External Memory Algorithms for State Space Exploration in Model Checking and Action Planning

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik
von

Shahid Jabbar

Dortmund

2008

Tag der mündlichen Prüfung: 6. Juni 2008

Dekan/Dekanin: Prof. Dr. Peter Buchholz

Gutachter: Priv. Doz. Dr. Stefan Edelkamp,
Prof. Dr. Bernhard Steffen.

Dedicated to the wonderful and peace-loving people of Deutschland, meine zweite Heimat.

Abstract

RAM is a scarce resource. Several real-world problems in model checking and action planning are beyond the reach of traditional RAM-based algorithms, due to the so-called state space explosion problem. This dissertation aims at designing a set of algorithms that mitigates the memory bottleneck problem in model checking and planning, through a controlled and efficient usage of secondary storage mediums, such as hard disks. We consider a broad variety of system models ranging from simple undirected and unweighted state spaces to highly complex Markov decision processes (MDP). Path/cycle search problem in the case of deterministic state spaces and policy search problem in the case of MDPs are the focal points of this thesis. The state spaces, or the implicit graphs, are not provided beforehand, but are generated on-the-fly through a set of initial states and a set of transformation rules. The proposed algorithms belong to the category of External Memory (EM) algorithms and are analyzed for their asymptotic I/O complexity.

An EM guided search algorithm, called External A* (for being derived from the famous Best-First Search algorithm A*), is developed. External A* distinguishes itself from other external guided search approaches by being completely oblivious to the state space structure. Directed model checking has proved itself to be very effective in delivering shorter error trails and in memory savings. We incorporate external search into automata-based LTL model checking of concurrent systems through an extended variant of External A*. Accepting cycle detection lies at the heart of LTL model checking. Due to the inherent difficulty in cycle search in large graphs, earlier disk-based approaches distanced themselves from taking care of the full LTL model checking. In this dissertation, two algorithms for accepting cycle detection are put forward: a blind search algorithm based on Breadth-First traversal, and a guided algorithm evolved from External A*. To be able to utilize the full potential of modern multi-core architectures and easily accessible networks of workstations, External A* is further extended into a distributed algorithm. For model checking large real-time systems and optimal real-time scheduling, EM algorithms for exploration in timed automata and priced timed automata are presented.

Graph-based action planning methods have achieved a significant level of maturity in the field of planning and scheduling. To integrate external heuristic search into planning, External Enforced Hill-Climbing is contributed. For optimal planning in PDDL3 domains involving preferences, a Cost-Optimal External Breadth-First Search is proposed. Nondeterministic and probabilistic state spaces are encountered both in model checking of stochastic systems and in planning under uncertainty. In such state spaces, one is interested not in a path but rather in a policy that maximizes the reward in reaching to a goal state. Due to the

back-propagation of information in policy search, no efficient disk-based solution was ever contributed. We present an EM algorithm based on the standard Value Iteration procedure for policy search. The algorithm, External Value Iteration, is able to solve Bellman equations not only for large MDPs, but also for AND/OR graphs and Game trees.

The algorithms developed in this dissertation have been successfully integrated in some state-of-the-art tools including the SPIN model checker, MIPS-XXL (based on FF) planning system and UPPAAL-CORA for real-time scheduling. The largest reported exploration consumed 3 Terabytes of hard disk, while using only 3 Gigabytes of RAM lasting for 479 hours – time went down to 196 hours when 4 processors were engaged.

Acknowledgements

In Germany, a Ph.D supervisor is called '*Doktorvater*' (Doctor Father). Priv. Doz. Dr. Stefan Edelkamp has been more than just my Doktorvater – he has been my Doktorfreund too. I am extremely lucky to have had a supervisor that always had time to exchange ideas. His dedication to the scientific development and his unmatched ability to go into the depths, has been a guiding principle for me.

Thanks are due to Prof. Bernhard Steffen for his helpful comments throughout my research. His amazing energy and his ambition to bridge the gap between the industry and the academia have always been a source of inspiration to me. I also extend my gratitudes to other members of my committee: Prof. Jan Vahrenhold and Dr. Henrik Björklund.

I am grateful to the anonymous referees of all the papers (accepted and rejected) that helped improve my scientific reasoning and writing. I thank Stefan Schrödl and Blai Bonet for the fruitful collaborations, and Gerd Behrmann and Kim Larsen for sharing the code of UPPAAL-CORA. Deepak Ajwani, Shah Jamal Alam, Alberto Lluch-Laufente, Clemens Renner, and Pavel Šimeček deserve special thanks for their proof-reading that improved the quality of this dissertation. Despite their helpful efforts, errors may still remain – they would be entirely on my part. Oliver Rüthing, thank you for discussions on modeling formalisms. I would like to express my sincere gratitude towards Markus Bajohr and Mathias Weiß for answering my countless questions about computing hardware and operating systems, and towards Marco Bakera and Clemens Renner for broadening my knowledge of temporal logics. I cannot thank Alberto enough for his prompt replies to my emails, while doing the 'dissection' of SPIN. I acknowledge the contributions of my master's students, particularly, Mohammed Nazih in the planning system, Damian Sulewski in the C++ verification system, and Mark Kellershoff for drawing some of the diagrams.

Clemens – once again – thank you for the long sessions of *Kicker* that refreshed my 'External Memory' and for the unlimited supply of *Kekse*. I thank the whole of Lehrstuhl-5, particularly, Rita Bartels, Claudia Herbers, Sven Jörges, Martin Karusseit, Christian Kubczak, Kirsten Lindner-Schwentick, Ralf Nagel, Harald Raffelt, Felicitas Schulze, Thomas Wilk, and Holger Willebrandt for creating an inspiring and cheerful atmosphere. Thanks are due to Peter Kissmann for maintaining a pleasant working climate and for tolerating me in his room. My long-term colleague and roommate Tilman Mehler and his wife Frida deserve special thanks for friendship and encouragement. I am deeply grateful to Naveed Ahmed, Shah Jamal Alam, and Imran Rauf for years of friendship and many stimulating exchanges of ideas during my numerous visits to Saarbrücken. My life in Dortmund could have been very difficult without Ulrike, who took care of me during my stay in Dortmund and introduced me to

the wonderful culture of Germany. Toni's pizza parlor is thanked for catering to the dietary needs of a Ph.D. student by delivering one tasty pizza after the other with low cheese and extra Tabasco during External Memory experiments running into the small hours.

I am deeply indebted to Karolina, without whom this thesis would never have become a concrete reality. I wish I would be able to reciprocate her dedication to my work some day. I not only thank her for the moral support, but also for several grammatical corrections and stylistic improvements in this document.

Thanks to my family in Pakistan: Jabbar, Nelofar, Javed, Samreen, Humera, Danish, and Ayesha for their love and support. I am specially grateful to my mother for her cheerful and uplifting words. I thank Birgitta and Clas for showing me the charm of learning and knowing about the world outside computer science.

Finally, I would like to acknowledge *Deutsche Forschungsgemeinschaft* (German Research Foundation) for financially supporting my research through projects *Directed Model Checking*, *Heuristic Search*, and *Algorithm Engineering*.

Table of Contents

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

RAM is a scarce resource. Many real world problems remain unsolvable because of the bottleneck of limited memory. Although, with the advent of 64-bit processors, the memory address limit has been increased to 16 Exibytes (16,777,216 Terabytes), most modern computing machines and operating systems support only 64 Gigabytes of physical memory. On the other hand, recent research in magnetic disk media has made large amounts of memory easily available – and, most importantly, at affordable prices. At the time of writing, 2GB of RAM cost around 180 Euro, while a 500 GB hard disk is available for mere 80 Euro.

Random Access Memory (RAM), as the name suggests, supports fast and constant time *random* accesses to any memory address. This access time is usually referred to as *latency* time. A hard disk, on the contrary, is a mechanical device; accessing a certain address requires physical movements of parts. Consequently, the latency time for accessing different data items could vary considerably, depending on the relative position of the read-write head and of the data.

One victim of this memory bottleneck problem is the *path search* in graphs. Several problems in computer science can be modeled as path search problems in graphs that aim at finding an optimal path from an initial node to a target node. For many decades, a number of efforts have been made to reduce the time and space complexity of search problems. Nonetheless, for very large graphs, the problem of limited internal memory appears to be a real bottleneck. The general purpose methods like Virtual Memory management schemes can, in fact, result in a drastic slowdown of the performance. Such methods are ignorant of actual memory accesses in the algorithm. Hence, the predictions on the *locality-of-references* are highly likely to go wrong. This situation calls for specialized and better informed algorithms that can utilize the hard disk memory in an efficient way, by explicitly managing the movement of data between the fast RAM and the slow disk. Such algorithms are categorized under the name of *External Memory algorithms* (Meyer, Sanders, & Sibeyn 2003).

Model checking and *action planning* are two such sub-disciplines of computer science that use graph search algorithms as underlying procedures. Model checking (Clarke, Grumberg, & Peled 1999; Müller-Olm, Schmidt, & Steffen 1999) is an exhaustive automated procedure to formally verify whether a system conforms to some required behavior or not. The dependence of humans on software is rising at an astonishing rate. This dependence, in turn,

highlights the significance of 'correct' or 'verified' software systems. Although the need for verification is crucial to *every* system, particularly in environments where a small error in the running system could have a direct or indirect impact on human lives, this need is *incalculable*.

With the advent of multi-core machines, there has been a wide interest in parallel algorithms. Sharing of common resources, such as main memory, is an important topic in the design of parallel algorithms. Model checking provides the necessary means to formally verify if the resources will be correctly shared among many concurrently running processes. The success of model checking in dealing with the verification of large systems has made it an active area of research in the last decades. Especially during the last twenty years it has evolved into one of the most successful verification techniques. Examples range from mainstream applications such as protocol validation, software and embedded systems' verification to areas such as business work-flow analysis, and scheduler synthesis and verification.

Action planning (Russell & Norvig 2003) is a subfield of Artificial Intelligence. It deals with finding a sequence of actions to achieve a required goal in an environment. Examples include scheduling in logistics domains, puzzles' solving and planning robots to perform various tasks efficiently. Both model checking and action planning suffer from the so-called *state space explosion* problem – for large graphs, the search is doomed to failure due to limited RAM!

1.2 State spaces

This dissertation is about external memory algorithms for search in state spaces. Before discussing state spaces, the notion of a state should be defined.

Definition 1.1 (State) *A state is a bit-vector encoding the overall configuration of a system at a particular instant.*

A synonym is a *snapshot*. An easy intuition can be derived from a board game, for example, Chess or Checkers. A state in such games is a set of configurations of all the pieces on the board.

Definition 1.2 (State Space) *A state space, written $\mathcal{S}$, is the set of all reachable configurations of a system.*

State spaces are provided in the form of one or more *initial states*, and a set of *transition rules* that transform the contents of a state when applied. A repetitive application of these transitions generates a graph where the nodes correspond to the states and the edges to the transformation rules. The process of applying the transition rules is referred to as *successor generation* or *expansion*. The graphs thus generated are referred to as *implicit graphs*. On the other hand, a network of nodes and edges that we refer to as *explicit graphs*, are fully provided beforehand in the form of adjacency lists/matrices. Examples include a road network, a network of computers, etc.

Figure 1.2 depicts an example of an implicit graph for 8-puzzle sliding-tiles game. Given an initial configuration of tiles, the available operations, up, down, left, and right are successively applied on the *blank* space, until the goal is reached (tiles are aligned). The total number of distinct reachable states in 8-puzzle are $9!/2 = 181,440$. For 15-puzzle, where a

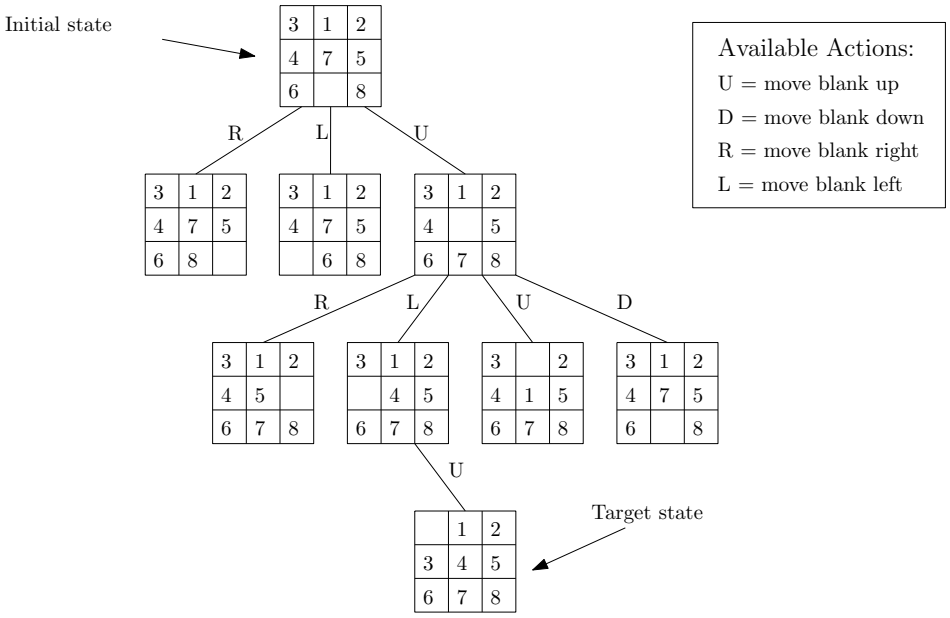

Figure 1.1: A fragment of the state space of 8-puzzle. Starting from an initial random configuration, a search is performed for the target state.

square of 4×4 is used, the reachable states shoot up to $16!/2 = 10,461,394,944,000$. This phenomenon corresponds to what is usually referred to as *"state space explosion"*.

The state space in the figure induces an *undirected* graph, i.e., it is always possible to reach the previous configuration by applying the inverse rule: left for right, and up for down. Moreover, there is no cost associated with the application of any of the rules. This leads to the categorization of our state space as inducing an *unweighted*, undirected graph. However, *all state spaces are not created equal*. It is not always possible to apply the inverse transformation rule. The graphs thus induced are *directed*. Similarly, the use of transformation rules might carry a cost or a penalty, leading to *weighted* state spaces.

1.3 External Memory Algorithms

In practice, algorithms significantly deviate from their theoretical asymptotic upper bounds when applied on large inputs. The reason is the unstructured access to data at different memory levels. The disparity grows manifold, once the data movement crosses the internal memory boundaries and mechanical devices, such as hard disks, have to be involved.

External Memory algorithms correspond to the set of algorithms that explicitly manage the transfer of data between different levels of memory hierarchy: starting from the fast CPU registers and ranging to the secondary storage devices such as hard disks. Such algorithms are analyzed in a separate computational model called External Memory model (Aggarwal & Vitter 1988), instead of the usual von Neumann/RAM model. The algorithms are evaluated on the asymptotic number of Input/Output operations between different memory levels. The most interesting communication bottleneck exists between the fast internal memory and the slow hard disks – a focal point of this thesis. Chapter 2 is dedicated to a discussion on External Memory algorithms. We will examine some basic search algorithms designed

for explicit graphs and implicit graphs.

1.4 Problem Formulation

1.4.1 Conditions

In the following, the conditions that constitute different state space models addressed in this thesis are enumerated. For each condition, only one of the sub-enumerations (a, b, c, ...) is allowed to be selected exclusively.

C1. a discrete state space $\mathcal{S}$,

C2. a set of initial states $\mathcal{I} \subseteq \mathcal{S}$,

C3. a non-empty subset of target/goal/terminal states $\mathcal{T} \subseteq \mathcal{S}$,

C4. Transition relation:

a. a symmetric transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ that defines a direct transition between two states:

$$\forall s_1, s_2 \in \mathcal{S}, (s_1, s_2) \in \mathcal{R} \iff (s_2, s_1) \in \mathcal{R}.$$

A state space with a symmetric transition relation is referred to as an *undirected* state space.

b. an asymmetric transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$:

$$\exists s_1, s_2 \in \mathcal{S}, (s_1, s_2) \in \mathcal{R} \wedge (s_2, s_1) \notin \mathcal{R}.$$

An asymmetric transition relation induces a *directed* state space.

A *path* between two states s_0 and s_n is a sequence of states $s_0, s_1, s_2, \ldots, s_n$, such that for all $0 \leq i \leq n-1$, $(s_i, s_{i+1}) \in \mathcal{R}$.

A state s is *reachable*, if there exists a path $s_0, s_1, s_2, \ldots, s_n$, such that $s_0 \in \mathcal{I}$, $s_n = s$ and for all $0 \leq i \leq n-1$, $(s_i, s_{i+1}) \in \mathcal{R}$. The *reachability problem* refers to the problem of finding a path between two states.

A reachable *cycle* is a lasso-shaped path that is rooted at one of the initial states and visits at least one state on the path twice. The *cycle detection problem* refers to finding a reachable cycle.

C5. Weight function:

a. a weight function $w : \mathcal{R} \to \{1\}$ that assigns a uniform weight of 1 to each transition. State spaces with such a weight function are referred to as *unweighted* state spaces.

b. a weight function $w : \mathcal{R} \to \{1, 2, \ldots C\}$ that assigns a positive integer weight in the range $[1, C]$ to each transition.

c. a weight function $w : \mathcal{R} \to \mathbb{R}$ that assigns a real weight to each transition.

State spaces where either of the last two conditions are satisfied are categorized as *weighted* state spaces; for condition b, they are further classified as being *bounded weighted*.

A shortest path function between two states $s, t \in \mathcal{S}$ given by the Bellman equation:

$$\delta(s,t) = \min_{(s,s')\in\mathcal{R}} \left\{ w(s,s') + \delta(s',t) \right\}.$$

C6. Nature of cost function:

a. w is monotone, or in other words, it satisfies the triangular property:

$$\forall s_1, s_2, s_3 \in \mathcal{S} : \delta(s_1, s_3) \leq \delta(s_1, s_2) + \delta(s_2, s_3).$$

b. w is non-monotone, i.e., the triangular property is not satisfied. In this case, cost is not accumulated on the path, but is evaluated for each state based on its contents.

It is assumed that an internal memory/RAM of limited size $M \ll |\mathcal{S}|$ is available. Similarly, the availability of a hard disk with sufficient space to keep the generated state space is a necessary condition.

1.4.2 Problem Models

The following state space models will be considered in this thesis.

M1. **Undirected and unweighted state spaces:** ⟨C1, C2, C3, C4a, C5a, C6a⟩ *Example:* single-agent games like 15-puzzle, Towers of Hanoi, Rubik's cube, etc.

M2. **Undirected and bounded weighted state spaces:** ⟨C1, C2, C3, C4a, C5b, C6a⟩.

M3. **Directed and unweighted state spaces:** ⟨C1, C2, C3, C4b, C5a, C6a⟩.

M4. **Directed and bounded weighted state spaces:** ⟨C1, C2, C3, C4b, C5b, C6a⟩. *Example*: Automata-based Linear Temporal Logic (LTL) model checking.

M5. **Directed and weighted state spaces with monotone real cost:** ⟨C1, C2, C3, C4b, C5c, C6a⟩ *Example*: Scheduling with priced timed automata.

M6. **Directed and weighted state spaces with non-monotone real cost:** ⟨C1, C2, C3, C4b, C5c, C6b⟩. *Example*: Action planning in the presence of preferences and temporal constraints.

M7. **Probabilistic and non-deterministic state spaces:** All the above-mentioned models share *determinism* as a common property – the output of each action is fully determined. In this model, we consider a more general class of state spaces, namely *non-deterministic* and *probabilistic* state spaces. In these state spaces, the outcome of an action is not fully determined and is stochastic. An action, when applied to a given state, may succeed with a probability p, while it might not have any effect with a probability $1 - p$. In order to define these state spaces, we need some new notations and terminologies. To keep the introduction simple, the formalism is not reproduced here, but will be presented later in a unified search model in Chapter 10. The unified search model is general enough to cover AND/OR graphs, Game trees, and Markov decision processes. In these state spaces, one is interested in a *policy* (most profitable action assignment to states) – rather than a start-target path – that maximizes the reward for reaching a target state. *Example*: Markov decision processes (MDP) as they appear in probabilistic model checking and planning under uncertainty.

1.4.3 Objectives

This dissertation aims at designing External Memory algorithms that are able to:

- solve a reachability or path search problem in models M1 to M6,
- solve a cycle detection problem in models M1 to M4, and
- solve a policy search problem in model M7.

1.4.4 Evaluation of the Algorithms

The algorithms presented in this dissertation will be evaluated on the following criteria.

1. Cost-optimality: whether the algorithm delivers a path that is optimal with respect to a certain cost function.
2. I/O efficiency: the asymptotic number of Input/Output operations the algorithm will perform until it terminates.

1.5 Organization of the Thesis

In Chapter 2, we will discuss the External Memory model and some basic algorithms defined in it. External Memory algorithms are evaluated on the asymptotic number of I/O operations they incurr. In this regard, we will also study a formal framework to analyze the I/O complexity of External Memory algorithms.

The next chapter, Chapter 3, addresses the path search problem in model M1 (undirected and unweighted state spaces). The internal heuristics search algorithm A* is extended to explicitly work on External Memory. The algorithm that we term as External A*, achieves efficient I/O performance by working on sets of states rather than on individual states. We evaluate the proposed algorithm on several instances of a single-agent game that has an underlying undirected graph. This chapter is based on the following publications:

- Stefan Edelkamp, Shahid Jabbar, and Stefan Schrödl, *External A**. In Twenty-Seventh German Conference on AI (KI'04) by Biundo, Frühwirth and Palm (Eds.). Lecture Notes in Artificial Intelligence (LNAI), vol. 3238, pages 226–240, Springer, Ulm, Germany, August 2004.
- Stefan Edelkamp, Shahid Jabbar, and Stefan Schrödl, *External A**. Technical Report, 785, University of Dortmund, April 2004.

From the Chapter 4 onwards, the thesis is divided into two parts: Model checking and action planning. The details of each part are given in the following.

1.5.1 Part I: Model Checking

Chapter 4 provides an introduction to model checking. We discuss a modeling formalism to describe protocols and concurrent systems, and how the formalism can be exploited for model checking. To describe the specification properties that one would like to check in a system, we discuss the use of Linear Temporal Logic (LTL). The usage of heuristic search for

guidance in model checking algorithms is introduced – an approach that has been christened *directed model checking* by Edelkamp, Leue, & Lluch Lafuente (2004). Moreover, we discuss different heuristic estimates that can be used for guidance during model checking.

Chapter 5 extends External A* to work on directed and weighted graphs as they appear in LTL model checking. Extensions for the three models M2, M3, and M4 will be presented. We will restrict ourselves to *safety properties* in this chapter. Safety properties are used for specifications of the form "nothing bad will ever happen" and can be reduced to the problem of finding a particular state in a graph. This chapter extends many results from the paper:

- Shahid Jabbar, Stefan Edelkamp, *I/O Efficient Directed Model Checking*. In Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05) by Cousot (Ed.). Lecture Notes in Computer Science (LNCS), vol. 3385, pages 313–329, Springer, Paris, France, January 2005.

In temporal logic, *Liveness properties* correspond to the specifications that can be stated as "something good will eventually happen". Unlike safety properties, liveness properties need to look for a lasso-shaped path that violates the property. In Chapter 6, we extend External A* for liveness properties. The cycle detection algorithm thus developed is applicable to the models M1 – M4. We also present some new heuristic functions for improved guidance during the search. This chapter formulates the discussion presented in the paper:

- Stefan Edelkamp, Shahid Jabbar, *Large-Scale Directed Model Checking LTL*, In Thirteenth International Spin Workshop on Model Checking Software (Spin'06) by Valmari (Ed.). Lecture Notes in Computer Science (LNCS), vol. 3925, pages 1–18, Springer, Vienna, Austria, March 2006.

