit Large parallel Weak-Scaling

- ✓ SA wurde korrekt berechnet.
- ✗ SA wurde falsch berechnet.
- 🕒 Berechnung hat Zeitlimit des Systems erreicht.
- 💾 Berechnung hat Speicherlimit des Systems erreicht.
- ϟ Berechnung brach mit einem Laufzeitfehler ab.
- ⊘ Nicht durch Implementierung unterstützt.

				commoncrawl.txt				dna.txt							wiki.txt						
Threads:	1	2	4	8	12	16	20	1	2	4	8	12	16	20	1	2	4	8	12	16	20
BPR_par	0.47	0.89	1.78	3.76	7.67	11.15	–	0.57	1.09	2.15	4.37	8.48	11.59	–	0.54	1.02	2.06	4.33	8.50	12.43	–
DC3-Parallel-V1	3.46	3.98	4.67	7.53	–	–	–	3.43	3.79	4.77	7.08	–	–	–	3.61	4.03	5.06	7.77	–	–	–
DC3-Parallel-V2	2.98	3.59	4.50	7.26	–	–	–	3.02	3.52	4.49	6.81	–	–	–	3.09	3.64	4.74	7.36	–	–	–
Deep-Shallow_par	6.59	6.24	9.10	10.25	–	–	–	1.84	2.33	2.99	4.11	6.11	8.87	10.88	1.40	1.71	2.17	3.24	4.93	5.53	6.44
Discarding2Parallel	5.59	6.30	7.66	9.77	–	–	–	3.24	3.74	4.43	5.24	–	–	–	3.20	3.64	4.28	5.60	–	–	–
Discarding4Parallel	3.55	4.09	5.03	6.62	–	–	–	2.45	2.95	3.54	4.53	–	–	–	2.33	2.71	3.25	4.40	–	–	–
DivSufSort_PARALLEL$_{ref}$	1.44	2.01	2.59	3.55	5.59	7.31	7.15	1.35	1.71	2.18	2.77	5.37	5.73	7.04	1.29	1.72	2.11	2.95	5.64	5.90	6.93
DivSufSort_par$_{ref}$	0.43	0.78	1.50	3.06	5.33	7.28	11.79	0.51	0.99	2.02	4.14	8.91	11.51	15.62	0.46	0.85	1.66	3.32	5.74	7.91	10.21
NaivIps4oParallel	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
NaivParallel	7.12	7.43	8.46	10.87	14.83	15.92	–	3.06	3.73	4.16	4.63	5.96	5.91	–	2.72	3.08	3.45	4.25	6.01	5.49	–
Osipov_parallel_wp	4.21	5.48	7.58	–	–	–	–	2.95	3.82	5.26	–	–	–	–	2.91	3.68	5.11	–	–	–	–
PARALLEL_SAIS	0.93	1.66	3.42	7.23	–	–	–	1.11	1.81	3.66	7.54	–	–	–	1.07	1.90	3.94	8.11	–	–	–

Tabelle B.8: Laufzeit in Minuten Large parallel Weak-Scaling

Grün Die besten drei Werte.
Rot Die schlechtesten drei Werte.