As External Memory algorithms work on sets of states rather than on individual states, they are ideal candidates for distributed processing. In Chapter 7, we present a distributed variant of External A* that is capable of working on both multi-core machines and on multiple machines distributed across a network. *Distributed External A** is applicable to models M1–M4, and with some refinements, to M5 and M6 too. In this chapter, results from the following publication are explained in more detail:

- Shahid Jabbar, Stefan Edelkamp, *Parallel External Directed Model Checking with Linear I/O*. In Seventh International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'06) by Emerson and Namjoshi (Eds.). Lecture Notes in Computer Science (LNCS), vol. 3855, pages 237–251, Springer, Charleston, SC, USA, January 2006.

Systems that involve *time* pose a great challenge to the model checking community. Time-critical safety systems frequently appear in every-day life. A common example is a set of traffic signals at a crossing that need to follow strict timing requirements. Any error in such systems could be fatal for humans. The usual formalism to model timed systems is *timed automata*. It is an extension of ordinary finite automata with real-valued clocks and constraints defined on clock values. An extension of timed automata is *priced timed automata*, where the automata are annotated with cost variables. It has been shown in (Larsen *et al.* 2001) that priced timed automata can be used for optimal scheduling problems. In Chapter 8, we discuss the external search in both timed automata and priced timed automata. The new algorithms cover the model M5. This chapter is based on the following publication:

- Stefan Edelkamp and Shahid Jabbar, *Real-Time Model Checking on Secondary Storage*. In ECAI (European Conference on AI) Fourth Workshop on Model Checking and Artificial Intelligence (MoChArt'06), Rival del Garda, Italy, Aug. 2006. Post-Proceedings published in Lecture Notes in Artificial Intelligence (LNAI), vol. 4428, pages 67–83, Springer, 2007.

1.5.2 Part II: Action Planning

Action Planning shares many similarities with model checking. The states in both model checking and planning consist of a set of atomic propositions that hold in that state. A planning problem is usually provided in a well-formed language called Planning Domain Definition Language (PDDL). Contrary to model checking, where we actually have a set of start states, in Action Planning we work on a single start state and search for a path to a state that fulfills the desired goal criteria. The underlying graph is directed and sometimes weighted. In Chapter 9, external search algorithms are integrated into action planning, while covering model M6. We present a sub-optimal guided planning algorithm that prioritizes speed over the optimality of the solution. The algorithms are implemented in the MIPS-XXL planner (Edelkamp, Jabbar, & Nazih 2006). It was granted a *Distinguished Performance Award* in the 5^{th} international planning competition (IPC-5) in 2006. This chapter consolidates the discussion on external planning from the following manuscripts:

- Stefan Edelkamp and Shahid Jabbar, *Cost-Optimal External Planning*. In Twenty-First National Conference on Artificial Intelligence (AAAI'06), pages 821–826, AAAI Press, Boston, MA, USA, July 2006.

- Stefan Edelkamp, Shahid Jabbar, and Mohammed Nazih, *Cost-Optimal Planning with Constraints and Preferences in Large State Spaces*. In ICAPS (International Conference on Automated Planning and Scheduling) Workshop on Preferences and Soft Constraints in Planning, The English Lake District, Cumbria, U.K, June 2006.

Probabilistic and non-deterministic state spaces, i.e., state spaces that instantiate from model M7, are dealt with separately in Chapter 10. This chapter introduces a unified search model. An External Memory variant of the *Value Iteration* algorithm to optimally solve policy search problems in the new model, is then presented. This chapter stems from the following publications:

- Stefan Edelkamp, Shahid Jabbar, and Blai Bonnet, *External Memory Value Iteration*. In Seventeenth International Conference on Automated Planning and Scheduling (ICAPS'07), Pages 128–135, AAAI Press, Providence, Rhodes Island, USA, September 2007.

- Stefan Edelkamp, Shahid Jabbar, and Blai Bonnet, *External Memory Value Iteration*. Technical Report No. 813. Universität Dortmund. April 2007.

Finally, we draw conclusions and discuss future extensions.

Chapter 2

External Memory Model and Algorithms

External Memory algorithms, sometimes also called *Secondary Memory algorithms* or *Out-of-core algorithms* (May 2001), refer to the set of algorithms that explicitly manipulates the access to the secondary storage medium. Such algorithms are capable of dealing with massive data sets that cannot fit into the main memory. They work on sets of data or blocks and are designed in such a way so as to exploit the temporal and spatial locality in data accesses. In this chapter, we study an analysis technique that allows us to evaluate the performance of an algorithm based on the number of I/O operations it performs on a given input.

Structure of the Chapter: We start this chapter by a discussion of the von Neumann model, the traditional *RAM* model, and address its limitations in dealing with massive data. We also discuss the problems with the analysis of algorithms that are based on the von Neumann model. One common solution to deal with large data sets is to exploit a facility provided by most modern operating systems, called *Virtual Memory Management Scheme*. We will study its use and limitations. Then, we discuss the *External Memory* model as proposed by (Aggarwal & Vitter 1988) and the basic analysis techniques in this model. To get a flavor of the working of *External Memory algorithms*, we have chosen Breadth-First Search in large explicit graphs, i.e., the graphs that are provided in the form of edge lists.

We will then switch our attention to implicit graphs, i.e., the graphs that are generated *on-the-fly* from a set of rules and a set of initial states. In this thesis, we are mainly concerned with such implicit graphs. We discuss the Breadth-First Search in implicit graphs and perform a detailed I/O complexity analysis on the algorithm.

Notations: In the following, the vertices and edges of an implicit graph are termed as *states* $\mathcal{S}$ and *transitions* $\mathcal{R}$ to distinguish them from their explicit graph's counterparts – *nodes* V and *edges* E. The adjacent nodes in an explicit graph are accessible by the function *Nbrs*, while the successor states in an implicit graph are generated by a successor generation function *Succ*.

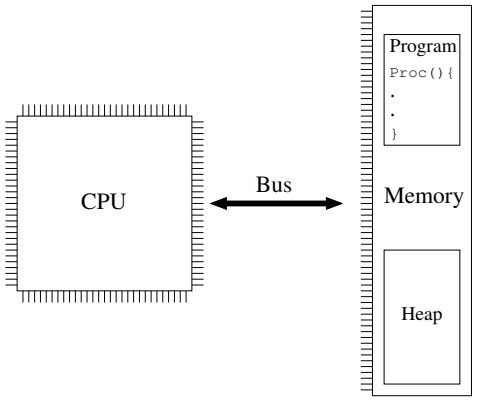

Figure 2.1: von Neumann/RAM model.

2.1 von Neumann/RAM Model

John von Neumann first presented his idea of a computer architecture in 1945 in a report for making EDVAC (Godfrey & Hendry 1993). It assumes a *central processing unit*, an *arithmetic logic unit*, an *Input/Output controller*, a *memory module*, and a *bus* to connect all these components. The most critical assumption is that, during the execution of a program, all the memory required by a process to work on its input is available, i.e., the size of the memory is greater than or equal to the size of the program *plus* the size of the heap allocated by the program. Figure 2.1 depicts the architecture as proposed by von Neumann, where a CPU is connected to a memory module through a bus. The memory holds both the program and its allocated heap.

Problem with the von Neumann/Random Access Model: The von Neumann model, which is sometimes also referred to as Random Access model, assumes that any part of the data required by the program can be accessed randomly and in $O(1)$ time. The current field of analysis of algorithms, as we know it today is based on exactly the same assumption i.e., all operations can be executed in a constant time. However, this assumption is not quite true for modern computer architectures, where there is a hierarchy of memory levels: starting from the fast registers in-built into the CPU to the slow secondary storage devices such as hard disks.

Another apparent problem is the rise in processor speeds that could consume all the available RAM in a matter of minutes. For example, a search algorithm working on states of just 100 bytes and generating an average of 50,000 new states per second will consume 2 GB in mere 6 minutes. These situations call for special approaches to deal with problems that have a huge memory requirement.

2.2 Virtual Memory

Most modern operating systems provide a large consistent address space through a mechanism called *Virtual Memory* (VM). This large address space is divided into sections referred to as *memory pages* that reside on the hard disk, or when in use, in the main memory. When

a memory access request is issued from the CPU, the *Virtual Memory Manager* checks if the corresponding address is in the RAM or not. If it is, the issued VM address is translated into an address in the RAM and the contents are returned. If the address is not in the RAM, the VM Manager issues a *Page Fault* interrupt. It then finds the memory page containing the issued address and moves the whole page from the hard disk to the RAM and returns the contents to the CPU.

The success of this simple mechanism is due to the fact that many application programs, used by common users, exhibit *locality of references* in their pattern of memory accesses. This implies that data residing in a few pages will be repeatedly referenced for a while, before the program shifts attention to another working set; a typical scenario being a user working simultaneously on a word processor, a spread sheet program, and on a drawing tool. Only one of the programs is used at a time and the other two programs can be swapped to the hard disk.

By nature, however, these methods are general-purpose. For programs that lack locality of references, e.g., search algorithms, virtual memory scheme would end up continuously swapping the data in and out of memory – a situation commonly known as *thrashing*. A sever performance degradation in such scenarios is inevitable.

2.3 Hard Disks

As we go down the memory hierarchy, starting from the fast CPU registers to hard disks, we observe two major phenomenon. Firstly, the size of the memory type that can be handled by a typical modern computer increases – compare typical registers size to typical hard disks' size. Secondly, the cost in terms of money per data-unit decreases – compare the price differences in one Gigabyte of RAM and one Gigabyte of hard disk space. But this price difference comes with its own catch.

A hard disk consists of a rotatable platter on which data are written as magnetic polarity differences. Data are arranged in the form of circular tracks that are partitioned in sectors. The reading and writing is done through a read/write head connected to an armature that can move along its axis. With the rotating platter and a moving armature, it is possible to position the head at any point on the surface of the platter. An illustration of the mechanism is provided in Figure 2.2.

The time required to read a data from the hard disk is actually divided into two parts: seek time and data transfer time. *Seek time* is the time required to move the head from its current position to the new position, while *data transfer time* corresponds to the actual reading/writing of the data. A typical modern day hard disk with a SATA-II (Serial Advanced Technology Attachment) capability provides a seek time of around 10 milliseconds while offering a data transfer rate of about 60 MB/sec (though the interface supports 300 MB/sec). Hence, in order to read one single byte of data, the time required would be $0.01 + 1.6 \times 10^{-8}$ seconds. Clearly, the bottleneck lies in the seek time of 10 milliseconds, which can be reduced by decreasing the frequency by which the head has to move from one position to another.

2.4 External Memory Model

External Memory models aim at capturing the data blocks movement between different levels of memory. One of the earliest mentioning of an External Memory model in the lit-

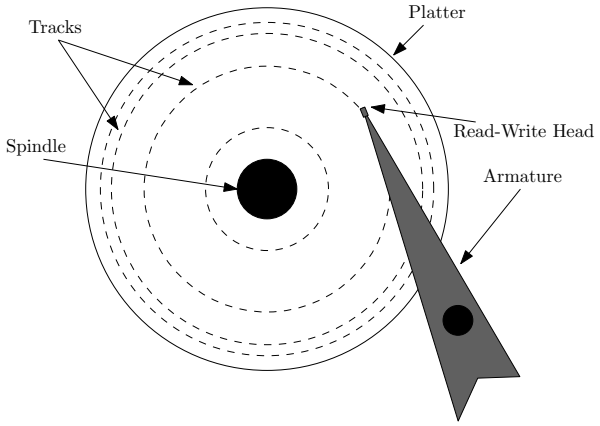

Figure 2.2: A hard disk.

erature can be traced back to Floyd's paper on *Permuting information in idealized Two-Level Storage* (Floyd 1972). Knuth in his monumental book series on *The Art of Computer Programming* (Knuth 1981) also discusses several external sorting methods for data records that cannot fit into the main memory.

In this work, we will deal with the computation model by Aggarwal & Vitter (1988) that we will henceforward refer to as the External Memory or EM model. It provides the necessary formalism to analyze an algorithm based on its input/output (I/O) performance between any two levels of memory, denoting the higher or faster level as internal memory and the slower as external memory. The model covers transfers between registers and caches, between caches and RAM, RAM and external storage devices, etc. The external storage devices may include hard disks or any other random access media.

We are particularly interested in the communication between the RAM and the hard disk. Hence, in the subsequent discussion, we will use the terms *external memory* and *hard disk* interchangeably. Similarly, the terms *internal memory* and *RAM* are also used synonymously.

The main parameters of the model are:

M**:** The size of the internal memory or RAM.

N**:** The size of the input residing on the hard disk.

B**:** The size of the data block that can be transferred from RAM to the hard disk and from hard disk to the RAM in one single I/O operation.

We furthermore assume that $N \gg M$ and $B < M$. It is usually assumed that at the beginning of the algorithm, the input data are stored as contiguous blocks on external memory, and the same must hold for the output. Only the number of blocks reads and writes are counted, computations in internal memory do not incur any cost.

An extension of the model considers D disks that can be accessed simultaneously. In the External Memory model by Aggarwal & Vitter (1988), the presence of D disks increases the bandwidth of the I/O channel to $D \cdot B$. Figure 2.3 depicts the External Memory model for D disks, all of them capable of simultaneously transferring data in blocks of size B to the RAM.

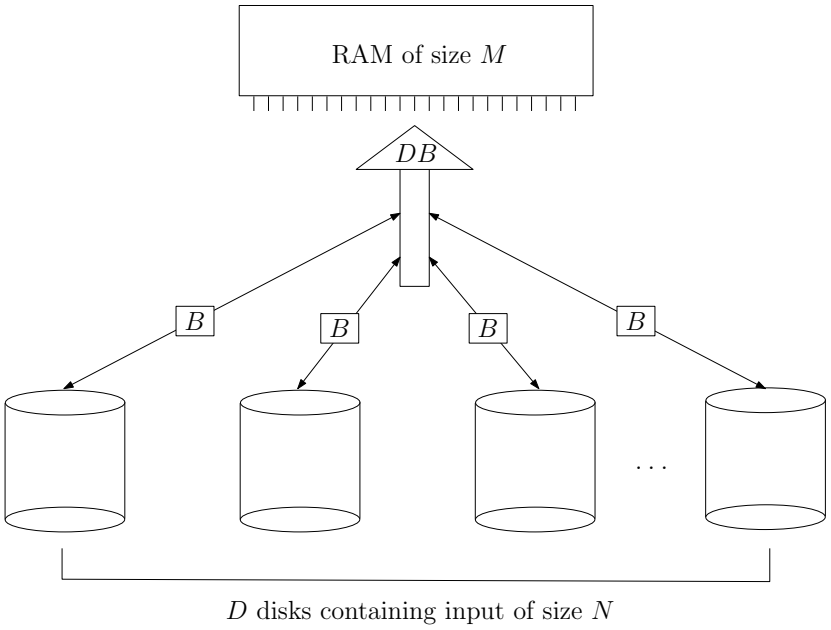

Figure 2.3: Aggarwal and Vitter External Memory Model.

One distinguishes between two general approaches to External Memory algorithms: either one can devise algorithms to solve specific computational problems, while explicitly controlling secondary memory access; or, one can develop general-purpose external memory data structures, such as stacks, queues, search trees, priority queues, and so on, and then use them in algorithms that are similar to their internal-memory counterparts. In this thesis, we have chosen the former approach of designing specific External Memory algorithms for search in different types of state spaces.

2.5 Basic Primitives of I/O-Efficient Algorithms

The complexity of External Memory algorithms is measured by the asymptotic number of I/Os they perform. It is often convenient to express the complexity of External Memory algorithms using a number of frequently occurring primitive I/O operations such as scanning the whole input and external sorting.

The simplest operation is *external scanning*, which means reading a stream of records stored consecutively on secondary memory. It can be achieved by sequentially reading the input of size N in blocks of size B and transferring them to the main memory. The number of I/Os is $\Theta(\frac{N}{B})$ for single disk. In this case, it is trivial to exploit disk- and block-parallelism. Given D disks containing the input, external scanning can be achieved in $\Theta(\frac{N}{DB})$ I/Os.

Another important operation is *external sorting*. The proposed algorithms fall into two categories: those based on the *merging* paradigm, and those based on the *distribution* paradigm. *External merge-sort* is an extension of the internal merge-sort algorithm. The algorithm first repeatedly reads M elements in blocks of size B. These M elements are sorted in the RAM and the sorted sequence is written back to the hard disk. A merging step is then applied that merges M/B sequences into one sorted sequence. Subsequently, sequences of

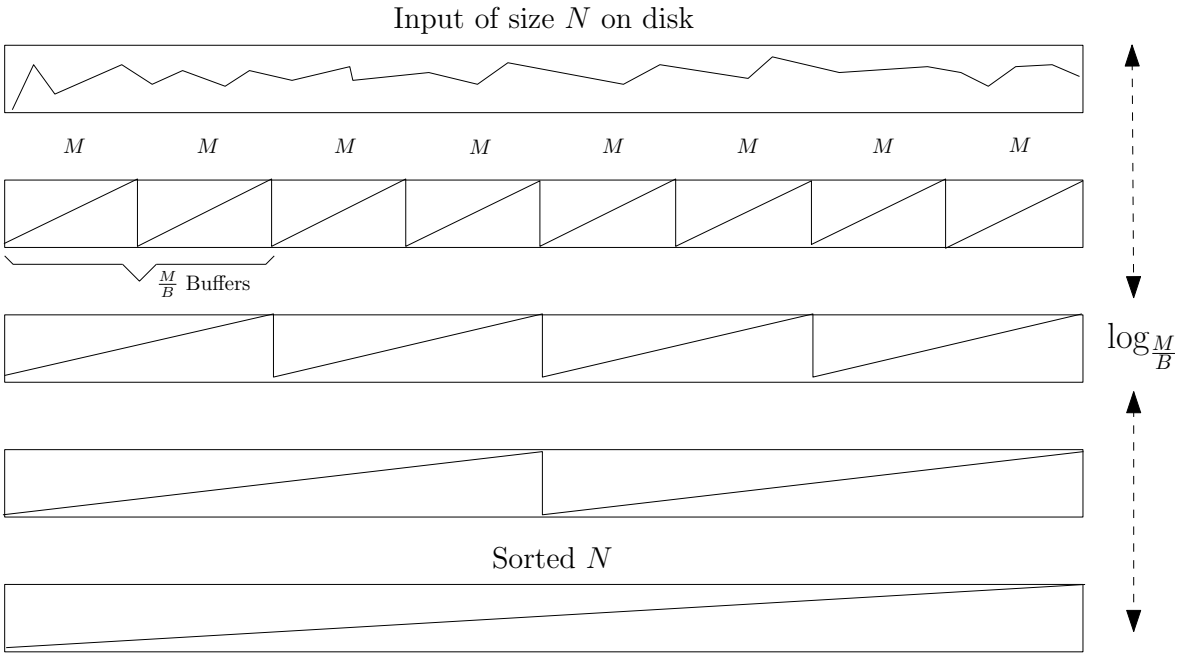

Figure 2.4: Working of External Merge-sort.

size M are merged until only one sorted sequence remains on the hard disk. A set of k sorted sequences can be merged into one run with $O(N)$ comparison operations by reading each sequence in block wise manner. In internal memory, k cursors p_k are maintained for each of the sequences; moreover, it contains one buffer block for each run, and one output buffer. Among the elements pointed to by the p_k, one with the smallest key, say p_i, is selected; the element is copied to the output buffer, and p_i is incremented. Whenever the output buffer reaches the block size B, it is written to disk, and emptied; similarly, whenever a cached block for an input sequences has been fully read, it is replaced with the next block of the run in external memory.

In Figure 2.4, we see an episode of external merge-sort with $k = 2$. Starting with an unsorted sequence, shown with zig-zag lines, B elements are selected, internally sorted and flushed back to the file. In the second phase, M/B sorted sequences are merged by iteratively reading the B elements from each of the sequences, merging them and flushing the resulting sorted sequence to the disk. The recursion continues until only one sorted sequence is left on the disk. The above operations can be I/O optimally achieved using a memory of size $M = 3B$, where $2B$ is needed for reading the two sequences and B for writing the output to the hard disk in block-wise manner.

When using one internal buffer block per sequence, and one output buffer, each merging phase uses $O(N/B)$ operations. The best result is achieved when k is chosen as large as possible, i.e., $k = M/B$. Then sorting can be accomplished in $O(\log_{M/B} \frac{N}{B})$ phases, giving a total I/O complexity of

$$O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right) \text{I/Os.}$$

On the other hand, algorithms based on the *distribution* paradigm partition the input

data into disjoint sets S_i, $1 \leq i \leq k$, such that the key of each element in S_i is smaller than that of any element in S_j, if $i < j$. In order to produce this partition, a set of *splitters* $-\infty = s_0 < s_1 < \cdots < s_k < s_{k+1} = \infty$ is chosen, and S_i is defined to be the subset of elements $x \in S$ with $s_i < x \leq s_{i+1}$. The splitting can be done I/O-efficiently by streaming the input data through an input buffer, and also using an output buffer. Then, each subset S_i is recursively sorted, unless its size allows sorting in internal memory. The final output is produced by concatenating all of the sorted subsequences. Optimality can be achieved by a good choice of splitters, i.e., such that $|S_i| = O(N/k)$. It has been proposed to calculate the splitters in linear time based on the classical internal-memory *selection* algorithm to find the k-smallest element.

These primitives, together with their complexities, are summarized in Table 2.1.

Operation	Complexity	Optimality achieved by
scan(N)	$\Theta\left(\frac{N}{DB}\right)$	Trivial sequential access
sort(N)	$\Theta\left(\frac{N}{DB}\log_{M/B}\frac{N}{B}\right)$	*Merge* or *Distribution Sort*

Table 2.1: Primitives of External Memory algorithms.

2.6 External Memory Search Algorithms for Explicit Graphs

In this section, we review some of the developments in designing External Memory algorithms for large explicit graphs that are provided beforehand. Let $\mathcal{G} = (V, E)$ be the graph with V denoting the set of nodes or vertices of the graph and E the set of edges provided in the form of an edge list. An element $e \in E$ in the edge list is a tuple (u, v) with u called the *source* of e and v the *target* of e. We are interested in finding an *optimal* path (based on the number of edges) from a given set of *initial states* $\mathcal{I}$ to one of the *goal states* $\mathcal{T}$.

We first study a basic Breadth-First Search algorithm defined on the EM model and discuss a recently proposed improvement on it. But before we proceed to discuss the External Breadth-First Search, we would like to briefly recall the standard internal memory Breadth-First Search.

2.6.1 Internal Breadth-First Search

Internal Breadth-First Search visits each vertex $u \in V$ of the input graph $\mathcal{G} = (V, E)$ in a breadth-wise fashion. The breadth-wise order is due to a FIFO queue also referred to as the *Open* list. This queue holds the list of nodes that are still waiting to be expanded. After a node u is extracted, the edge list (the sets of neighbors in $\mathcal{G}$) is examined, and those states that haven't been visited so far are inserted into *Open* in turn. The algorithm also maintains a hash table called *Closed* list that avoids re-expansion of the nodes. In Algorithm 2.1, we see a formal description of the internal BFS algorithm. The algorithm returns the optimal path length to one of the target nodes $t \in \mathcal{T}$, in case one is found, else $+\infty$ is returned. The BFS number of each node u is accessible through $g(u)$.

For large graphs, whose adjacency lists cannot fit in the internal memory and have to reside on the hard disk, Algorithm 2.1 is not applicable in its current form. However, with some modifications to process the nodes and edges residing on the hard disk, the algorithm

Algorithm 2.1 Internal Breadth-First Search

Input: $\mathcal{I}$: The set of initial nodes.
Output: Optimal path length from $\mathcal{I}$ to a node $t \in \mathcal{T}$, if one exists.

```
1:  Open ← ∅      //OPEN LIST AS A FIFO QUEUE
2:  Closed ← ∅    //CLOSED LIST AS A HASH TABLE
3:  Open ← I
4:  i ← 1
5:  while (Open ≠ ∅) do    //WHILE THERE ARE UNEXPLORED NODES IN THE Open QUEUE
6:    u ← dequeue(Open)
7:    if u ∈ T then    //IS u ONE OF THE GOAL NODES?
8:      return i − 1
9:    Closed ← Closed ∪ {u}    //u IS EXPANDED
10:   for all v ∈ Nbrs(u) do    //ITERATE ON ALL THE NEIGHBORS OF u
11:     if v ∉ Closed ∧ v ∉ Open then
12:       g(v) ← g(u) + 1    //INCREASE THE DEPTH VALUE OF v
13:       enqueue(Open, v)    //ENQUEUE THE NEIGHBOR v IF IT IS NEVER VISITED
14:   end for
15:   i ← i + 1
16: end while
17: return ∞
```

can be made to work on large graphs. Assume that the whole set of adjacency lists is saved on the hard disk, and since, we cannot afford a hash table in the internal memory, we use two FIFO files: one for the *Open* set and the other for the *Closed* set.

In Algorithm 2.1, the two places where we need to access the hard disk in an unstructured manner are statements 10 and 11. Statement 10 would result in $\Theta(|E|)$ I/Os to search for all the neighbors, while the lookup in statement 11 needs $\Theta(|V|)$ I/Os, to find out if the neighboring nodes have already been visited or not. For large graphs, with millions of nodes, this naïve extension is almost intractable.

2.6.2 External Breadth-First Search

One of the key breakthroughs in graph traversal algorithms was the External Breadth-First Search algorithm proposed by Munagala & Ranade (1999) that we refer to as **MR-BFS**. The input is usually assumed to be an unsorted edge list stored contiguously on disk. However, frequently, algorithms assume an *adjacency list representation*, which consists of two files on disk: one which contains all edges sorted by the start node, and one file of size $|V|$ which stores, for each vertex, its out-degree and offset into the first file. A preprocessing step can accomplish this conversion in $O\left(\frac{|E|}{|V|} sort(|V|)\right)$ I/Os. Note that for undirected graphs, for each edge $(u, v) \in E$, we have to insert its dual edge (v, u) in the adjacency list too.

The algorithm proceeds in three major steps when it builds the open list at level i, $Open(i)$ from the open list of the previous breadth-first level $Open(i-1)$. To keep the notations simple, we do not distinguish b/w *Open* and *Closed* list and use the term *Open* to refer to both. Whether the list is closed or still open can be inferred from the flow of the algorithm. The pseudo-code of the algorithm is shown in Algorithm 2.2, while Figure 2.5 shows an example graph on the right and a run of the algorithm on the left.

Algorithm 2.2 External Breadth-First Search – **MR-BFS**

Input: $\mathcal{I}$: The set of initial nodes
Output: Optimal path length from $\mathcal{I}$ to a node $t \in \mathcal{T}$, if one exists.

```
1:  Open(−1) ← Open(−2) ← ∅
2:  Open(0) ← I
3:  i ← 1
4:  while (Open(i − 1) ≠ ∅ ) do      //REPEAT UNTIL THERE ARE NO NODES LEFT IN LAYER i − 1
5:    if T ∩ Open(i − 1) ≠ ∅ then      //IF ONE OF THE GOALS IS REACHED
6:      return i − 1
7:    else
8:      for all u ∈ Open(i − 1) do      //EXPAND EACH NODE u
9:        Nbrs ← Read adjacency list of u
10:       A(i) ← A(i) ∪ Nbrs
11:     end for
12:     A′(i) ← sort-and-remove-duplicates(A(i))      //SORT AND COMPACT THE FILE A(i)
13:     Open(i) ← A′(i) \ (Open(i − 1) ∪ Open(i − 2))      //SUBTRACT PREVIOUS TWO LAYERS
14:   i ← i + 1      //EXPAND NEXT LAYER
15: end while
16: return ∞
```

1. Expansion Let $A(i)$ be the multi-set of neighbor vertices of nodes in *Open*$(i-1)$; $A(i)$ is created by reading and concatenating all adjacency lists of nodes in *Open*$(i-1)$ (see line 8 in Algorithm 2.2). The resulting list is represented internally by a small buffer. When full, the buffer is flushed to a file on the disk.

2. External sorting and compaction Duplicate elimination is achieved by sorting the list of neighbors $A(i)$ through an external sorting algorithm. Due to sorting, all similar vertices would come next to each other and a linear external scan is enough to generate a duplicate free copy $A'(i)$ [Line 12].

3. Subtraction Subtraction involves removing the nodes already expanded in the previous layers. The authors suggested that for undirected graphs, it is sufficient to remove the duplicates from just the previous two layers to guarantee that no node will be expanded twice. Since the list $A'(i)$ is sorted (due to the previous step), filtering out the nodes already contained in the sorted lists *Open*$(i-1)$ or *Open*$(i-2)$ is possible by parallel scanning of the three lists [Line 13].

Note that the algorithm works on a set of initial nodes. Such can be the case when the given graph is not connected and when there are several root nodes from which the search can be started.

I/O Complexity Analysis: Before we proceed to discuss the I/O complexity of the algorithm, we would like to prove that removing duplicates corresponding to two previous layers is enough to guarantee that no node is expanded twice. The following lemma proves the correctness of the **MR-BFS** for undirected, unweighted graphs*.

*The proof is by to (Katriel & Meyer 2002) that we reproduce here for its elegance.

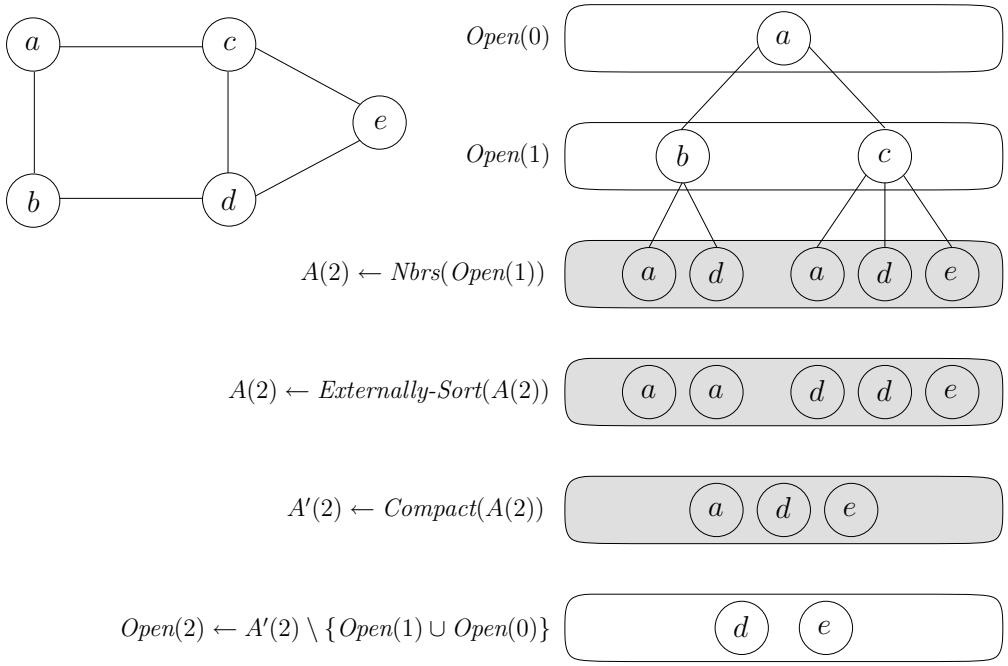

Figure 2.5: An example graph (left). Stages of MR-BFS in exploring the graph (right).

Lemma 2.1 (Correctness of MR-BFS) *In an undirected, unweighted graph, given that two previous layers are subtracted, the algorithm **MR-BFS** expands each reachable node exactly once and the BFS distances for each node are correctly computed.*

Proof We will prove it by contradiction. Assume that the layer *Open*(i) has to be expanded and till layer *Open*(i), no node has been expanded twice, i.e., the BFS numbers of each node from *Open*(0) till *Open*(i) are correct. Consider a node $u \in$ *Open*(i) with v being a neighbor of u that is now generated in layer *Open*$(i+1)$. Since the graph is undirected, the node v can only appear in layer *Open*(i), *Open*$(i-1)$, or in *Open*$(i+1)$. If v has appeared in any layer *Open*(j) with $j < i-1$, then the node u should have been appeared earlier too – a *contradiction* to our initial assumption that the BFS number of u is correct. ■

The following theorem reports the I/O complexity of the algorithm.

Theorem 2.2 (I/O Complexity of MR-BFS) *In an explicit, undirected and unweighted graph the reachability analysis problem can be solved by **MR-BFS** in*

$$O\left(|V| + sort(|V| + |E|)\right) \quad I/Os. \tag{2.1}$$

Proof The correctness of the algorithm follows from Lemma 2.1. We perform the I/O complexity analysis on individual BFS-level i for each step:

1. *Expansion:* Since after the preprocessing step the graph is stored in adjacency-list representation, it would take $O(|Open(i-1)|)$ for scanning the adjacency lists of each vertex in *Open*(i), while flushing the neighbors to the disk can be carried out in

$$O\left(scan(|Nbrs(Open(i-1))|)\right) \quad \text{I/Os}. \tag{2.2}$$

2. *External Sorting and compaction:* The I/O complexity of this step is clearly the time required to sort all the successors of layer i:

$$O\left(sort(|Nbrs(Open(i-1))|)\right) \text{ I/Os.} \tag{2.3}$$

3. *Subtraction:* Since both previous layers are already sorted, the subtraction can be done by a parallel scan of all 3 layers in

$$O\left(scan(|Nbrs(Open(i-1))|) + scan(|Open(i-1)| + |Open(i-2)|)\right) \text{ I/Os.} \tag{2.4}$$

Since $\sum_i |Nbrs(Open(i))| = O(|E|)$ (one node for each edge), and $\sum_i |Open(i)| = O(|V|)$, the expansion phase requires $O(|V|+scan(|E|))$ I/Os for reading the adjacency lists and flushing the concatenated list to the disk. The second phase needs a total of $O(sort(|E|) + scan(|E|))$ I/Os for sorting and compacting the list of neighbors. Since the layers are now sorted, the subtraction phase can be carried out by a parallel scan that takes at most $O(scan(|E|) + scan(|V|)) = O(scan(|E|)$ I/Os. Summing the three we get the overall I/O complexity of

$$O\left(|V| + sort(|V| + |E|)\right) \text{ I/Os.}$$

■

The algorithm can record the nodes' BFS-level in additional $O(|V|)$ I/Os using an external array.

2.6.3 Improvements on External Breadth-First Search

The dominating factor in the I/O complexity of **MR-BFS** for sparse graphs is '$|V|$', which is due to the unstructured accesses to the adjacency lists. Mehlhorn & Meyer (2002) succeeded in reducing this factor by using a more structured access to the adjacency list. We will refer to this new algorithm as **MM-BFS**. The algorithm proceeds in two phases. First, it divides the set of nodes into disjoint connected subgraphs $\mathcal{S}_i$ and consecutively writes their adjacency lists to a file. This phase brings regularity to the adjacency list accesses. In the second phase, a BFS algorithm similar to **MR-BFS** is used to traverse the graph. The subgraphs are *grown* from a set of nodes called *master nodes* that are selected at random with probability $\mu = min\left\{1, \sqrt{\frac{|V|+|E|}{|V|\cdot B}}\right\}$ with the exception that the initial node $\mathcal{I}$ will be the *master node* of partition $\mathcal{S}_0$. This gives the expected number of master nodes as $1 + \mu|V|$. These partitions are grown in parallel. For every partition, we maintain the adjacency lists of the boundary nodes as *active*. In a pass through these active lists, we append them to a file and construct a sorted list of requests for the neighboring nodes. The corresponding lists are then read in one single pass from the file. The process continues until every node has been added to one of the partitions.

The key lemma in the complexity proof of this phase is that a node v will belong to a partition after at most $1/\mu$ expected rounds. Since adjacency lists are read in a more structured way with several of them read in just one single pass through the file, the total amount of data read will be bounded by

$$O\left(\sum_{v\in V} \frac{1}{\mu}(1 + degree(v))\right) = O\left(\frac{|V| + |E|}{\mu}\right)$$

giving us an I/O complexity bound of

$$O\left(\frac{|V|+|E|}{(\mu \cdot B)} + \mathit{sort}(|V|+|E|)\right)$$

expected I/Os.

The second phase proceeds just like **MR-BFS** with the exception of a file $\mathcal{H}$ called *Hot adjacency lists* containing the adjacency lists of all the nodes at the level $i-1$. One single pass of $\mathcal{H}$ is then enough to generate the level i, as opposed to the unstructured accesses in **MR-BFS**. The price for this single pass is paid in the maintenance of $\mathcal{H}$ by keeping it sorted and up to date with the level $i-1$. The result is an I/O complexity of

$$O\left(\mu \cdot |V| + \frac{|V|+|E|}{\mu \cdot B} + \mathit{sort}(|V|+|E|)\right)$$

expected I/Os. Summing the two phases, we get

$$O\left(\sqrt{|V| \cdot \mathit{scan}(|V|+|E|)} + \mathit{sort}(|V|+|E|)\right)$$

expected I/Os for **MM-BFS**, which is a clear-cut improvement on the I/O complexity of **MR-BFS**.

Empirical Comparison of MR-BFS and MM-BFS

Ajwani, Dementiev, & Meyer (2006) report an extensive comparison of **MR-BFS** and **MM-BFS** on a variety of randomly generated graphs. They have found **MM-BFS** to be superior for graphs with large diameters while **MR-BFS** performed better on graphs with small diameters. More recently, Ajwani, Meyer, & Osipov (2007) presented a deterministic variant of **MM-BFS**, that we denote here as **MM-BFS-D**. They also suggested improvements on the implementation of these BFS algorithms. **MR-BFS** was improved by fine tuning of the sorting module, that now avoids the external sort if the internal sort is sufficient. Both **MM-BFS** and **MM-BFS-D** have been improved by caching the adjacency lists in a hash table. With these improvements, the running times of the algorithms have been reduced several folds. The deterministic version has been found to be superior to the randomized version in those examples, where earlier randomized variant outperformed **MM-BFS**.

2.7 External Memory Search Algorithms for Implicit Graphs

Until now, we have only looked at the graphs that are provided in the form of adjacency lists, beforehand. Such graphs are commonly known as *explicit* graphs. In contrast, an *implicit graph* is a graph that is not available beforehand, but is generated by successively applying a set of operators or rules to a given initial state. One of the most common examples is a graph generated during game playing whether single or two player games, where every single vertex is obtained by applying a *move* by one of the player. The advantage in implicit graph search is that since the graph is generated *on-the-fly*, no disk accesses for the adjacency lists are required. Implicit graphs are sometimes also referred to as *state spaces* as a vertex corresponds to *state* generated by the application of a sequence of *transitions*. To distinguish

implicit graphs from the explicit ones, we denote the sets of states and transitions with $\mathcal{S}$ and $\mathcal{R}$, respectively.

One of the first investigations that uses disk space for traversal in implicit graphs is (Korf 2003) that is based on the idea of *Frontier Search* (see (Korf 2005) for a detailed analysis of Frontier Search). The name *Frontier Search* comes from the nature of the algorithm as it keeps only the states that lie at the *frontier* of the search space. Throwing away the *Closed* set completely and saving only the frontier can free a significant amount of memory. Since the primary usage of *Closed* set is to avoid expansion of already expanded states, the algorithm avoids re-expansions by storing with each state, a bit vector representing the legal operators that can be applied on the state. Let v be the successor of the state u. When u is expanded, the operator in v's vector that can generate u is marked and not used during v's future expansion. If v is already found in *Open* set, the operator vector is updated with the union of the new and old copies of v. The algorithm also maintains a *cost* parameter with every state that is updated to the minimum of the two copies (old v and the new v). For undirected graphs, this technique of *used operators* is enough to avoid the search *leaking back* into the *Closed* set.

For directed graphs, avoiding the *leakage* becomes a bit more complicated. The algorithm now requires that each state has access to all of its predecessors. When expanding u, we also generate all the predecessors of u and insert them in the *Open* list as "dummy" nodes, set their cost to infinity and mark the operator generating u as used. These nodes will not be expanded until they are reached on a legal path (that would decrease the cost). Once they are expanded, u will not be generated because of the marked operator.

Unfortunately, for complex directed graphs, such as those that appear in model checking, Frontier Search is not very useful due to two major reasons. First, the predecessor of a state is not always available, or is very difficult to identify. Second, even if we have access to predecessors, high branching factors may lead to the generation of a large number of "dummy" nodes.

The approach of Frontier Search was extended for External Breadth-First Search in implicit, undirected, and unweighted large graphs. The approach was named *Delayed Duplicate Detection for Frontier Search* (Korf 2003). It is basically a variant of Munagala and Ranade's **MR-BFS** algorithm for implicit graphs. We will refer to this External BFS for implicit graphs as **DDD**. The algorithm maintains BFS layers on disk. Let $\mathcal{I}$ be the set of initial states, and *Succ* the successor generation function. Layer *Open*$(i-1)$ is scanned and the set of successors are put into a buffer of a size close to the main memory capacity. If the buffer becomes full, internal sorting followed by a duplicate elimination phase generates a sorted duplicate-free state sequence in the buffer that is flushed to disk.

In the next step, *external merging* is applied to unify the files into *Succ*(i) by a simultaneous scan containing all the neighbors of *Open*$(i-1)$. Duplicates are eliminated while unifying the files through external sorting. One also has to eliminate *Open*$(i-1)$ and *Open*$(i-2)$ from *Open*(i) to avoid re-expansions of the states. The process is repeated until *Open*$(i-1)$ becomes empty, or the goal has been found. The corresponding pseudo-code is shown in Algorithm 2.3. Since the graph is unweighted, the optimal cost for reaching a goal is same as its depth, and the algorithm can be terminated as soon as a goal node is generated.

I/O Complexity Analysis: Korf (2003) does not analyze the I/O complexity of the **DDD** algorithm. In the following, we report an I/O complexity analysis of the algorithm using

similar techniques as developed for **MR-BFS**.

Theorem 2.3 (I/O Complexity of DDD) *In an implicit, undirected and unweighted graph the reachability analysis problem can be solved by **DDD** in*

$$O\left(scan(|\mathcal{S}|) + sort(|\mathcal{R}|)\right) \quad I/Os. \tag{2.5}$$

Proof Removing two previous layers from the successor layer prevents the re-expansion of nodes similar to **MR-BFS** (c.f. Lemma 2.1). For I/O complexity, as with the **MR-BFS**, we distinguish three sub-complexities of the algorithm during a single iteration i.

- *Expansion:* As we just have to scan through the previous layer, it is clearly $scan(|Open(i-1)|)$ for scanning the layer $i-1$ and $scan(|Succ(Open(i-1))|)$ I/Os for sequentially saving the successors to the disk. Note the difference to **MR-BFS** algorithm because of the lack of adjacency lists.
- *External Sorting:* $O(sort(|Succ(Open(i-1))|))$ I/Os for external sorting and removing duplicates from the list of neighbors.
- *Subtraction:* Externally scanning previous two layers simultaneously with $Succ(Open(i-1))$ amounts to $O(scan(|Succ(Open(i-1))| + |Open(i-1)| + |Open(i-2)|))$ I/Os.

Summing all *Open* lists gives us $\sum_i |Open(i)| = O(\mathcal{S})$. Since each state will be re-generated as many times as there are incoming edges, we have

$$\sum_i |Succ(Open(i))| = O(|\mathcal{R}|).$$

Using these upper bounds, we can say that **DDD** needs $O(scan(|\mathcal{S}|) + scan(|\mathcal{R}|))$ I/Os for scanning the layer for expansion and for saving the generated neighbors back through a buffered file. Sorting and subtraction take at most $O(sort(|\mathcal{R}|) + scan(|\mathcal{R}|))$ I/Os. Summing them up, we arrive at a total I/O complexity of **DDD** as

$$O\left(scan(|\mathcal{S}|) + sort(|\mathcal{R}|)\right) \quad \text{I/Os.}$$ ■

In exploration problems where the branching factor is bounded, we have $|\mathcal{R}| = O(|\mathcal{S}|)$, and thus the complexity for implicit external BFS reduces to $O(sort(|\mathcal{S}|))$ I/Os.

The algorithm applies $scan(|Open(i-1)| + |Open(i-2)|)$ I/Os in each phase. Does summing these quantities in fact yield $O(scan(|\mathcal{S}|))$ I/Os, as stated? In very sparse problem graphs that are simple chains, if we keep each $Open(i)$ in a separate file, this would accumulate to $O(|\mathcal{S}|)$ I/Os in total. However, in this case the states in $Open(i)$, $Open(i+1)$, and so forth are stored consecutively in internal memory. Therefore, I/O is only needed if a level has $\Omega(B)$ states, which can happen only for $O(|\mathcal{S}|/B)$ levels.

2.8 Summary

This chapter aims to give an introduction to External Memory algorithms. We discussed the problems with the traditional von Neumann model in dealing with massive data and the lack of analysis techniques. This problem has been circumvented by the External Memory model.

Algorithm 2.3 Delayed Duplicate Detection in Frontier Search – **DDD**

Input: $\mathcal{I}$: The initial state
Output: Optimal path length from $\mathcal{I}$ to a node $t \in \mathcal{T}$, if one exists.

1: $Open(-1) \leftarrow Open(-2) \leftarrow \emptyset$
2: $Open(0) \leftarrow \mathcal{I}$
3: $i \leftarrow 1$
4: **while** $(Open(i-1) \neq \emptyset)$ **do** //REPEAT UNTIL THERE ARE NO STATES LEFT IN LAYER $i-1$
5: **if** $\mathcal{T} \cap Open(i-1) \neq \emptyset$ **then** //IF ONE OF THE GOALS IS REACHED
6: **return** $i-1$
7: $A(i) \leftarrow Succ(Open(i-1))$ //GENERATE SUCCESSORS
8: $A'(i) \leftarrow$ *sort-and-remove-duplicates*$(A(i))$ //SORT AND COMPACT THE FILE $A(i)$
9: $Open(i) \leftarrow A'(i) \setminus (Open(i-1) \cup Open(i-2))$ //SUBTRACT PREVIOUS TWO LAYERS
10: $i \leftarrow i+1$ //EXPAND NEXT LAYER
11: **end while**
12: **return** ∞

Perhaps, the best way to understand the workings of External Memory algorithms is to study the Breadth-First Search in large graphs. Breadth-First Search has many useful applications. One important application includes the web-crawling by search bots to collect web-graph structure that can facilitate the searching process.