	commoncrawl.txt							dna.txt							wiki.txt						
Threads:	1	2	4	8	12	16	20	1	2	4	8	12	16	20	1	2	4	8	12	16	20
BPR_par	1.027	1.824	3.606	7.159	20.140	26.817	–	0.784	1.565	3.128	6.255	18.758	25.010	–	0.952	1.733	3.436	6.837	19.647	26.173	–
DC3-Parallel-V1	4.167	8.333	16.667	33.333	–	–	–	4.167	8.333	16.667	33.333	–	–	–	4.167	8.333	16.667	33.333	–	–	–
DC3-Parallel-V2	3.119	8.333	16.667	33.333	–	–	–	3.056	8.333	16.667	33.333	–	–	–	3.105	8.333	16.667	33.333	–	–	–
Deep-Shallow_par	0.003	0.016	0.081	0.393	–	–	–	0.002	0.004	0.009	0.017	0.043	0.057	0.071	0.003	0.005	0.010	0.020	0.050	0.062	0.079
Discarding2Parallel	2.346	4.692	9.383	18.766	–	–	–	2.346	4.692	9.383	18.766	–	–	–	2.346	4.692	9.383	18.766	–	–	–
Discarding4Parallel	3.908	7.817	15.633	31.266	–	–	–	3.908	7.817	15.633	31.266	–	–	–	3.908	7.817	15.633	31.266	–	–	–
DivSufSort_PARALLEL$_{ref}$	0.369	0.725	1.449	2.901	7.897	10.527	13.151	0.296	0.619	1.243	2.336	6.315	8.354	10.552	0.331	0.662	1.323	2.623	7.121	9.499	11.868
DivSufSort_par$_{ref}$	0.000	0.000	0.000	0.000	0.001	0.001	0.001	0.000	0.000	0.000	0.000	0.001	0.001	0.001	0.000	0.000	0.000	0.000	0.001	0.001	0.001
NaivIps4oParallel	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
NaivParallel	0.000	1.563	3.125	6.250	18.750	25.000	–	0.000	1.563	3.125	6.250	18.750	25.000	–	0.000	1.563	3.125	6.250	18.750	25.000	–
Osipov_parallel_wp	6.249	17.183	34.369	–	–	–	–	6.250	17.188	34.375	–	–	–	–	6.248	17.181	34.367	–	–	–	–
PARALLEL_SAIS	0.411	0.514	0.699	1.043	–	–	–	0.399	0.484	0.654	1.015	–	–	–	0.417	0.525	0.732	1.102	–	–	–

Tabelle B.9: Extra-Speicher in GiB Large parallel Weak-Scaling

Grün Die besten drei Werte.
Rot Die schlechtesten drei Werte.

B.2.2 Strong Scaling

	commoncrawl.txt							dna.txt							wiki.txt						
Threads:	1	2	4	8	12	16	20	1	2	4	8	12	16	20	1	2	4	8	12	16	20
BPR_par	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DC3-Parallel-V1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DC3-Parallel-V2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Deep-Shallow_par	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Discarding2Parallel	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Discarding4Parallel	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DivSufSort_PARALLEL$_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DivSufSort_par$_{ref}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NaivIps4oParallel	◴	◴	◴	◴	◴	◴	◴	◴	◴	◴	◴	◴	◴	◴	◴	◴	◴	◴	◴	◴	◴
NaivParallel	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Osipov_parallel_wp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PARALLEL_SAIS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tabelle B.10: SA Korrektheit Large Parallel Strong-Scaling

✓ SA wurde korrekt berechnet.
✗ SA wurde falsch berechnet.
◴ Berechnung hat Zeitlimit des Systems erreicht.
Berechnung hat Speicherlimit des Systems erreicht.
Berechnung brach mit einem Laufzeitfehler ab.
⊘ Nicht durch Implementierung unterstützt.