A web-graph is a typical example of an *explicit graph*. However, for many real-world problems as they appear in model checking or game-playing, the graphs are not available beforehand. Such graphs, usually referred to as *implicit graphs*, are generated by a successive application of rules on a given set of initial states. We presented an overview of *Frontier Search* that uses delayed duplicate detection to perform an external Breadth-First Search in implicit graphs.

An efficient implementation of External Memory algorithms is a difficult task. TPIE (Arge, Procopiuc, & Vitter 2002) and STXXL (Dementiev 2006) are two freely available C++ libraries that provide implementations of several External Memory algorithms. TPIE (Templated Parallel I/O Environment) concentrates more on the computational geometry algorithms and provide special support for spatial data structures (Arge *et al.* 1999). STXXL, on the other hand, is designed with graph based algorithms in mind.

Our point of focus in this thesis will be different search strategies for various types of implicit graphs, and several I/O complexity results will be based upon the techniques presented in this chapter.

Chapter 3

Heuristic Search

In this chapter, we propose the use of external memory for guided search in implicit state spaces. Earlier, we have seen some Breadth-First Search algorithms defined on memory hierarchy. Breadth-First Search and Depth-First Search search are two trivial algorithms for reachability in a state space. Depth-First Search utilizes a *stack* to insert the newly generated successors of a state. The traversal policy thus induced prefers the states with a higher depth g value for expansion to the states with a lower g value. Breadth-First Search is analogous with the stack replaced by a *queue*, and hence, prefers states with a lower g-value to the ones with a higher g-value.

Both these algorithms fall into the category of *blind* search algorithms, i.e., they do not utilize any information about the problem being solved to guide the search. Algorithms that exploit such information are called *heuristic* search algorithms (Pearl 1985). They differ from the blind search algorithms like Breadth-First Search or Depth-First Search in the way they *order* the search space during the exploration. Unlike blind search algorithms, heuristic search algorithms utilize some guidance either provided by the user, or inferred from the problem, to hone in to the goal. A typical example being a route-planning system that prioritizes the nodes that minimize the Euclidean distance (air distance) to the destination, for expansion. Having such a guidance or *heuristic* can drastically reduce the total number of states considered until the goal is reached.

In this chapter, we present *External A** (Edelkamp, Jabbar, & Schrödl 2004), an External Memory variant of the heuristic search algorithm A*. Our presentation is restricted to unweighted and undirected state spaces only; the extensions to weighted and directed state spaces will be presented in the subsequent chapters. Unlike Breadth-First Search or Depth-First Search, heuristic search algorithms require a priority queue to order the search. For large graphs, an external priority queue is hence needed. External A*, instead, utilizes the properties of heuristic estimates to induce an implicit priority queue.

Structure: We first study heuristic search algorithms for internal memory and discuss the requirements on the heuristic estimates that characterize the found solution. We then present one important data structure called Bucket that is a building block of External A*. The expansion order is presented next along with a detailed I/O complexity analysis of the algorithm. To empirically evaluate the performance of External A*, we report results on solving different random instances of the single-agent game 15-Puzzle that has $16!/2 \approx 10^{13}$ reachable

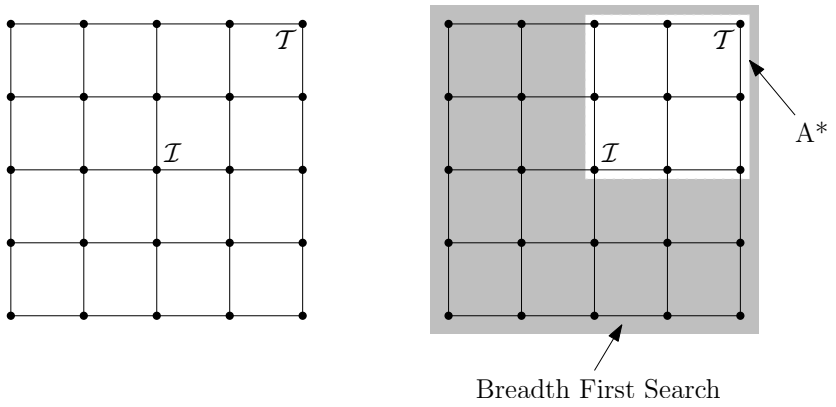

Figure 3.1: Comparison of states expanded by A* and BFS on a grid graph. States enclosed in the white square are expanded by A*, while expanded nodes by BFS enclosed in the shaded square.

states.

3.1 Heuristic Search with A*

Heuristic search algorithms constitute the set of algorithms that exploit some form of guidance or heuristics to guide the search into the direction of the goal state. The guidance or the *heuristic* can either be inferred from the problem structure itself or be supplied by the user.

A* (*pronounced 'A star'*) (Hart, Nilsson, & Raphael 1968) is a heuristic search algorithm. It is sometimes also referred to as *Goal-directed Dijkstra* (Wagner & Willhalm 2003) because of the similarities it shares with Dijkstra's single-source shortest path algorithm. Dijkstra's algorithm (Dijkstra 1959; Mehlhorn 1984) traverses a graph by prioritizing the nodes that have the smallest shortest path distance δ from the start state. It maintains a list of candidate states waiting to be expanded as an ordered priority queue with the top element being the one with the minimum δ value. Iteratively, the top element u is removed and expanded. For every successor node v, Dijkstra's algorithm checks if $\delta(\mathcal{I}, v) + c(u, v) < \delta(\mathcal{I}, v)$, where $c(u, v)$ is the weight of the edge (u, v). In such a case, we have found a better path to v via u. For the time being, we assume that there is just one start state, i.e., $|\mathcal{I}| = 1$.

A* works on a similar principle, except that it not only prioritizes the nodes with respect to the shortest distance value δ, but also the heuristic estimate to the goal. Formally, a heuristic estimate is a function $h : \mathcal{S} \to \mathbb{R}$ that assigns, for each state $u \in \mathcal{S}$, its estimated distance to the goal. The priority of a state u is denoted by $f(u)$ and is computed as $f(u) = g(u) + h(u)$, where $g(u) = \delta(\mathcal{I}, u)$.

A simple illustration of the workings of A* is shown in Figure 3.1. On a grid, the number of nodes visited by a Breadth-First Search and by A* are highlighted. The initial state is in the center of the grid and the target at the top right corner. Breadth-First Search expanded almost all the states before arriving at the target. On the contrary, A* only expanded the states that are *promising* in reaching to the target.

Algorithm 3.1 depicts the pseudo-code of A*. It utilizes two state sets: *Open* containing the states waiting to be expanded, and *Closed* containing the expanded states. In each iteration, A* removes the state with the lowest f-value from the priority queue *Open* [Line 6]

for expansion. Before the expansion, u is first checked for inclusion in the set of target states $\mathcal{T}$ [Line 7]. If $u \in \mathcal{T}$, the path from $\mathcal{I}$ to u is constructed by back-tracking on the predecessor pointers and the algorithm is terminated. If the desired path is not established, u is expanded by applying the applicable operators/transformation rules on u. This process is represented by the successor generation function *Succ*. For every successor v of u, either of the three cases can be true:

- Case I [Line 11]: v has never been visited before, i.e., it does not exist in either the *Open* or the *Closed* sets. In such a case, v is simply inserted into the priority queue with the priority $f(v) = g(u) + c(u, v) + h(v)$.
- Case II [Line 15]: v was waiting to be expanded, i.e., it exists in the *Open* queue. Instead, a better path to v is found via u. This case is dealt with by decreasing the priority of v in the priority queue *Open*.
- Case III [Line 19]: v has already been expanded and now exists in the *Closed* list. Instead, a better path to v is found via u. In this case, it is removed from the *Closed* list and inserted in the *Open* queue with the priority $f(v) = g(u) + c(u, v) + h(v)$.

A* continues expanding the states until either one of the target states is chosen for expansion, or the priority queue is empty.

3.1.1 Admissibility and Consistency

If the heuristic estimates are *admissible* or *consistent*, one can argue about the complexity of the search and the quality of the obtained solution.

Definition 3.1 (Admissible Heuristic) *A heuristic is admissible, if it never over-estimates the actual distance to the goal state and if the heuristic estimates of the goal states are zero. Formally, let $h : \mathcal{S} \rightarrow \mathbb{R}$ be the heuristic function, then h is admissible, if and only if, for all $u \in \mathcal{S}$, we have*

$$h(u) \leq \min\{\delta(u, t) | t \in \mathcal{T}\},$$

and for all $t \in \mathcal{T}$, $h(t) = 0$.

Note that admissibility does not imply that states that have a heuristic value of zero should be classified as target states. In practice, however, most of the heuristics assign a zero estimate to only the target states.

For A* to terminate with an optimal path, the admissibility condition is sufficient.

Theorem 3.1 (Correctness of A*) *Given that the heuristic function h is admissible, A* terminates with the optimal solution (Hart, Nilsson, & Raphael 1968).*

Definition 3.2 (Consistent/Monotone Heuristic) *A heuristic is consistent or monotone, if for each state u and its successor v, we have*

$$c(u, v) \geq h(u) - h(v) \ .$$

Algorithm 3.1 A* Algorithm

Input: $\mathcal{I}$: The initial state

Output: A path from $\mathcal{I}$ to the closest state $t \in \mathcal{T}$ along with $\delta(\mathcal{I}, \mathcal{T})$, if one exists.

1: *Open* list as a priority queue on f-values (lower is better)
2: $g(\mathcal{I}) \leftarrow 0$ //THE DISTANCE OF INITIAL STATE TO ITSELF IS 0
3: $f(\mathcal{I}) \leftarrow g(\mathcal{I}) + h(\mathcal{I})$ //CALCULATE THE f-VALUE OF THE INITIAL STATE FROM ITS g- AND h-VALUES
4: insert(*Open*, $\mathcal{I}$, $f(\mathcal{I})$) //INSERT THE INITIAL STATE IN THE PRIORITY QUEUE *Open* WITH PRIORITY $f(v)$
5: **while** *Open* $\neq \emptyset$ **do** //WHILE THERE ARE STATES IN THE *Open* QUEUE
6: $u \leftarrow$ deleteMin(*Open*) //GET THE STATE WITH THE MINIMUM f-VALUE
7: **if** $u \in \mathcal{T}$ **then** //GOAL REACHED?
8: **return** *Construct-Solution*(u) //RETURN THE PATH FOUND FROM $\mathcal{I}$ TO u
9: *Closed* $\leftarrow$ *Closed* $\cup \{u\}$
10: **for all** $v \in$ *Succ*(u) **do** //ITERATE ON ALL THE SUCCESSORS v
11: **if** $v \notin$ *Open* **and** $v \notin$ *Closed* **then** //IF v IS NEVER VISITED BEFORE
12: $g(v) \leftarrow g(u) + c(u, v)$ //v IS $c(u, v)$ DISTANCE AWAY FROM u
13: $f(v) \leftarrow g(v) + h(v)$
14: insert(*Open*, v, $f(v)$) //INSERT v IN *Open* WITH PRIORITY $f(v)$
15: **else if** $v \in$ *Open* **and** $g(v) > g(u) + c(u, v)$ **then** //HAS THE PATH TO v BECOME BETTER?
16: $g(v) \leftarrow g(u) + c(u, v)$
17: $f(v) \leftarrow g(v) + h(v)$
18: decreaseKey(*Open*, v, $f(v)$) //DECREASE THE PRIORITY OF v TO THE NEW $f(v)$
19: **else if** $v \in$ *Closed* **and** $g(v) > g(u) + c(u, v)$ **then** //v HAS BEEN EXPANDED EARLIER BUT NOW THERE IS A BETTER PATH TO v
20: $g(v) \leftarrow g(u) + c(u, v)$
21: $f(v) \leftarrow g(v) + h(v)$
22: remove(*Closed*, v)
23: insert(*Open*, v, $f(v)$) //v SHOULD BE EXPANDED AGAIN
24: **end for**
25: **end while**
26: **return** $\langle \emptyset, +\infty \rangle$ //EMPTY PRIORITY QUEUE AND NO PATH FROM $\mathcal{I}$ TO $\mathcal{T}$ HAS BEEN FOUND!

Consistency guarantees that the drop in the heuristic value from a state to its successor should not be greater than the weight of the transition. It is a very important property as it prevents the re-openings of the states. If the heuristic is consistent, then along each search path, the evaluation function f is non-decreasing. No successor will have a smaller f-value than the current one. Therefore, the A* algorithm, which traverses the state set in f-order, expands each node at most once, i.e., Case III will never happen.

Theorem 3.2 (Optimal Efficiency of A*) *Let h be a consistent heuristic estimate and A be a heuristic search algorithm, such that A is no more informed than A*. Then, any node expanded by A* is also expanded by A (Hart, Nilsson, & Raphael 1968).*

Theorem 3.2 states that given consistent heuristics, A* expands the minimal number of nodes. Another important property of a consistent heuristic is that it is also admissible, i.e., consistency implies admissibility – the converse is not necessarily true.

Many heuristics that frequently appear in practice are admissible and consistent. Examples include Euclidean distances in solving shortest-path problems in graphs with a geometric layout, Manhattan distance in solving puzzles, etc. To acquire good, consistent heuristic estimates, *pattern databases* (PDBs) (Culberson & Schaeffer 1998) have received significant attention from both AI and model checking communities. In PDBs, a path-preserving abstract graph of the original graph is constructed by ignoring some features of the state vector. The ignored features are replaced by *don't-cares* that result in only few unique abstract states. A Breadth-First traversal starting from the target states and going backwards allows one to save the exact distance of each abstract state to the abstract target. During the exploration in the concrete graph, these distances can then be used as heuristic estimates. These heuristics are both admissible and consistent. Multiple PDBs can also be combined to form better estimates (Korf & Felner 2002).

3.1.2 Complexity

The data structure that provides fast inclusion test for *Open* and *Closed* sets is a hash table. Typically, a single hash table is used along with a separate priority queue where only a pointer to the state in the hash table is saved. Whether a state is already expanded or is still waiting to be expanded is characterized by the use of a boolean flag.

Since A* only changes the order of expansion of Dijkstra's algorithm, its worst-case complexity is the same as for Dijkstra's, i.e., $O(|\mathcal{R}|+|\mathcal{S}|\cdot\log(|\mathcal{S}|))$, when used with Fibonacci heaps (Tarjan & Fredman 1987). However, for graphs where the range of f-values is bounded, there is a more efficient data structure than a Fibonacci heap, namely a Dial's priority queue (Dial 1969). It assumes a graph where the f-values lie in a small integer range. An illustration of the data structure is presented in Figure 3.2. Pointers to the states with the same f-values are saved at the position f in an array. In case of more than one state with the same f-value, a linked list is formed. The actual state description, though, is saved in the hash table for inclusion checks. The priority queue works by keeping track of the current index being used. Once the index is exhausted, i.e., all the states in the linked list are expanded, the next non-empty index is searched and next state with a higher f-value is returned. The *insert*, *remove* and *decreaseKey* operations take constant $O(1)$ time. Similarly *deleteMin* takes a constant time in the best case. In the worst case, when there is a gap of unused indices, it has to iterate over the whole array to fetch the next state.

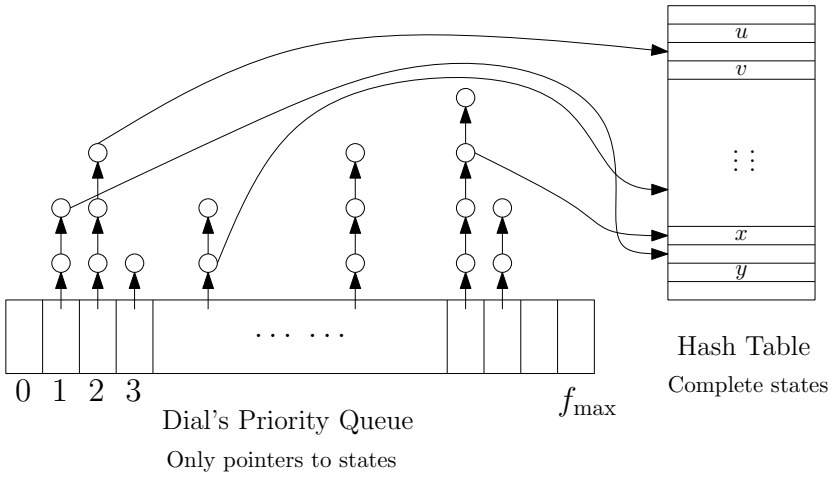

Figure 3.2: Dial's priority queue with a hash table.

For a detailed treatment of the applicability of A* on graphs whose cost functions belong to different algebraic groups, we refer the interested reader to (Edelkamp, Jabbar, & Lluch Lafuente 2005).

3.2 External Heuristic Search

A* is well-suited for problems that can fit into the main memory. As we have seen, the complexity and applicability of A* is very much dependent on the internal memory hash table that can contain all the visited states. Merely relying on Virtual Memory can result in excessive page faults due to random accesses in the hash table, which in turn can significantly slow down the performance. Discarding the hash table is not an option either – it would lead to an exponential number of state re-expansions.

In the following, we combine heuristic search with External Memory search. We assume an undirected and unweighted state space; extensions to weighted and directed state spaces are presented in Chapter 5. We also assume that the heuristic function assigns non-negative integers to the states, i.e., $h : \mathcal{S} \rightarrow \mathbb{Z}^*$. The new algorithm is termed as *External A**.

External A* extensively utilizes a data structure that we refer to as a *Bucket*. Before discussing the workings of the algorithm, we first take a careful look at this data structure.

3.2.1 Bucket Data Structure

A bucket is a set of states that share some common properties. We use the notion $Open(i, j)$ to represent a bucket. It contains all the states u with a path length $g(u) = i$ and heuristic estimate $h(u) = j$. Each bucket $Open(i, j)$ is saved in a separate file $File(Open(i, j))$ residing on the hard disk. As with the description of external BFS, we do not change the identifier *Open* to separate *generated* from *expanded* states (traditionally denoted as the *Closed* list). During the execution of External A*, bucket $Open(i, j)$ may refer to elements that are in the current search horizon or belong to the set of expanded nodes.

We say that a bucket is *active*, if it is either being read or written to. When active, a bucket is represented internally by a memory buffer of size B denoted as $M(Open(i, j))$ (note that

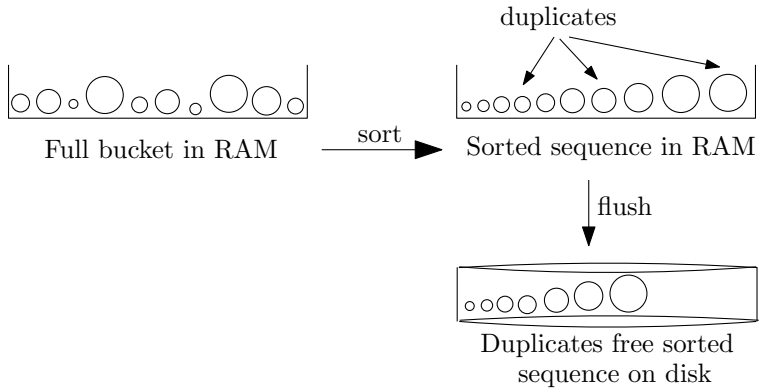

Figure 3.3: A bucket and its corresponding buffer.

B remains constant for all buckets and that it represents the number of states). States, while being accessed from a bucket, are first read in the form of a block of size B and are saved in the corresponding internal buffer. When all the states in a bucket have been processed, the bucket is emptied and a new block of size B is read from the corresponding bucket file. While writing to a bucket *Open*(i, j), states are first written in the internal memory buffer. When the buffer gets full, it is sorted and only the states that are unique wrt. the other states in the buffer are flushed to the file corresponding to *Open*(i, j). Figure 3.3 illustrates the workings of a bucket.

3.2.2 Expansion Order

External A* follows the same expansion order as A*, apart from a slight difference in the tie-breaking. A* does not distinguish between states with the same f-value. Take for example the Dial's priority queue (cf. previous section). There is no requirement on how an individual linked list should be maintained. New states can either be inserted at the back or at the front of the list. External A*, on the other hand, prioritizes the states with a lower g-value over the ones with a higher g-value among the states that share the same f-value. The result is an extra dimension in Dial's arrangement – which further explains our choice for the bucket data structure. Imagine a 2D-matrix structure (see Figure 3.4) with columns corresponding to different heuristic values and rows to different depth values. A bucket *Open*(i, j) is a cell in that matrix, containing all states with a heuristic estimate of i and a depth value of j.

Algorithm 3.2 gives the pseudo-code of *External A** for consistent and integral estimates and unweighted and undirected state spaces. We also assume that for all $u, v \in \mathcal{S}$ with $v \in \mathit{Succ}(u)$, $h(u) - h(v) \in \{-1, 0, 1\}$. This condition is often met in a number of single-agent games and model checking graphs. Let $f_{\min}$ denote the diagonal currently being expanded. Along with $g_{\min}$, the minimum g-value in the $f_{\min}$ diagonal, $g_{\min}$ and $f_{\min}$ are used to address the currently active bucket. The corresponding h-value is determined by $h_{\max} = f_{\min} - g_{\min}$. External A* is capable of dealing with more than one initial state.

All initial states are first inserted into their corresponding buckets depending on their h-value [Line 2]. The g-values of all initial states are initialized to zero. This step would result in a number of non-empty buckets in the top row of the matrix. The diagonal containing

Algorithm 3.2 External A* for Consistent and Integral Heuristics

Input: $\mathcal{I}$: the set of initial states; *Succ*: the successor generation function.
Output: A path from $\mathcal{I}$ to a state in $\mathcal{T}$, if one exists.

1: **for all** $s \in \mathcal{I}$ **do** //INSERT EACH INITIAL STATE INTO ITS CORRESPONDING BUCKET
2: $Open(0, h(s)) \leftarrow \{s\}$
3: **end for**
4: $f_{\min} \leftarrow \min\{i | Open(0, i) \neq \emptyset\}$ //SELECT THE MINIMUM $f_{\min}$ DIAGONAL TO EXPAND
5: **while** $(f_{\min} \neq \infty)$ **do**
6: $g_{\min} \leftarrow \min\{i \mid Open(i, f_{\min} - i) \neq \emptyset\}$ //SELECT THE FIRST NON-EMPTY BUCKET ON THE $f_{\min}$ DIAGONAL
7: **while** $(g_{\min} \leq f_{\min})$ **do** //LOOP FOR THE WHOLE $f_{\min}$ DIAGONAL
8: $h_{\max} \leftarrow f_{\min} - g_{\min}$
9: **if** $h_{\max} = 0$ **and** $\exists u \in Open(g_{\min}, h_{\max})$ s.t. $u \in \mathcal{T}$ **then** //HAS A GOAL BEEN REACHED?
10: **return** *ConstructPath*(u) //RETURN THE PATH FOUND FROM $\mathcal{I}$ TO u
11: $Open(g_{\min}, h_{\max}) \leftarrow$ *external-sort*$(Open(g_{\min}, h_{\max}))$ //EXTERNAL SORTING
12: $Open(g_{\min}, h_{\max}) \leftarrow$ *remove-duplicates*$(Open(g_{\min}, h_{\max}))$ //SCAN TO REMOVE ADJACENT DUPLICATES
13: $Open(g_{\min}, h_{\max}) \leftarrow Open(g_{\min}, h_{\max}) \setminus (Open(g_{\min}-1, h_{\max}) \cup Open(g_{\min}-2, h_{\max}))$ //SUBTRACT THE PREVIOUS TWO BUCKETS
14: $A(f_{\min}), A(f_{\min}+1), A(f_{\min}+2) \leftarrow Succ(Open(g_{\min}, h_{\max}))$ //GENERATE AND DISTRIBUTE THE SUCCESSORS BASED ON THEIR f VALUES
15: $Open(g_{\min}+1, h_{\max}-1) \leftarrow A(f_{\min}) \cup Open(g_{\min}+1, h_{\max}-1)$ //THERE CAN BE STATES FROM PREVIOUS TWO DIAGONALS IN THIS BUCKET
16: $Open(g_{\min}+1, h_{\max}) \leftarrow A(f_{\min}+1) \cup Open(g_{\min}+1, h_{\max})$ //THERE CAN BE STATES FROM THE PREVIOUS DIAGONAL
17: $Open(g_{\min}+1, h_{\max}+1) \leftarrow A(f_{\min}+2)$ //THIS BUCKET HAS NOT BEEN GENERATED, SO A SIMPLE ASSIGNMENT IS ENOUGH
18: $g_{\min} \leftarrow g_{\min} + 1$
19: **end while**
20: $f_{\min} \leftarrow \min\{i + j > f_{\min} \mid Open(i, j) \neq \emptyset\} \cup \{\infty\}$ //SELECT THE NEW f DIAGONAL TO EXPAND
21: **end while**
22: **return** $\emptyset$

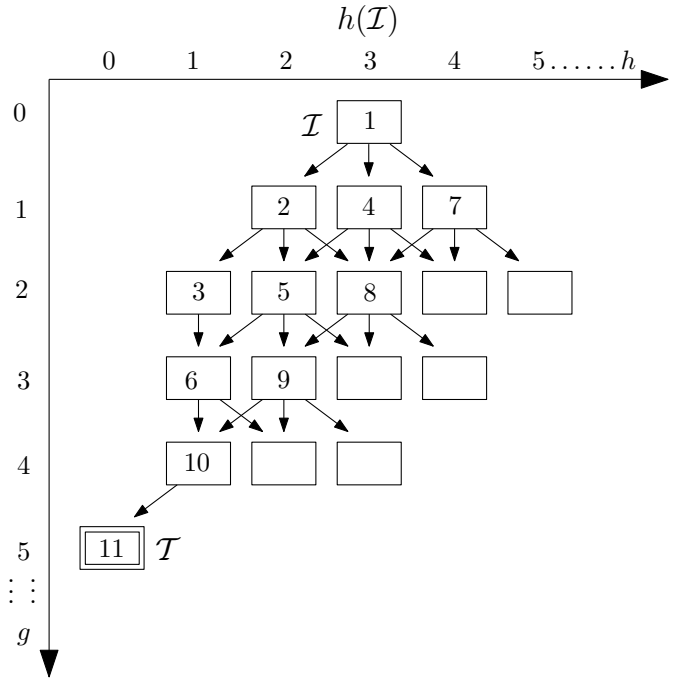

Figure 3.4: Buckets' selection in External A*. Rectangles represent buckets. Numbers in rectangles correspond to the order of expansion. Heuristic is consistent and for all $u, v \in \mathcal{S}$ with $v \in \mathit{Succ}(u)$, $|h(u) - h(v)| \in \{-1, 0, 1\}$. Unnumbered buckets are not *expanded* but *generated*.

the bucket with the minimum h-value is selected for expansion [Line 4]. The algorithm then iterates [Line 5] on the f-diagonals in increasing order, starting from the diagonal that contains the best (estimated) start state. The variable $h_{\max} = f_{\min} - g_{\min}$ is used for indexing the buckets. Within a $f_{\min}$ diagonal, buckets are selected in lexicographic order for $(i = g_{\min}, j = h_{\max})$ [Line 7]. This order results in buckets $\mathit{Open}(i', j')$ with $i' < i$ and $i' + j' = f_{\min}$ as *closed*, whereas the buckets $\mathit{Open}(i', j')$ with $i' + j' > f_{\min}$ or with $i' > i$ and $i' + j' = f_{\min}$ as *open*. For states in active buckets, the status can either be *open* or *closed*.

The selected bucket $\mathit{Open}(g_{\min}, h_{\max})$ is first scanned for the presence of any target state [Line 9]. If a target state is found within the bucket, a solution path is reconstructed and returned (details on solution reconstruction are delayed till Section 3.4). Since an admissible heuristic is used, this check needs to be performed only on the buckets that have a h-value of zero. This scan operation can easily be avoided by performing the target inclusion test at the time of expansion, instead.

The next step is to remove all the duplicate states from the bucket $\mathit{Open}(g_{\min}, h_{\max})$ to avoid any re-expansion efforts. There are two stages of duplicates removal: removal of duplicate states from the bucket, and removal of states that have already been expanded in the previous buckets. Since a bucket can be larger than the main memory, we employ an external sorting [Line 11] and compaction routine [Line 12] in the first stage. In the second phase, the previous two buckets sharing the same h-value, are subtracted from $\mathit{Open}(g_{\min}, h_{\max})$. The correctness of the duplicate's removal procedure is further discussed in Section 3.2.4.

After removing all the duplicate states, the bucket is ready for expansion. The expansion

is performed by the successor generation function *Succ*. The whole bucket is scanned and each state is individually expanded and its heuristic value is computed. At this point, we exploit our assumption that $|h(u) - h(v)| \in \{-1, 0, 1\}$ to bound the number of buckets required to carry the newly generated states. If for a successor v of u, $h(v) = h(u) - 1$, v must belong to the same diagonal $f_{\min}$. This follows from the fact that since $g(v) = g(u) + 1$, we have

$$f(v) = g(v) + h(v) = g(u) + 1 + h(u) - 1 = f(u) = f_{\min}.$$

All such successors are saved in a multi-set $A(f_{\min})$, which is managed similarly to the bucket data structure. For the other two cases, when $h(v) = h(u)$ and $h(v) = h(u) + 1$, the successors are saved in $A(f_{\min} + 1)$ and $A(f_{\min} + 2)$, respectively. Hence, at each instance only four buckets have to be accessed by I/O operations. For each of them, we keep a separate buffer of size B, thus reducing the internal memory requirements to $4B$.

The successors gathered in the temporary A files are not inserted into their respective buckets. We first consider the case for the set $A(f_{\min})$. The states in this set actually belong to the bucket *Open*$(g_{\min} + 1, h_{\max} - 1)$. It is important to observe here, that this bucket might have received successors from both $f_{\min} - 1$ and $f_{\min} - 2$ diagonals. Hence, the two multi-sets are simply concatenated* [Line 15]. Similarly, $A(f_{\min} + 1)$ belongs to *Open*$(g_{\min} + 1, h_{\max})$, which might have received states from only $f_{\min} - 1$ diagonal. Consequently, $A(f_{\min} + 1)$ is also concatenated with the previous list [Line 16]. This observation however is not true for $A(f_{\min} + 2)$ as the bucket *Open*$(g_{\min} + 1, h_{\max} + 1)$ could not have received any state from previous diagonals. A simple assignment to the bucket suffices here [Line 17]. The inner *while* loop then continues working on the next bucket on $f_{\min}$ diagonal. Figure 3.4 depicts the workings of External A*. The numbers in the buckets represent the order of expansion.

3.2.3 Number of Expanded Buckets

For an optimal heuristic, i.e., a heuristic that computes the shortest path distance, say f^*, External A* will consider the buckets *Open*$(0, f^*)$, *Open*$(1, f^*)$, ..., *Open*$(f^*, 0)$ – staying on just the f^* diagonal. On the other hand, if the heuristic is equal to zero, it considers the buckets *Open*$(0, 0), \ldots,$ *Open*$(f^*, 0)$, hence simulating a Breadth-First Search. This observation may lead to the conjecture that External A* looks at f^* buckets only. Unfortunately, this is not true.

The total number of required buckets is a function of the last expanded f-value, i.e., the f^* diagonal. In Figure 3.5, a worst case scenario is shown. It should be noted in the figure that expanded buckets are also shown in the 0-th column. Such a situation can arise when the heuristic function assigns a zero estimate to states that are not the target states. Assignments like these, are allowed in an admissible heuristic function.

Trivially, the number of expanded buckets is bounded by $1 + 2 + \ldots f^*$, which can be summed up to $f^*(f^* + 1)/2$. We can, however, achieve a slightly tighter bound. The proof of the following lemma has been extended from a related proof by Edelkamp (1999) given in the context of a symbolic search algorithm, BDDA*.

Lemma 3.3 (Number of Buckets Expanded by External A*) *The number of buckets Open*(i, j) *that are expanded by External A* in an undirected and unweighted state space with a consistent heuristic is bounded by* $\frac{(f^*+1)^2}{3}$.

*Note that the union operators in the pseudo-code should be interpreted as concatenation operators over multi-sets.

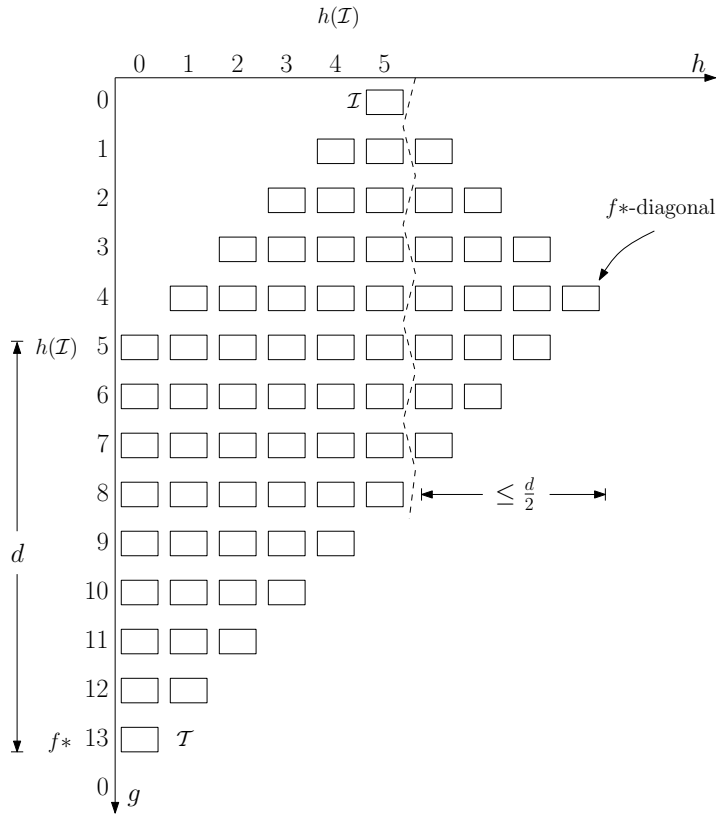

Figure 3.5: Worst-case scenario: Buckets expanded by External A*.

Proof Let $d = f^* - h(\mathcal{I})$ as shown in Figure 3.5. Below $h(\mathcal{I})$ there are at most $d \cdot h(\mathcal{I}) + h(\mathcal{I})$ buckets. The left side of $h(\mathcal{I})$ column (the zig-zag line) has at most $1 + 3 + \ldots + 2(d/2) - 1$ buckets (counted from left to right). Since the sum evaluates to $d^2/4$, we need at most

$$\begin{aligned} d \cdot h(\mathcal{I}) + h(\mathcal{I}) + \frac{d^2}{4} &= (f^* - h(\mathcal{I})) \cdot h(\mathcal{I}) + h(\mathcal{I}) + \frac{(f^* - h(\mathcal{I}))^2}{4} \\ &= \frac{f^{*2} + 2f^* \cdot h(\mathcal{I}) - 3h^2(\mathcal{I}) + 4h(\mathcal{I})}{4} \\ &= \frac{f^{*2}}{4} - \frac{3h^2(\mathcal{I})}{4} + h(\mathcal{I}) \cdot \left(\frac{f^*}{2} + 1\right) \end{aligned}$$

buckets altogether. To compute the maximum number of buckets, we differentiate the equation to find the point where a slope of 0 is reached:

$$\frac{\Delta}{\Delta h(\mathcal{I})} \left(\frac{f^{*2}}{4} - \frac{3h^2(\mathcal{I})}{4} + h(\mathcal{I}) \cdot \left(\frac{f^*}{2} + 1\right)\right) = -\frac{3h(\mathcal{I})}{2} + \frac{f*}{2} + 1\,.$$

Taking right side equals to zero, gives us

$$-\frac{3h(\mathcal{I})}{2} + \frac{f*}{2} + 1 \quad = \quad 0,$$

or

$$h(\mathcal{I}) \quad = \quad \frac{f^* + 2}{3}\,.$$

Hence, the maximum number of buckets

$$\frac{(f^*)^2 + f^* + 1}{3}$$

in the worst case is reached for

$$h(\mathcal{I}) = \frac{f^* + 2}{3}\,.$$

■

If the h-values for adjacent nodes differ by 1, only every second possible f value is encountered.

3.2.4 Duplicates Removal

Similarly to the algorithm of Munagala and Ranade (cf. Chapter 2), we can exploit the observation that in undirected state spaces, duplicates of a state residing in BFS-level i can at most occur in levels i, $i-1$ and $i-2$. Moreover, since h is a total function, we have $h(u) = h(v)$ if $u = v$. This implies the following result.

Lemma 3.4 (Disjoint Buckets) *During the course of executing External A*, for all i, i', j, j' with $j \neq j'$ we have that Open$(i,j) \cap$ Open$(i',j') = \emptyset$.*

Proof Since h is a total function, we have $h(u) = h(v)$ if $u = v$. If $j = h(u)$, $j' = h(v)$ and $j \neq j'$ then $u \in$ *Open*(i,j) and $v \in$ *Open*(i,j') are different, so that *Open*$(i,j) \cap$ *Open*$(i,j') = \emptyset$. ■

Lemma 3.4 allows to restrict duplicate detection to buckets of the same h-value. Assuming an undirected state space problem graph, we can put additional constraints on the restrict aspirants for duplicate detection. In the algorithms of Munagala and Ranade, we already used the observation, that if all duplicates of a state with BFS-level i are removed with respect to the levels i, $i-1$ and $i-2$, then there will be no duplicate state for the entire search process (see Theorem 2.1).

It suffices to perform the duplicate removal only for the bucket that is to be expanded next. The other buckets might not have been fully generated and hence we can save the redundant scanning of the files for every iteration of the inner-most *while* loop. This optimization is in contrast to the Breadth-First Search, where there was no other choice for expansion other than expanding the next layer.

Another consequence of Lemma 3.4 is that during the course of executing External A*, *Open*$(i,j) \cap$ *Open*$(i+c, j-c) = \emptyset$, for all $c \in \mathbb{N}$. This property holds because $i+j = i+c+(j-c)$ and hence preserving the f-value is only possible by decreasing the h value. It does not mean that every two states in the exploration sharing the same f-value are different.

3.3 I/O Complexity of External A*

Since *External A** simulates A*, and only changes the order of expanded states that have the same f-value, completeness and optimality are inherited from the properties shown for A* (Pearl 1985).

Theorem 3.5 (I/O Complexity of External A*) *In an undirected and unweighted state space, if a consistent heuristic function is available, the shortest path from an initial state to a target state can be found by External A* with at most*

$$O\left(scan(|\mathcal{S}|) + sort(|\mathcal{R}|)\right) \quad I/Os. \tag{3.1}$$

Proof By simulating internal A*, the delayed duplicate elimination scheme looks at each edge in the state space problem graph at most once. The individual I/O complexities for every step can be computed as follows:

Expansion: A linear scan of the whole bucket is sufficient for expansion requiring at most

$$O(scan(|Open(g_{\min}, h_{\max})|)) \text{ I/Os.}$$

Since each state is expanded at most once,

$$\sum_{g=0}^{f^*} \sum_{h=0}^{f^*-g} O(scan(|Open(g,h)|)) = O(scan(|\mathcal{S}|)) \,. \tag{3.2}$$

Moreover,

$$O(scan(|Succ(Open(g_{\min}, h_{\max}))|)) \text{ I/Os}$$

are required to sequentially save the successors to the disk. Since each state is generated as a successor for every transition/edge, the total I/O complexity for saving the successors is bounded by:

$$\sum_{g=0}^{f^*} \sum_{h=0}^{f^*-g} O(scan(|Succ(Open(g,h))|)) = O(scan(|\mathcal{R}|)). \tag{3.3}$$

Duplicates removal by external sorting: In this stage, the previous two buckets are subtracted from the new list of successors. Since a bucket may contain successors from three different buckets at $g_{\min} - 1$ row,

$$O(sort(|Succ(Open(g_{\min}-1, h_{\max}))| + |Succ(Open(g_{\min}-1, h_{\max}-1))|$$

$$+|Succ(Open(g_{\min}-1, h_{\max}+1))|)) \text{ I/Os}$$

are required for sorting the list of successors in the bucket $Open_(g_{\min}, h_{\max})$. Since each successor is due to a unique transition, the above quantity can be summed up to

$$\sum_{g=1}^{f^*} \sum_{h=1}^{f^*-g} O(sort(|Succ(Open(g-1,h))| + |Succ(Open(g-1,h-1))|$$

$$+|\mathit{Succ}(\mathit{Open}(g-1,h+1))|)) = O(|\mathit{sort}(|\mathcal{R}|)) \; . \tag{3.4}$$

Duplicates removal by subtraction: As two previous buckets in the same column are enough, file subtraction requires

$$O(\mathit{scan}(|\mathit{Succ}(\mathit{Open}(g_{\min}-1,h_{\max}))| + |\mathit{Succ}(\mathit{Open}(g_{\min}-1,h_{\max}-1))|$$
$$+|\mathit{Succ}(\mathit{Open}(g_{\min}-1,h_{\max}+1))|)$$
$$+\mathit{scan}(|\mathit{Open}(g_{\min}-1,h_{\max})|) + \mathit{scan}(|\mathit{Open}(g_{\min}-2,h_{\max})|)) \;\; \text{I/Os.}$$

Using a similar construction as in Equation 3.4, the first three parts of the above term can be summed up to $O(\mathit{scan}(|\mathcal{R}|)$ I/Os. The last two terms involve the scanning of duplicates-free files that, using a similar construction as in Equation 3.2, lead to a total I/O complexity of $O(|\mathcal{S}|)$ I/Os.

The accumulated I/O complexity of the expansion step is $O(\mathit{scan}(|\mathcal{S}|) + \mathit{scan}(|\mathcal{R}|))$ I/Os. Duplicates removal by external sorting for all buckets adds $\mathit{sort}(|\mathcal{R}|)$ I/Os. Finally, subtraction contributes $O(\mathit{scan}(|\mathcal{S}|)+\mathit{scan}(|\mathcal{R}|))$ I/Os in total. The overall I/O complexity of all three phases accumulates to

$$O\left(\mathit{scan}(|\mathcal{S}|) + \mathit{sort}(|\mathcal{R}|)\right) \;\; \text{I/Os.}$$

■

If we additionally have $|\mathcal{R}| = O(|\mathcal{S}|)$, the complexity reduces to $O(\mathit{sort}(|\mathcal{S}|))$ I/Os.

Internal costs have been neglected in the above analysis. For all operation that are based on batched accesses, we can scale the internal memory requirements down to $O(1)$, namely 2-3 states, depending on the internal memory needs for external sorting. Since each state is considered for expansion only once, the internal time requirements are $|\mathcal{S}|$ multiplied to the duration t_{exp} for successor generation, plus the efforts for internal duplicate elimination and sorting, if applicable. By setting edge weights $c(u,v)$ to $h(u) - h(v) + 1$, for consistent heuristics, A* is a variant of Dijkstra's algorithm that requires internal costs of $O(C \cdot |\mathcal{S}|)$, $C = \max\{c(u,v) \mid v \text{ successor of } u\}$ on a Dial. Due to consistency we have $C \leq 2$, so that, given $|\mathcal{R}| = O(|\mathcal{S}|)$, internal costs are bounded by $O(|\mathcal{S}|\cdot(t_{exp}+\log|\mathcal{S}|))$, where $O(|\mathcal{S}|\log|\mathcal{S}|))$ refers to the internal sorting efforts.

3.4 Solution Reconstruction

In an internal non memory-limited setting, a solution path is constructed by backtracking from the goal node to the start node. This is facilitated by saving with every node a pointer to its predecessor. However, there is one problem in external search: predecessor pointers are not available on disk.

For memory-limited frontier search, a divide-and-conquer solution (Hirschberg 1975) reconstruction is needed for which certain relay layers have to be stored in main memory. In external search, divide-and-conquer solution reconstruction and relay layers are not needed, since the exploration fully resides on disk.

To reconstruct a solution path, we could store predecessor information with each state on disk, and apply backward chaining, starting with the target state. However, this is not strictly necessary: For a state in depth g, we intersect the set of possible predecessors with

the buckets of depth $g-1$. Any state that is in the intersection is reachable on an optimal solution path, so that we can recur. The time complexity is bounded by the scanning time of all buckets in consideration and is in $O(\mathit{scan}(|\mathcal{S}|))$.

In practice, to save disk space when expanding bucket $\mathit{Open}(g_{\min}, h_{\max})$, we can eliminate the bucket $\mathit{Open}(g_{\min}-2, h_{\max})$ after file subtraction. In this case, the solution path has to be reconstructed through divide-and-conquer strategy.

3.5 Refinements and Extensions

3.5.1 Lower Bound

Is $O(\mathit{sort}(|\mathcal{S}|))$ I/O-optimal? To devise lower bounds for delayed duplicate elimination, the following definition for *big-oh* is appropriate:

$$\begin{aligned} O(f(n,M,B)) &= \{g \mid \exists k \in \mathbb{R}^+ \, \forall M, B \in \mathbb{N} \, \exists n_0 \in \mathbb{N} \, \forall n \geq n_0 : \\ &\quad g(n,M,B) \leq k \cdot f(n,M,B)\}. \end{aligned}$$

The classes Θ and Ω are defined analogously. Of course, external algorithms are dependent on the machine architecture parameters M and B. The intuition for universally quantifying M and B, is that the adversary first chooses the machine, and then we, as the *good guys*, evaluate the bound. Aggarwal & Vitter (1987) showed that external sorting has an I/O complexity of

$$\Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$$

and provided two algorithms that are asymptotically optimal. As RAM based *set inequality*, *set inclusion* and *set disjointness* require at least $N\log N - O(N)$ comparisons, the lower bound on the number of I/Os for these problems is also bounded by $\Omega(\mathit{sort}(N))$.

Arge, Knudsen, & Larsen (1993) considered the duplicate elimination problem. A lower bound on the number of comparisons needed is $N\log N - \sum_{i=1}^{k} N_i \log N_i - O(N)$ where N_i is the multiplicity of record i. The authors argue in detail that after the duplicate removal, the total order of the remaining records is known. This corresponds to an I/O complexity of at most

$$\Omega\left(\max\left\{N\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k} N_i \log_{\frac{M}{B}} N_i, N/B\right\}\right).$$

The authors also gave a more complex algorithm based on Mergesort that matches this bound. For the sliding-tiles puzzle with two preceding buckets and a branching factor $b \leq 4$ we have $N_i \leq 8$. For general consistent estimates in uniform graphs, we have $N_i \leq 3c$, with c being an upper bound on the maximal branching factor. An algorithm performs *delayed duplicate bucket elimination*, if it eliminates duplicates within a bucket and with respect to adjacent buckets that are duplicate free.