	commoncrawl.txt							dna.txt							wiki.txt						
Threads:	1	2	4	8	12	16	20	1	2	4	8	12	16	20	1	2	4	8	12	16	20
BPR_par	0.47	0.41	0.37	0.36	0.36	0.37	0.37	0.57	0.50	0.46	0.44	0.44	0.44	0.43	0.54	0.47	0.44	0.42	0.42	0.43	0.43
DC3-Parallel-V1	3.46	1.91	1.20	0.80	0.91	0.79	0.63	3.44	1.86	1.16	0.77	0.86	0.71	0.63	3.58	1.95	1.19	0.80	0.89	0.79	0.59
DC3-Parallel-V2	2.98	1.71	1.08	0.73	0.79	0.63	0.54	3.04	1.72	1.07	0.74	0.78	0.62	0.54	3.09	1.76	1.11	0.75	0.78	0.64	0.56
Deep-Shallow_par	6.59	3.37	1.75	0.95	0.70	0.58	0.50	1.84	1.03	0.59	0.43	0.41	0.31	0.30	1.40	0.77	0.45	0.29	0.25	0.22	0.20
Discarding2Parallel	5.59	2.98	1.66	1.00	0.80	0.69	0.62	3.24	1.73	0.97	0.58	0.46	0.40	0.36	3.20	1.71	0.96	0.58	0.46	0.40	0.36
Discarding4Parallel	3.54	1.93	1.10	0.68	0.56	0.50	0.46	2.45	1.33	0.76	0.47	0.38	0.34	0.31	2.33	1.27	0.72	0.45	0.37	0.33	0.30
DivSufSort_PARALLEL$_{\text{ref}}$	1.44	0.93	0.56	0.34	0.43	0.30	0.23	1.35	0.77	0.46	0.30	0.32	0.28	0.24	1.29	0.78	0.46	0.31	0.38	0.29	0.22
DivSufSort_par$_{\text{ref}}$	0.43	0.36	0.33	0.32	0.32	0.32	0.32	0.51	0.46	0.44	0.44	0.45	0.45	0.45	0.46	0.40	0.36	0.35	0.35	0.35	0.35
NaivIps4oParallel	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
NaivParallel	7.12	5.41	3.23	2.61	1.22	2.07	1.47	3.06	1.54	0.80	0.41	0.35	0.25	0.18	2.71	1.40	0.72	0.37	0.33	0.23	0.17
Osipov_parallel_wp	4.20	2.58	1.61	1.20	1.27	1.08	1.07	2.95	1.80	1.13	0.84	0.91	0.78	0.73	2.91	1.76	1.09	0.81	0.83	0.80	0.72
PARALLEL_SAIS	0.93	0.76	0.74	0.74	0.90	0.90	0.90	1.11	0.92	0.91	0.90	1.05	1.08	1.03	1.08	0.88	0.86	0.85	1.04	0.98	1.05

Tabelle B.11: Laufzeit in Minuten Large Parallel Strong-Scaling

Grün Die besten drei Werte.

Rot Die schlechtesten drei Werte.

	commoncrawl.txt							dna.txt							wiki.txt						
Threads:	1	2	4	8	12	16	20	1	2	4	8	12	16	20	1	2	4	8	12	16	20
BPR_par	1.027	1.027	1.222	1.614	2.005	2.397	2.789	0.784	0.784	0.784	0.786	0.789	0.791	0.793	0.952	0.952	1.087	1.359	1.631	1.904	2.176
DC3-Parallel-V1	4.167	4.167	4.167	4.167	4.167	4.167	4.219	4.167	4.167	4.167	4.186	4.167	4.273	4.167	4.167	4.167	4.167	4.167	4.253	4.232	4.167
DC3-Parallel-V2	3.119	4.167	4.167	4.167	4.167	4.080	4.167	3.056	4.167	4.167	4.610	4.167	4.167	4.167	3.105	4.167	4.167	4.167	4.253	4.242	4.219
Deep-Shallow_par	0.003	0.008	0.018	0.032	0.037	0.035	0.041	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.003	0.003	0.003	0.003	0.003	0.003	0.003
Discarding2Parallel	2.346	2.348	2.352	2.360	2.368	2.376	2.384	2.346	2.348	2.352	2.360	2.368	2.376	2.384	2.346	2.348	2.352	2.360	2.368	2.376	2.384
Discarding4Parallel	3.908	3.910	3.914	3.922	3.930	3.938	3.946	3.908	3.910	3.914	3.922	3.930	3.938	3.946	3.908	3.910	3.914	3.922	3.930	3.938	3.946
DivSufSort_PARALLEL$_{\text{ref}}$	0.369	0.369	0.369	0.369	0.369	0.369	0.369	0.296	0.296	0.296	0.296	0.296	0.296	0.296	0.331	0.331	0.331	0.331	0.331	0.331	0.331
DivSufSort_par$_{\text{ref}}$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
NaivIps4oParallel	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
NaivParallel	0.000	0.781	0.781	0.781	0.781	0.781	0.781	0.000	0.781	0.781	0.781	0.781	0.781	0.781	0.000	0.781	0.781	0.781	0.781	0.781	0.781
Osipov_parallel_wp	6.249	8.592	8.592	8.592	8.397	8.592	8.475	6.250	8.594	9.180	8.887	8.594	8.594	8.594	6.248	8.589	8.589	8.589	8.589	8.589	8.589
PARALLEL_SAIS	0.411	0.411	0.411	0.411	0.411	0.411	0.411	0.399	0.399	0.399	0.399	0.399	0.399	0.399	0.417	0.417	0.417	0.417	0.417	0.417	0.417

Tabelle B.12: Extra-Speicher in GiB Large Parallel Strong-Scaling

Grün Die besten drei Werte.

Rot Die schlechtesten drei Werte.

Literaturverzeichnis

[1] Alok Aggarwal, Jeffrey Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] Nasir Al-Darwish. Formulation and analysis of in-place msd radix sort algorithms. *Journal of Information Science*, 31(6):467–481, 2005.

[3] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory. In *Proceedings of the 29th annual ACM symposium on Theory of computing*, pages 540–548. ACM, 1997.

[4] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: `http://drops.dagstuhl.de/opus/volltexte/2017/7854`, `doi:10.4230/LIPIcs.ESA.2017.9`.

[5] Johannes Bahne, Nico Bertram, Marvin Böcker, Jonas Bode, Hermann Foot, Florian Grieskamp, Marvin Löbel, Oliver Magiera, Rosa Pink, David Piper, and Christopher Poeplau. Github: sacabench. URL: `https://github.com/sacabench/sacabench`.

[6] Uwe Baier. Gsaca repository. URL: `https://github.com/waYne1337/gsaca`.

[7] Uwe Baier. Linear-time Suffix Sorting - A New Approach for Suffix Array Construction. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:12, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: `http://drops.dagstuhl.de/opus/volltexte/2016/6069`, `doi:10.4230/LIPIcs.CPM.2016.23`.

[8] Jon L Bentley and M Douglas McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, 1993.

[9] Jon L Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–369. Society for Industrial and Applied Mathematics, 1997.

[10] Timo Bingmann. SqlPlotTools. URL: `https://github.com/bingmann/sqlplot-tools`.

[11] Guy E. Blelloch. Prefix sums and their applications. Technical report, Synthesis of Parallel Algorithms, 1990.

[12] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[13] Common crawl, `commoncrawl.org`. URL: `http://commoncrawl.org/`.

[14] 1000 Genomes Project Consortium et al. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

[15] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[16] Alex Couture-Beil. Package 'rjson'. URL: `https://cran.r-project.org/web/packages/rjson/rjson.pdf`.

[17] Andreas Crauser and Paolo Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[18] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. *Journal of Experimental Algorithmics (JEA)*, 12:3–4, 2008.

[19] Mrinal Deo and Sean Keely. Parallel suffix array and least common prefix for the gpu. *SIGPLAN Not.*, 48(8):197–206, February 2013.

[20] Technische Universität Dortmund. LiDO3. URL: `https://www.itmc.tu-dortmund.de/cms/de/dienste/hochleistungsrechnen/lido3/index.html`.

[21] Paolo Farragina and Gonzalo Navarro. Pizza&Chili corpus - compressed indexes and their testbeds. URL: `http://pizzachili.dcc.uchile.cl/index.html`.

[22] Paolo Ferragina and Roberto Grossi. The string b-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM (JACM)*, 46(2):236–280, 1999.

[23] Johannes Fischer and Florian Kurpicz. Dismantling divsufsort. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2017*, pages 62–76, Czech Technical University in Prague, Czech Republic, 2017.

[24] Keisuke Goto. Optimal time and space construction of suffix arrays and lcp arrays for integer alphabets. *arXiv preprint arXiv:1703.01009*, 2017.

[25] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[26] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, November 2006. doi:10.1145/1217856.1217858.

[27] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. STOC '72, pages 125–136, 1972.

[28] Clyde P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*, C-32(10):942–946, Oct 1983.

[29] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. volume 30, pages 943–955. Springer, 2003. doi:10.1007/3-540-45061-0.

[30] Julian Labeit, Julian Shun, and Guy E Blelloch. Parallel lightweight wavelet tree, suffix array and fm-index construction. In *2016 Data Compression Conference (DCC)*, pages 33–42. IEEE, 2016.

[31] Labeit, Julian and Shun, Julian. Github: Parallel divsufsort, 2016. URL: https://github.com/jlabeit/parallel-divsufsort.