Theorem 3.6 (I/O Performance Optimality of External A*) *If $|\mathcal{R}| = \Theta(|\mathcal{S}|)$, delayed duplicate bucket elimination in an implicit unweighted and undirected graph, A* with consistent estimates needs at least $\Omega(\mathit{sort}(|\mathcal{S}|))$ I/O operations.*

Proof Since each state gives rise to at most c successors and there are at most 3 preceding buckets in A* search with consistent estimates in an uniformly weighted graph, given

that previous buckets are mutually duplicate free, we have at most $3c$ states that are the same. Therefore, all sets N_i are bounded by $3c$. Since k is bounded by N, we have that $\sum_{i=1}^{k} N_i \log N_i$ is bounded by $k \cdot 3c \log 3c = O(N)$. Therefore, the lower bound for duplicate elimination for N states is $\Omega(\mathit{sort}(N) + \mathit{scan}(N))$. ■

A related lower bound, also applicable to the multiple disk model (Munagala & Ranade 1999), establishes that solving the duplicate elimination problem with N elements having P different values needs at least $\Omega(\frac{N}{P}\mathit{sort}(P))$ I/Os. This complexity is due to the fact that the depth of any decision tree for the duplicate elimination problem is at least $N \log(P/2)$. For state space search with consistent estimates and bounded branching factor, we assume to have $P = \Theta(N) = \Theta(|\mathcal{R}|) = \Theta(|\mathcal{S}|)$, so that the I/O complexity reduces to $O(\mathit{sort}|\mathcal{S}|)$.

3.5.2 Pipelining

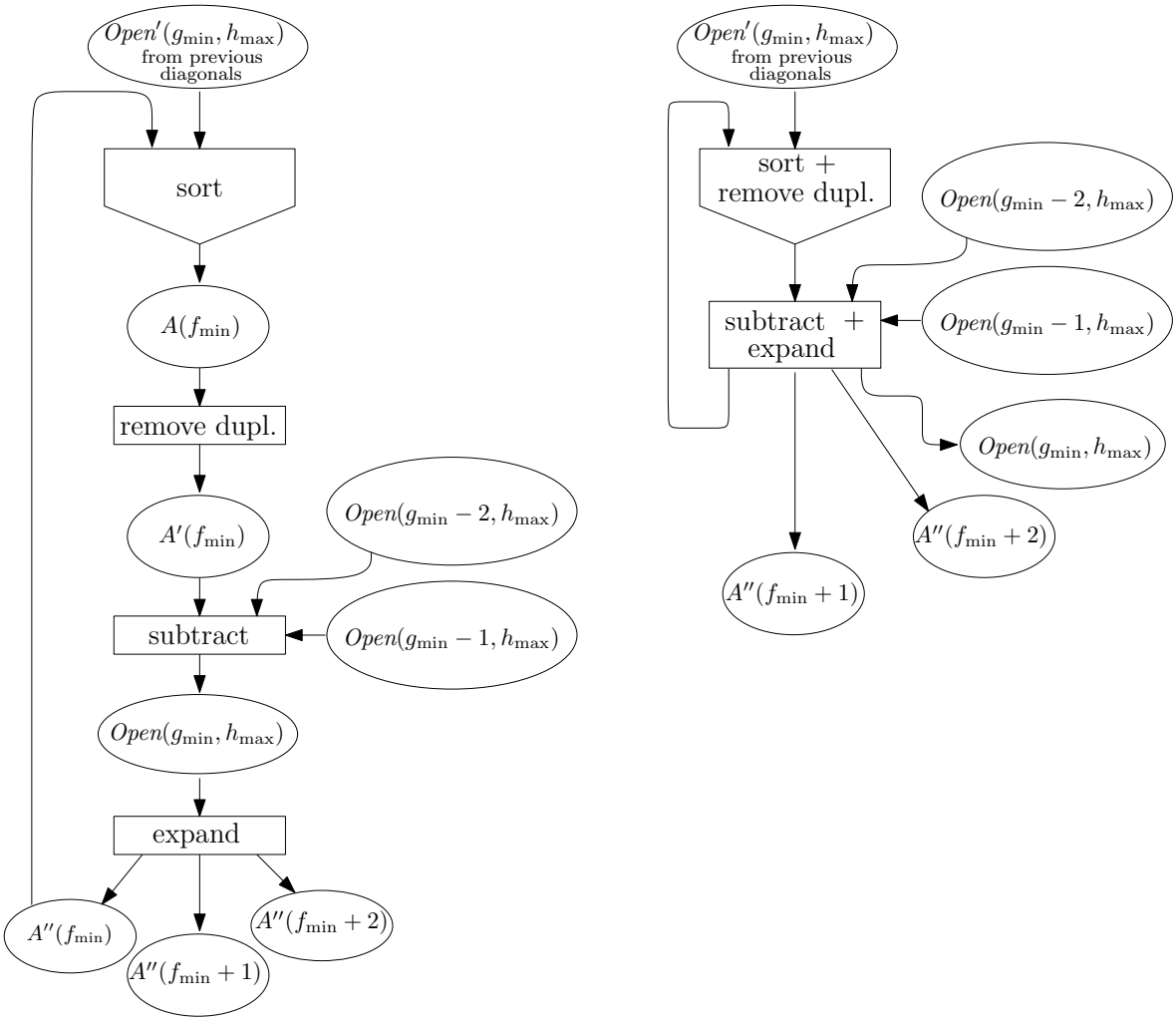

Figure 3.6: External A* without Pipelining (left), with Pipelining (right).

Ajwani, Dementiev, & Meyer (2006) proposed an efficient implementation of external

BFS for explicit graphs. The underlying concept is called *Pipelining*. It targets on reducing the number of I/O operations by feeding the input of one stage directly into the other stage without creating an intermediate file. Using similar notations as presented in (Ajwani, Dementiev, & Meyer 2006), we can extend the concept of pipelining to External A* in implicit graphs.

In Figure 3.6, we demonstrate the workings of External A* with and without pipelining by a flow diagram. The ovals represent a file on the disk. A sorting operation is depicted with a five-sided polygon, while scanning operations for states' expansion and subtraction are illustrated with rectangles. The files A, A' and A'' are temporary files that are generated as an output to each phase. To the left of Figure 3.6, External A* with a trivial implementation is depicted. On the right side, External A* is extended with pipelining.

A comparison of the diagrams reveal three major improvements. First, the external sorting and duplicates removal are combined in a single routine. This saves **three** scanning operations each of complexity $\mathit{scan}(|\mathcal{R}|)$: one for writing the sorting output, one for reading the sorted file to remove duplicate states, and one for writing the output after removing consecutive duplicates. Second, since the output of subtraction is a duplicate free file, it can be used directly for expansion. This is realized by combining both the subtraction and expansion modules. But, since the actual output of the subtraction is also needed for later references, the stream of states is also flushed to the file $\mathit{Open}(g_{\min}, h_{\max})$. With this optimization, we have saved **two** scanning operations: one scanning operation with $\mathit{scan}(|\mathcal{R}|)$ I/Os for reading the file for subtraction and one scanning operation with a complexity of $\mathit{scan}(|\mathcal{S}|)$ for reading during expansion.

The third optimization is in redirecting the output of expansion phase directly into the sorting module. For external sorting, one scan is required to create the memory-sized sorted sequences. Recall that each bucket can have successor states from three different buckets that are in the previous row. Let us assume that the successors of a bucket are uniformly distributed. We can then bound the sizes of each $A''(f_{\min})$, $A''(f_{\min}+1)$, $A''(f_{\min}+2)$ in the left diagram as being equal to $1/3 \cdot |\mathcal{R}|$. In pipelining, using the output of the expansion phase, **one** scanning operation of $\mathit{scan}(1/3|\mathcal{R}|)$ I/Os can be completely avoided.

In External A*, the above mentioned pipelining can easily save

$$(4 + 1/3) \cdot \mathit{scan}(|\mathcal{R}|) + \mathit{scan}(|\mathcal{S}|)$$

I/O operations. Assume that for an input of size N, $\mathit{sort}(N) = 2 \cdot \mathit{scan}(N)$. This follows by carefully choosing the block size B (Dementiev 2006) and given that the machine allows opening of sufficient number of file pointers. It is easy to see in Figure 3.6 that pipelining saves almost *half* of the I/O operations required by External A*!

In brief, although the asymptotic I/O complexity is not affected by pipelining, in practice pipelining can result in considerable savings in terms of time.

3.6 Experiments

Implementation: External A* has been implemented in C++ along with external scanning, external mergesort and external set subtraction routines. The fundamental component, or the building block, of the implementation is the class called `Bucket`. It provides the functionality of a buffered file. A `Bucket` consists of an array of states and a set of memory and

file pointers for controlling the flow of data between the RAM and the hard disk. New states are inserted in a bucket until it is full. At that time, an internal sorting is invoked to bring the duplicate states in the buffer adjacent to each other. Internal sorting is done by the built-in Quick-sort `qsort` routine. The sorted buffer is flushed to the disk, while removing the duplicate states in a single pass. External sorting is realized by first scanning the file for the location of sorted buffers. If enough memory is available and the system allows the opening of sufficient file pointers, the sorted buffers are merged into a sorted output stream. Duplicate states are removed from the output stream before being flushed to the disk. Subtraction of a previous level bucket from a current bucket is performed by simultaneously scanning the two files.

One interesting feature of our implementation from a practical point of view is the ability to pause and resume the program execution. For large problem instances, this is a desirable feature in case we reach the system bounds of secondary storage, and after upgrading the system, want to resume the execution.

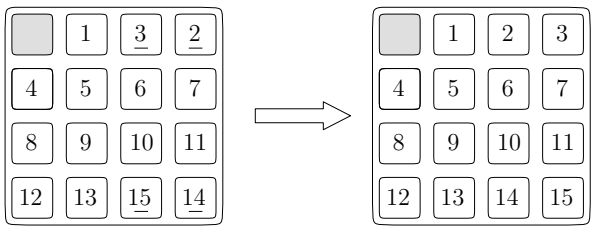

Figure 3.7: Sliding-tiles puzzle in 4 x 4 configuration. On the left is a random permutation with 4 misplaced tiles; the right side shows the solved puzzle. The Manhattan distance between the two is 4.

Sliding-Tiles Puzzle: We selected 15-Puzzle problem instances for the evaluation of External A*. The puzzle consists of 15 tiles and a blank space arranged in a 4×4 pattern. Each tile has a number and the only action allowed is to move a tile to the blank place. The target is to order the tiles such that each numbered tile is placed at its corresponding position. Figure 3.7 shows an example puzzle. Many instances cannot be solved internally with A* and the Manhattan distance. Each state is packed into 8 bytes corresponding to two 32-bits integers. In 15-Puzzle, at a given time, there can only be three active buckets: the bucket $Open(i, j)$ which is being expanded and the two buckets $Open(i+1, j-1)$ and $Open(i+1, j+1)$ for successors' collection. This implies an internal memory requirement of at least $3B$.

The program utilizes an implicit priority queue. For sliding-tiles puzzles, during expansion, the successor's f value differs from the parent state by exactly 2. This implies that in case of an empty diagonal, the program terminates. By the restriction for f-values in the sliding-tiles puzzles, only about half the number of buckets have to be allocated. Note that f^* is not known in advance and, consequently, we have to construct and maintain the files on the fly.

In Table 3.1 we show the diagonal pattern of states that appeared while solving a toy problem instance where the tiles 1 and 2, and 4 and 5 were swapped. The entry $x + y$ in the cell (i, j) implies that x and y number of states are generated from the expansion of $Open(i-1, j-1)$ and $Open(i-1, j+1)$, respectively.

The pattern of duplicate states in each bucket is shown in Table 3.2. An entry $u + v$ in

g/h	1	2	3	4	5	6	7	8	9	10	11
0	-	-	-	1+0	-	-	-	-	-	-	-
1	-	-	-	-	2+0	-	-	-	-	-	-
2	-	-	-	0+4	-	2+0	-	-	-	-	-
3	-	-	-	-	7+3	-	4+0	-	-	-	-
4	-	-	-	0+7	-	13+4	-	10+0	-	-	-
5	-	-	-	-	5+15	-	24+10	-	24+0	-	-
6	-	-	-	0+6	-	12+26	-	46+28	-	44+0	-
7	-	-	-	-	9+10	-	20+51	-	99+57	-	76+0
8	-	-	-	0+8	-	15+25	-	48+137	-	195+0	-
9	-	-	-	-	4+17	-	45+52	-	203+0	-	-
10	-	-	-	0+3	-	13+49	-	92+0	-	-	-
11	-	-	-	-	2+19	-	46+0	-	-	-	-
12	-	-	-	0+5	-	31+0	-	-	-	-	-
13	-	-	0+2	-	10+0	-	-	-	-	-	-
14	-	0+2	-	5+0	-	-	-	-	-	-	-
15	0+2	-	5+0	-	-	-	-	-	-	-	-

Table 3.1: States inserted in the buckets for the instance (0 2 1 3 5 4 6 7 8 9 10 11 12 13 14 15).

the cell (i, j) implies that u states are the duplicate states *within* a bucket and v states are the duplicates that are found due to the *subtraction* of the states of the bucket $(i - 2, j)$. The actual number of states remaining in a bucket after duplicate removal and subtraction can be obtained by subtracting the (i, j)th entry of Table 3.2 from the (i, j)th entry of Table 3.1.

The impact of internal buffer size on the I/O performance is clearly observable in Table 3.3. We show the I/O performance of two instances by varying the internal buffer size B. A larger buffer implies fewer flushes during writing, fewer block reads during expansion and less processing time due to internally sorting larger but fewer buffers. The presented I/O and timing data in this table were collected using the task manager of Windows XP.

Korf's 100 instances: For a more extensive performance evaluation of External A*, we chose the 100 15-Puzzle instances reported by Korf's seminal paper on IDA* algorithm (Korf 1985). External A* has been able to solve all 100 instances. Tables 3.4 and 3.5 report the results of external exploration using External A*. This set of experiments is performed on a dual processor AMD Opteron machine with 4 Gigabytes of RAM and 6 Terabytes of remotely shared hard disk space available through NFS (Network File System) running Linux. The actual instance is shown in the second column labeled $\mathcal{I}$ for initial state. The number at the i-th position corresponds to the name of the tile placed at position i. The *blank* is represented by '0'. In the third column, the initial heuristic estimate $h(\mathcal{I})$ is shown, while the fourth column reports the shortest path distance to the goal state obtained by External A* that completely matches with that of (Korf 1985). Columns 5, 6, & 7 show the number of transitions performed, the total number of nodes expanded till the goal is found, and the number of nodes actually stored after duplicates removal, respectively. Since we used a consistent heuristic, for 15-Puzzle instances, the algorithm can stop when the goal node is generated with an f-value equal to f_{min} (the working diagonal in the matrix). Finally, total space consumption on the disk is reported in the last column, in Megabytes. Through out all the runs, the total process size in RAM remained constant at 1.2 Gigabytes.

g/h	1	2	3	4	5	6	7	8	9	10	11
0	-	-	-	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	1+1	-	-	-	-	-	-	-
3	-	-	-	-	2+2	-	-	-	-	-	-
4	-	-	-	3+2	-	3+2	-	-	-	-	-
5	-	-	-	-	8+6	-	6+4	-	-	-	-
6	-	-	-	1+2	-	16+12	-	14+10	-	-	-
7	-	-	-	-	6+6	-	24+24	-	26+24	-	-
8	-	-	-	3+10	-	10+10	-	52+50	-	-	-
9	-	-	-	-	9+7	-	29+23	-	-	-	-
10	-	-	-	0+2	-	21+20	-	-	-	-	-
11	-	-	-	-	6+5	-	-	-	-	-	-
12	-	-	-	0+1	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-	-
15	1+0	-	-	-	-	-	-	-	-	-	-

Table 3.2: Duplicate states within a bucket + Duplicate states in top layers for the instance (0 2 1 3 5 4 6 7 8 9 10 11 12 13 14 15).

Initial State	B	I/O Reads	I/O Writes	Time (*sec*)
(0 1 2 3 5 4 7 6 8 9 10 11 12 13 14 15)	10	5,214	6,525	2
	25	3,086	3,016	1
	50	2,371	1,843	< 1
	100	2,022	1,265	< 1
(0 2 1 3 5 4 7 6 8 9 13 11 12 10 14 15)	50	7,312	8,463	4
	75	6,093	6,377	3
	100	5,481	5,329	3
	150	4,873	4,287	3

Table 3.3: Effects on I/O performance due to different internal buffer sizes

In the two last columns, we report the time taken by each instance. We distinguish between the wall-clock time and the total time spent by the CPU. The main reason for discrepancy between the two is the I/O wait time. Moreover, since the CPU was a shared unit on a network cluster, the CPU wait time also results in an increase in the total time.

3.7 Related Work

The algorithm presented in this chapter shares some resemblance with BDDA* and Set A*. BDDA* (Edelkamp & Reffel 1998) algorithm was presented in the context of Binary Decision Diagrams (BDD) based heuristic search. Heuristic search requires the heuristic estimates to be saved with every state. In case of BDD based search, this problem becomes more crucial because of the implicit representation of states within a BDD.

Jensen, Bryant, & Veloso (2002) proposed not to have a single BDD but instead several BDDs based on the depth g and the heuristic h value of the states. The algorithm divides the *Open* list into sets that can be identified by the g and h values of the states in that set.

S. #	$\mathcal{I}$	$h(\mathcal{I})$	f^*	Transitions	Expanded	Stored	Disk MB	Time Real	Time CPU
1	14 13 15 7 11 12 9 5 6 0 2 1 4 8 10 3	41	57	138,741,107	46,790,659	112,386,916	860	2m15.575s	1m42.444s
2	13 5 4 10 9 12 8 14 2 3 7 1 0 15 11 6	43	55	28,820,796	9,764,504	24,416,690	183	0m32.915s	0m20.711s
3	14 7 8 2 13 11 10 4 9 12 5 0 3 6 1 15	41	59	363,376,327	121,941,366	294,204,177	2,244	5m24.762s	4m33.137s
4	5 12 10 7 15 11 14 0 8 2 1 13 3 4 9 6	42	56	69,532,034	23,407,012	69,532,034	437	1m10.833s	0m50.979s
5	4 7 14 13 10 3 9 12 11 5 6 15 1 2 8 0	42	56	29,454,271	9,952,495	24,972,095	191	0m33.728s	0m21.310s
6	14 7 1 9 12 3 6 15 8 11 2 5 10 0 4 13	36	52	12,887,335	4,294,115	10,682,639	82	0m17.280s	0m9.539s
7	2 11 15 5 13 4 6 7 12 8 10 1 9 3 14 0	30	52	72,400,900	24,172,359	57,054,953	435	1m16.588s	0m54.070s
8	12 11 15 3 8 0 4 2 6 13 9 5 14 1 10 7	32	50	25,525,113	8,495,695	20,423,730	155	0m29.665s	0m18.748s
9	3 14 9 11 5 4 8 2 13 12 6 7 10 1 15 0	32	46	3,239,383	1,070,041	2,648,133	20	0m7.853s	0m2.768s
10	13 11 8 9 0 15 7 10 4 3 6 14 5 12 2 1	43	59	231,478,022	77,559,095	189,855,528	1,448	3m39.456s	2m53.288s
11	5 9 13 14 6 3 7 12 10 8 4 0 15 2 11 1	43	57	47,380,696	16,036,959	39,629,613	303	1m3.996s	0m34.705s
12	14 1 9 6 4 8 12 5 7 2 3 0 10 11 13 15	35	45	493,990	163,768	423,522	4	0m5.431s	0m0.970s
13	3 6 5 2 10 0 15 14 1 4 13 12 9 8 11 7	36	46	6,689,109	2,238,659	5,393,386	41	0m16.999s	0m5.224s
14	7 6 8 1 11 5 14 10 3 4 9 13 15 2 0 12	41	59	297,583,236	100,411,201	238,446,001	1,818	4m55.536s	3m48.290s
15	13 11 4 12 1 8 9 15 6 5 14 2 7 3 10 0	44	62	342,465,010	116,893,168	284,077,111	2,166	5m24.010s	4m18.496s
16	1 3 2 5 10 9 15 6 8 14 13 11 12 4 7 0	24	42	5,180,710	1,703,049	3,975,725	31	0m12.706s	0m4.307s
17	15 14 0 4 11 1 6 13 7 5 8 9 3 2 10 12	46	66	633,735,361	218,977,080	528,156,645	4,031	9m51.557s	8m5.486s
18	6 0 14 12 1 15 9 10 11 4 7 2 8 3 5 13	43	55	17,656,188	5,991,730	15,123,018	116	0m24.795s	0m12.795s
19	7 11 8 3 14 0 6 15 1 4 13 9 5 12 2 10	36	46	2,044,657	677,551	1,698,609	14	0m7.338s	0m1.980s
20	6 12 11 3 13 7 9 15 2 14 8 10 4 1 5 0	36	52	24,540,861	8,167,234	20,223,289	155	0m29.433s	0m17.986s
21	12 8 14 6 11 4 7 0 5 1 10 15 3 13 9 2	34	54	56,419,409	19,037,308	44,871,689	357	1m35.580s	0m38.270s
22	14 3 9 1 15 8 4 5 11 7 10 13 0 2 12 6	41	59	165,491,810	55,457,756	134,810,471	1,045	3m58.957s	1m51.880s
23	10 9 3 11 0 13 2 14 5 6 4 7 8 15 1 12	33	49	15,973,541	5,329,963	12,863,941	107	0m26.228s	0m10.660s
24	7 3 14 13 4 1 10 8 5 12 9 11 2 15 6 0	34	54	52,707,594	17,615,530	42,090,433	334	1m9.044s	0m35.190s
25	11 4 2 7 1 0 10 15 6 9 14 8 3 13 5 12	32	52	64,642,092	21,661,428	50,597,016	408	1m35.300s	0m44.370s
26	5 7 3 12 15 13 14 8 0 10 9 6 1 4 2 11	40	58	243,606,293	82,437,599	197,906,900	1,525	4m56.319s	2m46.690s
27	14 1 8 15 2 6 0 3 9 12 10 13 4 7 5 11	33	53	121,484,199	40,351,956	95,577,145	744	2m50.882s	1m22.100s
28	13 14 6 12 4 5 1 0 9 3 10 2 15 11 8 7	36	52	11,720,647	3,988,883	9,719,057	81	0m21.622s	0m7.970s
29	9 8 0 2 15 1 4 14 3 10 7 5 11 13 6 12	38	54	45,923,548	15,462,811	37,195,403	294	1m1.864s	0m30.900s
30	12 15 2 6 1 14 4 8 5 3 7 0 10 13 9 11	35	47	2,708,802	907,424	2,254,616	31	0m8.577s	0m2.110s
31	12 8 15 13 1 0 5 4 6 3 2 11 9 7 14 10	38	50	2,686,215	910,375	2,270,950	24	0m13.060s	0m1.980s
32	14 10 9 4 13 6 5 8 2 12 7 0 1 3 11 15	43	59	219,094,538	73,987,773	180,410,748	1,398	4m22.527s	2m28.840s
33	14 3 5 15 11 6 13 9 0 10 2 12 4 1 7 8	42	60	237,639,205	80,577,859	195,773,501	1,510	5m26.331s	2m42.470s
34	6 11 7 8 13 2 5 4 1 10 3 9 14 0 12 15	36	52	27,063,242	9,014,788	22,333,992	181	0m49.998s	0m18.000s
35	1 6 12 14 3 2 15 8 4 5 13 9 0 7 11 10	39	55	57,658,630	19,510,281	47,362,247	379	1m23.401s	0m38.660s
36	12 6 0 4 7 3 15 1 13 9 8 11 2 14 5 10	36	52	26,499,809	8,912,473	21,365,881	173	0m44.917s	0m17.440s
37	8 1 7 12 11 0 10 5 9 15 6 13 14 2 3 4	40	58	245,439,808	82,633,947	197,787,534	1,530	5m8.648s	2m50.380s
38	7 15 8 2 13 6 3 12 11 0 4 10 9 5 1 14	41	53	12,082,589	4,113,433	10,277,805	100	0m36.103s	0m7.970s
39	9 0 4 10 1 14 15 3 12 6 5 7 11 13 8 2	35	49	12,687,200	4,228,910	10,160,576	85	0m25.986s	0m8.740s
40	11 5 1 14 4 12 10 0 2 7 13 3 9 15 6 8	36	54	75,071,930	25,185,960	60,262,426	476	1m53.759s	0m50.980s
41	8 13 10 9 11 3 15 6 0 1 2 14 12 5 4 7	36	54	69,980,238	23,253,223	56,793,647	448	1m41.056s	0m46.540s
42	4 5 7 2 9 14 12 13 0 3 6 11 8 1 15 10	30	42	629,613	210,854	524,441	33	0m6.354s	0m0.940s
43	11 15 14 13 1 9 10 4 3 6 2 12 7 5 8 0	48	64	129,382,960	44,510,663	112,599,398	879	2m57.131s	1m25.250s
44	12 9 0 6 8 3 5 14 2 4 11 7 10 1 15 13	32	50	28,984,788	9,666,224	22,895,602	186	0m48.369s	0m19.750s
45	3 14 9 7 12 15 0 4 1 8 5 6 11 10 2 13	39	51	7,168,353	2,420,603	6,001,694	50	0m14.975s	0m5.030s
46	8 4 6 1 14 12 2 15 13 10 9 5 3 7 0 11	35	49	16,540,654	5,498,947	13,373,141	111	0m33.419s	0m11.310s
47	6 10 1 14 15 8 3 5 13 0 2 7 4 9 11 12	35	47	2,221,034	742,024	1,828,633	28	0m9.128s	0m2.000s
48	8 11 4 6 7 3 10 9 2 12 15 13 0 1 5 14	39	49	1883769	625132	1616010	28	0m8.190s	0m1.670s
49	10 0 2 4 5 1 6 12 11 13 9 7 15 3 14 8	33	59	722,612,694	243,790,911	556,756,865	4,272	15m38.469s	8m41.840s
50	12 5 13 11 2 10 0 9 7 8 4 3 14 6 15 1	39	53	23,035,882	7,659,605	19,014,788	154	0m37.393s	0m15.190s

Table 3.4: External A* on Korf's 100 instances of 15-Puzzle (1-50). RAM consumption: 1.2 Gigabytes.