[32] Bin Lao, Ge Nong, Wai Hong Chan, and Yi Pan. Fast induced sorting suffixes on a multicore machine. *The Journal of Supercomputing*, 74(7):3468–3485, Jul 2018. URL: https://doi.org/10.1007/s11227-018-2395-5, doi:10.1007/s11227-018-2395-5.

[33] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007. doi:10.1016/j.tcs.2007.07.017.

[34] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I Tomescu. *Genome-scale algorithm design*. Cambridge University Press, 2015.

[35] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[36] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[37] Michael A. Maniscalco and Simon J. Puglisi. An efficient, versatile approach to suffix sorting. *J. Exp. Algorithmics*, 12:1.2:1–1.2:23, June 2008. doi:10.1145/1227161.1278374.

[38] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[39] Y. Mori. Divsufsort. URL: `https://github.com/y-256/libdivsufsort`.

[40] David Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27:983–993, 1997.

[41] Erich Neuwirth. Package 'rcolorbrewer'. URL: `https://cran.r-project.org/web/packages/RColorBrewer/index.html`.

[42] Ge Nong. Practical linear-time $\mathcal{O}(1)$-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.*, 31(3):15:1–15:15, August 2013. `doi:10.1145/2493175.2493180`.

[43] Ge Nong and Sen Zhang. Optimal lightweight construction of suffix arrays for constant alphabets. In *Workshop on Algorithms and Data Structures*, pages 613–624. Springer, 2007.

[44] Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011. `doi:http://doi.ieeecomputersociety.org/10.1109/TC.2010.188`.

[45] Vitaly Osipov. Parallel suffix array construction for shared memory architectures. In *String Processing and Information Retrieval*, pages 379–384, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[46] Simon J Puglisi, William F Smyth, and Andrew H Turpin. A taxonomy of suffix array construction algorithms. *acm Computing Surveys (CSUR)*, 39(2):4, 2007.

[47] Arch D. Robison. A Parallel Stable Sort Using C++11 for TBB, Cilk Plus, and OpenMP. URL: `https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp`.

[48] The comprehensive r archive network. URL: `https://cran.r-project.org/`.

[49] Klaus-Bernd Schürmann and Jens Stoye. An Incomplex Algorithm for Fast Suffix Array Construction. In *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics, ALENEX /ANALCO 2005, Vancouver, BC, Canada, 22 January 2005*, pages 78–85, 2005. URL: `http://www.siam.org/meetings/alenex05/papers/07kschuermann.pdf`.

[50] Julian Seward. On the performance of bwt sorting algorithms. In *Proceedings DCC 2000. Data Compression Conference*, pages 173–182. IEEE, March 2000. `doi:10.1109/DCC.2000.838157`.

[51] Johannes Singler and Benjamin Konsik. The gnu libstdc++ parallel mode: software engineering considerations. In *Proceedings of the 1st international workshop on Multicore software engineering*, pages 15–22. ACM, 2008.

[52] Tagme datasets. URL: `http://acube.di.unipi.it/tagme-dataset/`.

[53] The Common Crawl Team. Common crawl, 2019. URL: `http://commoncrawl.org`.

[54] The tudocomp authors. tudocomp - the tu dortmund compression framework. URL: `http://tudocomp.org`.

[55] L. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4(3):348–355, 1975.

[56] Phil Vines and Justin Zobel. Compression techniques for chinese text. *Software-Practice and Experience*, 28(12):1299–1314, 1998.

[57] Vital articles/expanded. URL: `https://en.wikipedia.org/wiki/Wikipedia:Vital_articles/Expanded`.

[58] Jeffrey Scott Vitter and Elizabeth AM Shriver. Algorithms for parallel memory, i: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[59] Leyuan Wang, Sean Baxter, and John D. Owens. Fast parallel skew and prefix-doubling suffix array construction on the gpu. *Concurr. Comput. : Pract. Exper.*, 28(12):3466–3484, August 2016.

[60] Sascha Witt. Github: In-place parallel super scalar samplesort (ips4o). URL: `https://github.com/SaschaWitt/ips4o`.

[61] Francois Yergeau. Utf-8, a transformation format of iso 10646. Technical report, 2003.