This partitioning on the *Open* list induces a matrix with rows identifying the depth g and columns the heuristic value h. Each cell $Open(i, j)$ then contains a BDD of all the states with their g value as i and h value as j. The algorithm proceeds in a diagonal manner over the matrix, expanding states with the same $f = g + h$ value. Ties are broken by prioritizing the cell with the smallest g value.

There is a tight connection between the exploration of externally stored sets of states, and an efficient *symbolic* representation for sets of states with *Binary Decision Diagrams (BDDs)*. The design of existing symbolic heuristic search algorithms seems to be strongly influenced by the delayed duplication and external set manipulation. Recently, Edelkamp (2005) has successfully combined both external and symbolic heuristic search. The author suggests to represent each bucket of External A* as a BDD.

There are two related but independent research results, considering external Best-First exploration that we will discuss in the following.

S. #	$\mathcal{I}$	$h(\mathcal{I})$	f^*	Transitions	Expanded	Stored	Disk MB	Time Real	Time CPU
51	10 2 8 4 15 0 1 14 11 13 3 6 9 7 5 12	44	56	42,721,229	14,406,858	35,847,441	285	1m2.891s	0m27.860s
52	10 8 0 12 3 7 6 2 1 14 4 11 15 13 9 5	38	56	129,297,474	43,309,829	102,601,257	800	2m57.029s	1m28.610s
53	14 9 12 13 15 4 8 10 0 2 1 7 3 11 5 6	50	64	521,381,622	177,244,032	432,934,965	3,324	10m12.019s	5m57.780s
54	12 11 0 8 10 2 13 15 5 4 7 3 6 9 14 1	40	56	83,518,765	27,996,743	67,331,539	531	1m55.830s	0m56.360s
55	13 8 14 3 9 1 0 7 15 5 4 10 12 2 6 11	29	41	601,350	199,298	483,920	31	0m6.450s	0m0.900s
56	3 15 2 5 11 6 4 7 12 9 1 0 13 14 10 8	29	55	421,060,873	141,157,390	315,495,465	2,429	8m13.025s	5m0.980s
57	5 11 6 9 4 13 12 0 8 2 15 10 1 7 3 14	36	50	6,384,253	2,142,586	5,296,630	45	0m14.628s	0m4.500s
58	5 0 15 8 4 6 1 14 10 11 3 9 7 12 2 13	37	51	6,204,286	2,071,549	5,127,083	44	0m14.047s	0m4.380s
59	15 14 6 7 10 1 0 11 12 8 4 9 2 5 13 3	35	57	474,750,228	158,913,129	361,645,335	2,788	9m34.707s	5m38.090s
60	11 14 13 1 2 3 12 4 15 7 9 5 10 6 8 0	48	66	2,269,242,379	767,584,678	1,869,286,091	14,287	50m17.243s	26m51.980s
61	6 13 3 2 11 9 5 10 1 7 12 14 8 4 0 15	31	45	5,803,018	1,922,762	4,674,420	38	0m15.488s	0m3.900s
62	4 6 12 0 14 2 9 13 11 8 3 15 7 10 1 5	43	57	28,366,273	9,531,413	24,203,301	196	0m41.241s	0m18.620s
63	8 10 9 11 14 1 7 15 13 4 0 12 6 2 5 3	40	56	231,048,052	76,578,003	180,410,146	1,396	5m4.382s	2m40.110s
64	5 2 14 0 7 8 6 3 11 12 13 15 4 10 9 1	31	51	57,107,378	19,004,600	44,461,520	353	1m16.079s	0m38.820s
65	7 8 3 2 10 12 4 6 11 13 5 15 0 1 9 14	31	47	9,911,597	3,300,492	7,862,304	63	0m19.610s	0m6.820s
66	11 6 14 12 3 5 1 15 8 0 10 13 9 7 4 2	41	61	812,612,732	275,076,044	647,743,576	4,961	15m22.699s	9m36.780s
67	7 1 2 4 8 3 6 11 10 15 0 5 14 12 13 9	28	50	79,064,219	26,260,192	60,043,601	470	1m50.018s	0m54.450s
68	7 3 1 13 12 10 5 2 8 0 6 11 14 15 4 9	31	51	56,808,903	18,932,593	44,409,221	352	1m17.404s	0m38.710s
69	6 0 5 15 1 14 4 9 2 13 8 10 11 12 7 3	37	53	49,900,395	16,734,666	39,763,650	316	1m11.376s	0m33.630s
70	15 1 3 12 4 0 6 5 2 8 14 9 13 10 7 11	30	52	85,159,420	28,389,386	66,538,015	521	1m55.517s	0m57.650s
71	5 7 0 11 12 1 9 10 15 6 2 3 8 4 13 14	30	44	5,585,622	1,854,930	4,418,830	39	0m12.370s	0m4.160s
72	12 15 11 10 4 5 14 0 13 7 1 2 9 8 3 6	38	56	338,858,471	113,914,825	260,269,048	2,005	6m42.257s	3m59.650s
73	6 14 10 5 15 8 7 1 3 4 2 0 12 9 11 13	37	49	2,524,502	843,056	2,166,095	31	0m8.896s	0m2.000s
74	14 13 4 11 15 8 6 9 0 7 3 1 2 10 12 5	46	56	5,193,296	1,753,029	4,564,533	43	0m11.570s	0m3.800s
75	14 4 0 10 6 5 1 3 9 2 13 15 12 7 8 11	30	48	35,484,987	11,702,492	27,574,353	218	0m45.361s	0m23.820s
76	15 10 8 3 0 6 9 5 1 14 13 11 7 2 12 4	41	57	88,470,814	29,503,808	71,726,654	563	1m54.070s	0m59.360s
77	0 13 2 4 12 14 6 9 15 1 10 3 11 5 8 7	34	54	27,600,333	9,319,512	27,600,333	183	0m40.683s	0m18.300s
78	3 14 13 6 4 15 8 9 5 12 10 0 2 7 1 11	41	53	9,179,796	3,102,048	7,818,326	64	0m16.664s	0m6.420s
79	0 1 9 7 11 13 5 3 14 12 4 2 8 6 10 15	28	42	729,935	240,336	605,534	31	0m6.080s	0m0.870s
80	11 0 15 8 13 12 3 5 10 1 4 6 14 9 7 2	43	57	49,074,024	16,626,064	41,511,200	327	1m1.342s	0m32.020s
81	13 0 9 12 11 6 3 5 15 8 1 10 4 14 2 7	39	53	9,700,043	3,276,877	8,083,983	66	0m18.596s	0m6.440s
82	14 10 2 1 13 9 8 11 7 3 6 12 15 5 4 0	40	62	1,634,489,339	549,557,758	1,267,637,450	9,695	30m25.006s	19m41.020s
83	12 3 9 1 4 5 10 2 6 11 15 0 14 7 13 8	31	49	19,804,883	6,565,822	15,542,109	128	0m41.677s	0m13.370s
84	15 8 10 7 0 12 14 1 5 9 6 3 13 11 4 2	37	55	128,873,666	43,387,389	103,980,661	809	2m45.908s	1m27.590s
85	4 7 13 10 1 2 9 6 12 8 14 5 3 0 11 15	32	44	1,701,401	563,689	1,388,925	26	0m9.127s	0m1.650s
86	6 0 5 10 11 12 9 2 1 7 4 3 14 8 13 15	35	45	2,567,676	849,706	2,125,377	27	0m8.965s	0m2.090s
87	9 5 11 10 13 0 2 1 8 6 14 12 4 7 3 15	34	52	54,314,155	18,153,779	43,692,751	344	1m11.094s	0m36.500s
88	15 2 12 11 14 13 9 5 1 3 8 7 0 10 6 4	43	65	2,956,384,330	999,442,568	2,325,362,973	17,769	54m8.057s	35m33.160s
89	11 1 7 4 10 13 3 8 9 14 0 15 6 5 2 12	36	54	41,917,341	14,036,832	33,882,122	270	0m59.447s	0m27.910s
90	5 4 7 1 11 12 14 15 10 13 8 6 2 0 9 3	36	50	10,406,566	3,510,468	8,597,942	69	0m18.743s	0m7.310s
91	9 7 5 2 14 15 12 10 11 3 6 1 8 13 0 4	41	57	213,962,766	72,334,731	172,968,798	1,338	3m54.637s	2m26.490s
92	3 2 7 9 0 15 12 4 6 11 5 14 8 13 10 1	37	57	451,859,674	151,699,571	349,123,403	2,684	8m48.693s	5m19.650s
93	13 9 14 6 12 8 1 2 3 4 0 7 5 10 11 15	34	46	3,611,926	1,201,535	2,981,207	31	0m9.937s	0m2.750s
94	5 7 11 8 0 14 9 13 10 12 3 15 6 1 4 2	45	53	961,296	323,299	868,960	34	0m6.389s	0m1.080s
95	4 3 6 13 7 15 9 0 10 5 8 11 2 12 1 14	34	50	8,807,790	2,947,831	7,261,012	59	0m16.577s	0m5.980s
96	1 7 15 14 2 6 4 9 12 11 13 3 0 8 5 10	35	49	15,386,755	5,119,907	12,523,237	103	0m23.822s	0m10.170s
97	9 14 5 7 8 15 1 2 10 4 13 6 12 0 11 3	32	44	2,243,134	745,066	1,838,220	28	0m8.365s	0m1.910s
98	0 11 3 12 5 2 1 9 8 10 14 15 7 4 13 6	34	54	100,078,264	33,239,269	78,519,758	613	2m6.379s	1m8.610s
99	7 15 4 0 10 9 2 5 12 11 13 6 1 3 14 8	39	57	66,905,668	22,729,559	55,146,752	434	1m32.709s	0m44.660s
100	11 4 0 8 6 10 5 13 12 7 14 3 1 2 9 15	38	54	49,686,112	16,557,635	40,434,617	325	28m4.357s	15m52.560s

Table 3.5: External A* on Korf's 100 instances of 15-Puzzle (51-100). RAM consumption: 1.2 Gigabytes.

3.7.1 Best-First Frontier Search

Korf (2004) (see also (Korf 2005)) extended Frontier Search with delayed duplicate detection to Best-First Search. He also considered omission of the visited list as proposed for *Frontier Search*. It turned out that any two of the three options were compatible: Breadth-First Frontier Search with delayed duplicate detection, Best-First Frontier Search, and Best-First Search with non-reduced visited list saved on external memory. For the latter case, Korf simulated the buffered traversal in a Dial priority queue.

In External A*, we have *successfully combined* all three facilities: best-first search, delayed duplicate detection, and omission of previously visited states.

As an additional feature, Korf showed how external sorting can be avoided, by a selection of hash functions that split larger files into smaller pieces which fit into main memory. As with the h-value, in our case, a state and its duplicate will have the same hash address. This method is termed as *hash-based delayed duplicate detection*.

3.7.2 Structured Duplicate Detection

The second algorithm for External Memory Best-First Search is a combination of *Structured Duplicate Detection* (SDD) and *Breadth-First Heuristic Search* (BFHS). The basic idea behind SDD is to exploit the regularities in the problem structure to accelerate duplicate detection. Zhou & Hansen (2004b) incorporated a projection function that partitions a state space into disjoint regions. The graph of those regions is termed as an *abstract state space* to differentiate from the original *concrete state space*.

Projections are state space homomorphisms that preserve the paths in the concrete state space. For each pair of consecutive abstract states, there exists a pair of concrete states. Let $\phi : \mathcal{S} \rightarrow \mathcal{S}_\phi$ be the projection function that maps the concrete states $\mathcal{S}$ to abstract states $\mathcal{S}_\phi$. The abstract transitions are denoted by $\mathcal{R}_\phi$. Given a path preserving abstraction, for a neighbor s' of a concrete state s, if $(s, s') \in \mathcal{R}$, then $(\phi(s), \phi(s')) \in \mathcal{R}_\phi \vee \phi(s) = \phi(s')$.

In the example of 15-Puzzle provided by the authors, the projection function was based on assigning the states that have the same blank position into one partition. Figure 3.8 shows an abstract state space graph of 15-Puzzle. A node B_i is the common abstraction of all the states that have the *blank* at the i-th position.

Structured duplicate detection, in contrast to delayed duplicate detection, does not use external sorting to remove duplicates. It, instead, relies on a set of hash tables that are associated with each abstract node. A hash table $H[s_\phi]$ contains all the concrete states s, such that $\phi(s) = s_\phi \in \mathcal{S}_\phi$. Upon expansion, the successors would either belong to the same abstract state or to one of the neighboring partitions. Hence, if the hash tables of the current state and of its neighbors are kept in RAM, it can be guaranteed that all duplicate states are detected as soon as they are generated. The rest of the hash tables are kept on the hard disk. In the case of 15-Puzzle, the shaded area in Figure 3.8 depicts the portion of the abstract graph that has to reside in RAM in order to capture all the duplicate states that are generated by expanding states in $H[B_5]$. The scope of duplicates include the hash tables $H[B_1]$, $H[B_4]$, $H[B_6]$, and $H[B_9]$. SDD requires all five hash tables to reside in RAM to guarantee that no node is expanded twice.

As an underlying search algorithm that picks the next abstract state to expand, the authors suggested to use Breadth-First Heuristic Search, which is explained later in this section. The I/O complexity is bounded by $O(|\mathcal{R}_\phi| \cdot \mathit{scan}(|\mathcal{S}|))$, where $\mathcal{R}_\phi$ is the set of transitions in the abstract state space graph.

Structured duplicate detection is crucially dependent on the availability of suitable partition functions. In contrast, External A* does not rely on any partitioning beside the h function and it is not required that all the neighboring hash tables fit into the RAM.

In fact, these two approaches (Best-First Frontier Search and Structured Duplicate Detection) are orthogonal to our method. By introducing the abstract state space concept, the spatial locality of the states in External A* can be further improved. Also, duplicate detection using external hashing within each of our buckets might result in better run-time for our algorithm, in practice. To sum up, all three approaches contribute different insights to the problem of External Memory heuristic search.

3.7.3 Breadth-First Heuristic Search

Another related research area is the internal memory-restricted algorithms, that are mainly interested in an early removal of duplicates by keeping as little as possible of the *Closed*

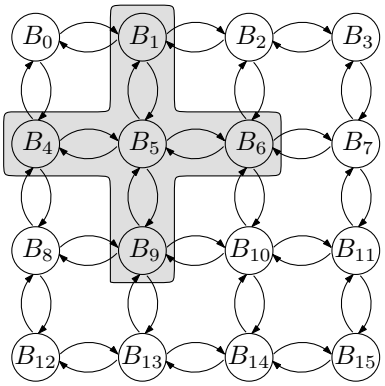

Figure 3.8: Abstract graph of 15-puzzle for Structured Duplicate Detection. All the states that map to one of the abstract states (shown in gray area) are required to be kept in the main memory. The rest of the states can be flushed to the hard disk.

list in RAM. The improved space-efficiency of a Breadth-First traversal ordering in heuristic search has led to improved internal memory algorithms such as *Breadth-First Heuristic Search* (BFHS) and *Iterative-Deepening Breadth-First Heuristic Search* (Zhou & Hansen 2006a).

BFHS is based on the observation that in a Best-First Search, the size of the frontier nodes – the *Open* list – can be much larger than the size of the frontier nodes in a Breadth-First Search. A more significant advantage is that the size of the *Closed* list can be reduced too by deleting some of the previous BFS-levels. For undirected graphs, it exploits the observation by Munagala & Ranade (1999) which states that only two previous BFS-levels are necessary to remove all the duplicates. For directed graphs, the authors utilize the concept of back-edges to bound the number of BFS-levels that have to be kept in the *Closed* list to guarantee that no state is expanded twice.

Heuristics are accumulated by pruning all the states s in a BFS-level g for which $g(s) + h(s)$ is greater than a provided upper bound on the solution length U. If the upper bound was very *optimistic* and no solution has been found, the whole search is *repeated* again with an increase value $U' > U$. This method is referred to as *Iterative-Deepening Breadth-First Heuristic Search* by Zhou & Hansen (2006a).

When comparing with BFHS, External A* performs better at three fronts. First of all, unlike BFHS, External A* does not need any upper-bound to avoid a complete Breadth-First Search. For many of the AI puzzle domains such an upper-bound is likely to be known in advance, but for irregular problems as they appear in model checking or planning, it is difficult to predict such a bound.

Secondly, incorporating such a bound in External A*, say *Iterative External A**, we can, in fact, get significantly lower number of I/Os when compared with multiple iterations of BFHS with increasing U. A bound U can be seen as a diagonal in the matrix-based exploration of External A*. All states u with $f(u) > U$ can then be discarded and not flushed to the disk – saving us large number of I/O operations. Moreover, if no solution has been found till U, and the search has to be restarted with a new $U' > U$, re-expanding the diagonal U is sufficient to guarantee completeness, as apposed to re-generating all the previous states in BFHS.

Thirdly, Iterative-Deepening BFHS inevitably has to be re-started from the beginning for every increase in the bound. All states that were generated in the last iteration have to be

regenerated and to undergo duplicate detection process. This disadvantage is due to the fact that the duplicate detection has to be performed on complete BFS levels. Contrarily, in External A*, the states are divided into (g, h) buckets that facilitate the structuring, and hence, limit the duplicate detection to only few buckets as opposed to whole BFS level.

Another technique to keep the number of I/Os lower in External A* is not to flush the states that do not fall into the current diagonal. If the current diagonal leads us to the goal, we have actually saved the I/Os needed to flush the next two diagonals. If the goal is not found on the current diagonal, we have to re-scan it again and flush only the nodes in the next diagonals, while those on the current diagonal can be discarded.

3.7.4 Real-valued weights and sparse graphs

BFHS offers two advantages on External A* when it comes to graphs with real-weights and spare graphs. Since the weight and heuristic functions are used by External A* for defining a suitable partition on the state space, graphs involving real-valued weights pose some challenges for External A*. BFHS, on the other hand, keeps states having the same g-values in the same bucket irrespective of their h-values or weights. Only when a layer g is completely expanded and its successors are inserted into the layer $g + 1$, BFHS shifts its attention to the duplicates removal and expansion of layer $g + 1$. If the cost of a path is a monotonic function (e.g., sum of the weight of the edges), optimality can be guaranteed through either of the two following minor extensions to BFHS. First, before expanding a layer g, it needs to be sorted according to the cost values of each state. Hence, as soon as a state $t \in \mathcal{T}$ is expanded, the search can be terminated. As sorting is an expensive operation, to guarantee the optimality once t is found, we need to scan the whole layer for any other target state that may have a better cost.

In case of sparse graphs, External A* might be required to flush buckets containing only few ($<$ B) states. Consequently, the actual number of I/O operations would be more than the worst-case I/O complexity of External A*. On the contrary, algorithms that are based on Breadth-First expansion order, such as Munagala and Ranade's BFS, Best-First Frontier Search with delayed duplicate detection, and BFHS are better equipped for sparse graphs. Note that these algorithms do not need to save each layer in a separate file, instead, several layers can be concatenated in one single file. Let layer g be one such layer that has less than B elements. Instead of flushing layer g to the disk, sorting and duplicates removal can be performed in the internal memory, resulting in the savings of several I/O operations involving less than B elements. This process can be continued on the subsequent layers until there are at least B elements that are ready to be flushed to the disk. However, for a controlled layer subtraction, we need to save, in internal memory, the offset of each layer on the file to avoid scanning of unnecessary layers.

3.8 Summary

In this chapter, we have proposed an External Memory algorithm for heuristic search in implicit, unweighted and undirected graphs. We have termed this algorithm as *External A**. One major challenge encountered while designing external memory heuristic search algorithms is the construction of an external priority queue that allows the selection of the *best* state from a state set. This state set could be several times larger than the available

RAM. By using an implicit priority queue in the form of a matrix of state sets, we have been successful in changing the search order to Best-First Search. This chapter also reports on the asymptotic I/O complexity of External A* and subsequently proves that, given consistent estimates in unweighted graphs, our algorithm achieves optimal I/O complexity. This is also the first report on I/O complexity of an EM algorithm designed for implicit graphs. Unlike other approaches for external heuristic search, External A* is oblivious to the state space structure and, as we will see in the subsequent chapters, is applicable to highly irregular model checking graphs. It also does not require any estimate on the depth of the solution. Extension of External A* to sparse graphs and real-valued edge weights is still an open problem.

External A* has been evaluated extensively on a large set of 15-Puzzle instances. We have successfully solved all the Korf's 100 instances while using only 1.2 Gigabytes of RAM. When compared with hard disk usage, we observe memory savings of up to 16 times. For an efficient implementation, we have also presented a *pipeline*-based approach, where the output of one intermediate phase is redirected to another, without performing any I/O operations.

This chapter serves as the basis of several other chapters in this dissertation, where External A* will be extended to directed and weighted graphs, and applied to the model checking of concurrent systems. In Chapter 7 a *distributed* variant of External A* will be presented, which utilizes a network of workstations, or multiple cores of a CPU, to distribute the internal workload.

Part I

Model Checking

Chapter 4

Introduction to Model Checking

The dependence of humans on software is rising at an astonishing rate. This dependence, in turn, increases the significance of 'correct' or 'verified' software systems. Though, the need for verification is crucial to *every* system, particularly in environments where a small error in the running system could have a direct or indirect impact on human lives, this need is *incalculable*.

The verification of software systems differs from the verification of physical systems. Holloway (1997) in his article '*Why engineers should consider formal methods*' writes:

> *"... in physical systems smooth changes in inputs usually produce smooth changes in outputs. That is, most physical systems are continuous. This allows the behavior of the system to be determined by testing only certain inputs, and using extrapolation and interpolation to determine the behaviors for untested inputs.*
>
> *Software systems are, by their very nature, discontinuous. A small change in input may change the outcomes at several decision points within the software, causing very different execution paths and major changes in output behavior. As a result, using extrapolation and interpolation to estimate output behaviors for untested inputs is risky at best, and exceedingly dangerous at worst."*

In the past, we have seen some very *expensive* software errors – both in terms of money and human life. Two of the most important ones are the destruction of the *Ariane-5* rocket (*cost: $500 Million*) and the belly landing of a Lufthansa flight in Poland (*cost: 3 human lives*).

Techniques like code testing by a quality assurance team cannot cover the whole range of possible errors in a running system. In *code testing*, an approach similar to the verification of mechanical systems is usually applied, where a set of representative tests is used to argue about the correctness of a system. For increased safety, a typical practice in critical systems, like in aircrafts, is to have multiple software systems implemented by different teams in completely different languages. All these systems are run simultaneously. A decision is then made by voting.

Model checking (Clarke, Grumberg, & Peled 1999; Müller-Olm, Schmidt, & Steffen 1999) is an exhaustive automated procedure to verify whether a system conforms to some required behavior or not. The success of model checking in dealing with the verification of large systems has made it an active area of research in computer science. Especially during the last two decades, it has evolved into one of the most successful verification techniques. Examples range from mainstream applications such as protocol validation, software and embedded

systems' verification to areas such as business work-flow analysis, and scheduler synthesis and verification.

Model checking relies on a formal modeling of the system. This formal model is then checked for the presence or absence of a specific behavior that we would like to be present or absent in the system. Such behaviors – or as they are usually referred to in the literature: specification properties – are mostly described by temporal logics. Temporal logics use modal temporal operators to capture the behavior of a system. In the following, we denote this model by $\mathcal{M}$, while a property specification is written as ϕ.

The task of model checking can be described as:

Definition 4.1 (Model Checking) *Given a model $\mathcal{M}$ of a system and a property specification ϕ,* model checking *verifies if the property ϕ is satisfied by the system or not, denoted $\mathcal{M} \models \phi$?*

Model Checking Paradigms: There are two different paradigms of model checking: a) *two-passes* model checking, where the whole state space of the system is explicitly generated and tested against ϕ; and b) *on-the-fly* model checking, where the state space is generated and checked against ϕ simultaneously. The latter is more efficient than the former one in most settings, due to its limited memory requirements: only relevant parts of the state space are generated.

In practice, the success of model checking is due to the on-the-fly model checking (Fernandez *et al.* 1992) to find errors/bugs and hence *falsify* a model. Phrasing it differently: instead of checking whether our model satisfies ϕ, we check if the model *does not* satisfy ϕ. Equivalently, we are now interested in finding out if $\mathcal{M} \models \neg\phi$. The falsification procedure mainly relies on some search or exploration strategy for finding an error trace in the model where ϕ is violated. This error trace can then be used by the engineers to localize the sources of the erroneous behavior and correct the system.

Similar to the other state spaces, model checking is also faced with the so-called *state space explosion* problem – for large models the search for errors is destined to failure due to huge space and time requirements. In the subsequent chapters, we will study different EM search algorithms to extend the scope of model checking in dealing with large and complex systems.

Structure of the chapter: We first discuss the formal modeling of complex systems along with a real-world example of elevated trains. An overview of Linear Temporal Logic (LTL) for defining specification properties is presented next. We then focus on automata-based model checking, where systems can be described by finite automata and the requirement specifications can be represented by a finite automata on *infinite* words. Then, we give an overview of directed model checking and present some heuristic functions that can be used to accelerate on-the-fly model checking.

4.1 Formal Modeling of Concurrent Systems

Concurrent systems can be modeled by several formalisms, such as communicating finite automata, Petri nets (Petri 1962), process algebras such as the calculus of communicating systems (CCS) (Milner 1980), communicating sequential processes (CSP) (Hoare 1978), algebra of communicating processes (ACP) (Bergstra & Klop 1984), etc. Building a model is

as much of an art as it is a science. In the next section we discuss a formalism to *represent* such a model and which is suited to model checking procedures. The formal notation used in this chapter is mostly built on the notation by Etessami (2002). At this point, it is suitable to discuss our example model checking problem that will be referred to several times in this document.

4.1.1 An Example Case Study – The H-Bahn Model

The word H-Bahn is an acronym for *Hänge-Bahn*, best translated from German as 'hanging train'. We consider a set of hanging cabins used as a public transport facility at the University of Dortmund to connect different campuses. As the name implies, the cabins are attached to inverted tracks raised several meters above the ground. In Figure 4.1, we see an illustration of the whole system.

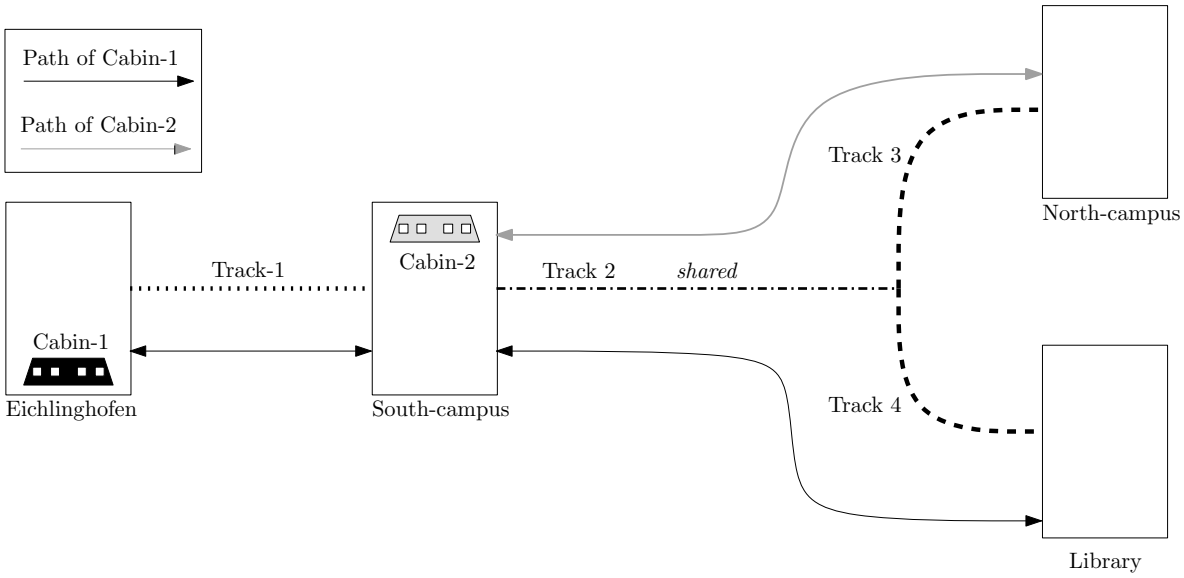

Figure 4.1: H-Bahn Model. The path of Cabin-1 is shown with black arrowed line, while the path of Cabin-2 is shown with gray arrowed line. Each track is shown with a different line style for clarity. Only Track-2 is shared between both the cabins.

There are two cabins (Cabin-1 & 2), and four tracks (Track-1,2,3 & 4). Cabins connect four stops where people can go in and out of the cabins. The stops are depicted by large rectangles and are named Eichlinghofen, North-campus, South-campus, and Library. Out of four tracks, only one of them, Track-2, is shared between the two cabins. The path of Cabin-2 is depicted with light gray arrows, while the path of Cabin-1 is shown with black arrows. As can be seen in the diagram, Cabin-2 travels between only two stops, while Cabin-1 has to transfer passengers between three stops. Each stop can accommodate two cabins at the same time. Track-1, 3 & 4 are of the same length, while Track-2 has twice the length of the others. Because of its length, it is required that Track-2 be shared among the cabins. We assume an asynchronous system where control alternates between the two cabins. We desire an automated control system that controls the cabins with a *feasible* utilization of the resources.

4.1.2 Variables, Guards, and Actions

A concurrent system is composed of two or more components. In order to communicate with each other, these components make extensive use of shared variables.

Definition 4.2 (Variable) *A variable is a place holder for an object from its domain. The set of variables in a system is denoted by* $Var = \{x_1, x_2, \ldots x_k\}$*, where each variable* x_i *can take its value from its finite domain* $D(x_i)$*.*

The finite domain restriction is a necessary condition for making the problem of model checking *decidable*; for infinite states system, readers are referred to (Burkart & Steffen 1997). We also assume a total order defined on the elements of the domain set for each variable. The variables can be combined by basic arithmetic operators: $+$ and $-$. Since the domains are finite, to avoid the underflow and overflow of values, we assume the operators as modulo operators. With γ, we denote a particular assignment to the set of variables Var. The set of all such assignments is denoted as Γ.

Definition 4.3 (Guard) *A guard* $: \Gamma \rightarrow \{\texttt{true}, \texttt{false}\}$ *is a function defined on the valuations and maps a valuation to either* `true` *or* `false`*. Guards can be formed by the following grammar.*

guard	$\Rightarrow$	**if** `true`	*holds unconditionally*
	|	**if** $\langle$Exp$\rangle$	*holds only if the boolean expression* $\langle$Exp$\rangle$ *evaluates to* `true`
	|	*guard* $\vee$ *guard*	*logical conjunction*
	|	*guard* $\wedge$ *guard*	*logical disjunction*
	|	$\neg$*guard*	*Negation*

Given that a *guard* on a transition evaluates to `true`, an action can be performed that changes the values of some of the variables.

Definition 4.4 (Action) *An action* $: \Gamma \rightarrow \Gamma$ *maps a given variable valuation to another valuation and is of the form:*

action	$\Rightarrow$	$x_i := \sum_{j=1}^{k} c_j \cdot x_j + c$	*Linear combination of variables and constants* c_i*s*
	|	*action; action*	*Chain of actions*

We will use '–' (dash) to denote an *empty action* that does not affect the values of the variables.

4.1.3 Extended Finite State Machines

Having defined the variables and guards, we are in a position to describe the Extended Finite State Machines or EFSM for short (Etessami 2002). An EFSM is a finite-state machine (FSM) with a set of *control* states and a set of transitions. Unlike an ordinary FSM, the transitions are *guarded* and are executable, if and only if, the guards defined on the transitions evaluate to `true`.

Definition 4.5 (Extended Finite State Machines (EFSM)) *An extended finite state machine is a 5-tuple,* $\mathcal{M} = \langle Q, Var, Guards, Actions, \Delta, \mathcal{I} \rangle$*, where*

- $Q = \{q_1, q_1, \ldots, q_m\}$ *is the set of control states,*
- $Var = \{x_1, x_2, \ldots, x_k\}$ *is the set of variables with finite domains* $D(x_1), D(x_2), \ldots, D(x_k)$*,*
- $Guards = \{guard_1, guard_2, \ldots, guard_g\}$ *is the set of guards,*

- *Actions* $= \{action_1, action_2, \ldots, action_l\}$ *is the set of actions,*
- $\Delta \subseteq Q \times$ Guards $\times$ (Actions) $\times Q$ *is the transition relation, and*
- $\mathcal{I} \subseteq Q \times \Gamma$ *is the set of initial states in the form of a pair formed by a control state* $q \in Q$ *and a valuation* $\gamma \in \Gamma$*.*

Graphical Representation of EFSM: An EFSM can be represented graphically as an automaton with the control states as the nodes and the transitions represented as directed edges between two nodes. The guards and actions are placed on the edges. For the sake of brevity, guards are placed on top of an edge while actions are placed below the edge.

Note that an EFSM assumes a set of initial states $\mathcal{I}$ instead of a single state as in ordinary finite-state machines. Moreover, the notion of control states is just for clarity and can, in fact, be completely replaced by a program counter pc variable. In the subsequent text, we will make heavy use of the pc variable to refer to the EFSM states. The notion of a state in an EFSM is a pair consisting of the control state and the valuation vector. For an EFSM state $s \in Q \times \Gamma$, we use the notation $s[v_i]$ to denote the value of the variable v_i in the state s. The control state q in s is accessible by $s.q$.

EFSM Model of the H-Bahn We model both cabins as separate automata. The communication is done through a set of global variables, which defines the occupancy of the tracks. The variable, *Track-<x>* is assigned a value of 'F' for `false` when *Track-<x>* is not occupied and can be used by any of the cabins. The EFSM model of the H-bahn example is shown in Figure 4.1.3. To model the sharing of the tracks, we have divided the shared track *Track-2* into two sub-tracks: *Track-2a* and *Track-2b*. Furthermore, we consider a discrete-time system where each transition takes the same time. The variables *forward1* and *forward2* are to avoid the cabins turning back before they have reached their end stops.

4.1.4 Composition of Concurrent Systems

The individual components in a concurrent system can be combined in either a *synchronous* or an *asynchronous* manner. In a synchronous composition all components have to take a transition.

Definition 4.6 (Synchronous Composition) *A synchronous composition of* n *EFSMs* $\mathcal{M}_i = \langle Q_i, Var_i, Guards_i, Actions_i, \Delta_i, \mathcal{I}_i \rangle$*, with* $1 \leq i \leq n$ *is an EFSM* $\mathcal{M} = \langle Q, Var, Guards, Actions, \Delta, \mathcal{I} \rangle$*, defined as:*

- $Q = Q_1 \times Q_2 \times \ldots \times Q_n$*,*
- $Var = Var_1 \cup Var_2 \ldots \cup Var_n$*,*
- $Actions = \bigcup_{i=1}^{n} Actions_i$*,*
- $\Delta = \{((q_1, q_2, \ldots, q_n), (guard_1, guard_2, \ldots, guard_n), (action_1, action_2, \ldots, action_n), (q'_1, q'_2, \ldots, q'_n))$*, s.t.* $\forall (1 \leq i \leq n) : (q_i, guard_i, action_i, q'_i) \in \Delta_i\}$*,*
- $\mathcal{I} = \mathcal{I}_1 \times \mathcal{I}_2 \times \ldots \times \mathcal{I}_n$*.*

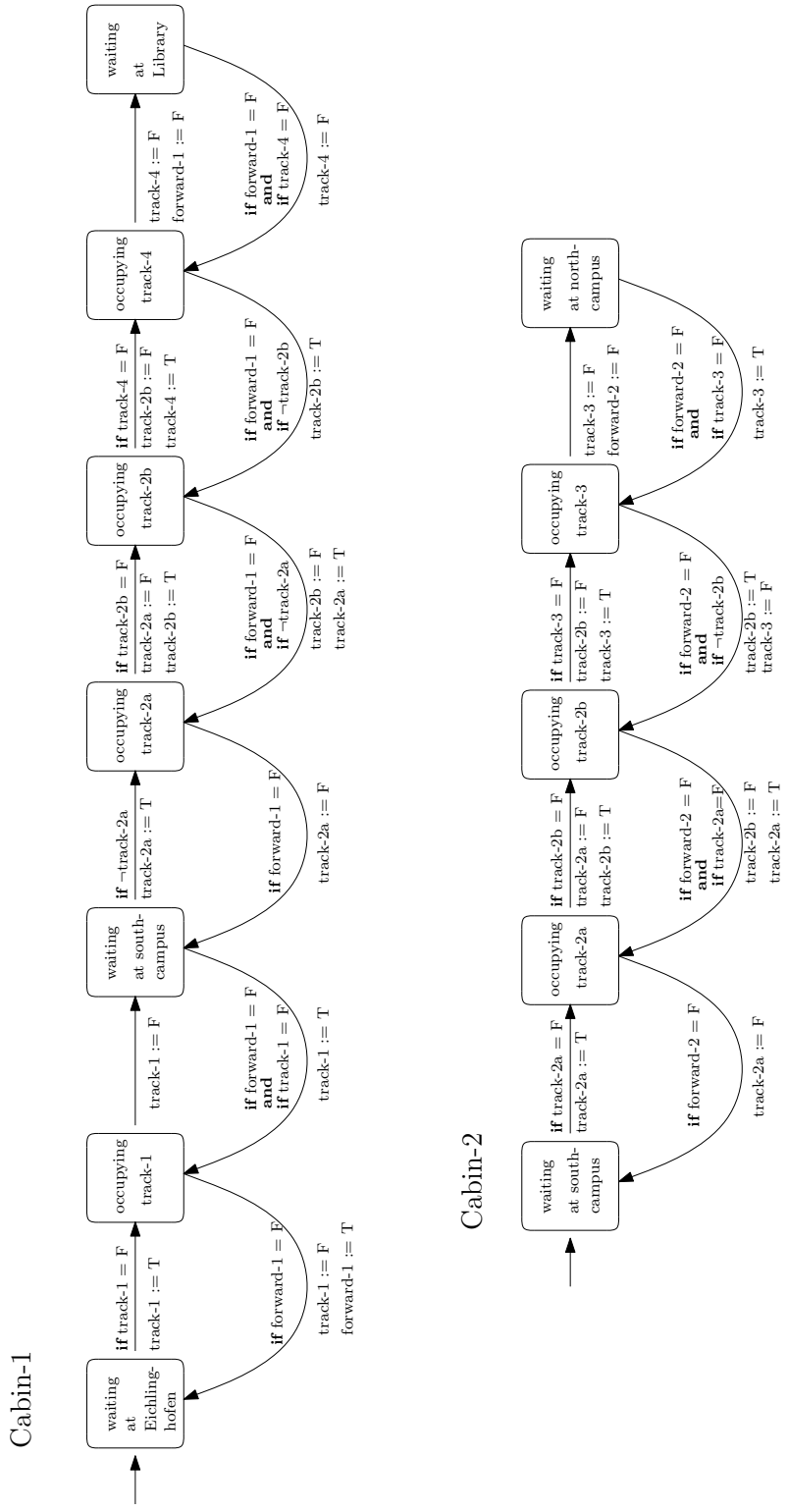

Figure 4.2: Extended Finite State Machine model of Cabin-1 and Cabin-2 from the H-Bahn model.

Communication systems, as they are used in computer networks, are mostly asynchronous. For each composite system transition there is just one transition in a component.

Definition 4.7 (Asynchronous Composition) *An asynchronous composition of n EFSMs $\mathcal{M}_i = \langle Q_i, Var_i, Guards_i, Actions_i, \Delta_i, \mathcal{I}_i \rangle$, with $1 \leq i \leq n$ is an EFSM $\mathcal{M} = \langle Q, Var, Guards, Actions, \Delta, \mathcal{I} \rangle$, defined as:*

- $Q = Q_1 \times Q_2 \times \ldots \times Q_n$,
- $Var = Var_1 \cup Var_2 \ldots \cup Var_n$,
- $Actions = \bigcup_{i=1}^{n} Actions_i \cup \{-\}$,
- $\Delta = \{((q_1, q_2, \ldots, q_n), (guard_1, guard_2, \ldots, guard_n), (action_1, action_2, \ldots, action_n), (q_1', q_2', \ldots, q_n'))$, *s.t.* $\exists (1 \leq j \leq n) : (q_j, guard_j, action_j, q_j') \in \Delta_j$ *and* $\forall (1 \leq i \leq n, i \neq j) : q_i = q_i' \wedge action_i = `-'\}$,
- $\mathcal{I} = \mathcal{I}_1 \times \mathcal{I}_2 \times \ldots \times \mathcal{I}_n$.

In this dissertation, we mainly deal with concurrent systems that result in an asynchronous composition of two or more sub-systems. Such a requirement practically does not restrict the applicability of the algorithms, as many software systems are asynchronous and hence not connected to a global clock, e.g., network protocols do not require all the connected nodes to be synchronized with an external clock.

4.1.5 State-Transition Systems

An EFSM can be regarded as a system description. The underlying behavior of an EFSM is captured by a state-transition system. This state-transition system can be obtained by *unfolding* an EFSM, where each *state* now consists of a valuation of the variables and the control state of the EFSM. A *transition* is any legal path between two control states of the EFSM.

Definition 4.8 (State-Transition System) *The state-transition system of an EFSM $\mathcal{M}$ is a triple $ST(\mathcal{M}) = \langle \mathcal{S}, \mathcal{R}, \mathcal{I} \rangle$, where*

- $\mathcal{S} \subseteq Q \times \Gamma$ *is a set of states,*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ *is the transition relation and is obtained from the transition relation Δ in the EFSM as $(s, s') \in \mathcal{R} \iff \exists (q, \gamma) \in \mathcal{S} \wedge (q', \gamma') \in \mathcal{S}$ such that $\exists (q, guard, action, q') \in \Delta$ and $guard(\gamma) =$ `true` and $action(\gamma) = \gamma'$, and*
- $\mathcal{I} \subseteq \mathcal{S}$ *is the set of initial states defined exactly as in the EFSM.*

Given that all variables take values from finite domains, the maximum number of states in a state-transition system can be bounded by

$$|\mathcal{S}| = O(|Q| \times |\Gamma|) = O(|Q| \times |D(x_1)| \times |D(x_2)| \times \ldots \times |D(x_k)|)$$

As many of the valuation vectors may not be reachable, the actual reachable state space might be much smaller.

4.1.6 Kripke Structures

In model checking, one is interested in a state or a path through the state-transition system that satisfies some particular property. To define such a property we make use of some basic predicates that we refer to as *Atomic propositions*. They represent observations of the state of the system.

Definition 4.9 (Atomic Propositions) *Atomic propositions* $AP = \{p_1, p_2, \ldots, p_k\}$ *are functions that when applied to a state evaluate to either* `true` *or* `false`.

The formalism that supports the definition of atomic propositions to be applied to state-transition systems is a *Kripke structure*, first presented by Saul Kripke in early 1960's. Kripke structures use a labeling function to assign a subset of atomic propositions to each state.

Definition 4.10 (Kripke Structures) *A Kripke structure of a model* $\mathcal{M}$ *extends the state-transition system* $ST(\mathcal{M}) = \langle \mathcal{S}, \mathcal{R}, \mathcal{I} \rangle$ *as a 5-tuple* $\mathcal{K}(\mathcal{M}) = \langle \mathcal{S}, AP, \sigma, \mathcal{R}, \mathcal{I} \rangle$, *where*

- $\mathcal{S}$ *(set of states),* $\mathcal{R}$ *(set of transitions), and* $\mathcal{I}$ *(set of initial states) are the same as in* $ST(\mathcal{M})$, *additionally,*
- AP *is the set of atomic propositions, and*
- $\sigma : \mathcal{S} \rightarrow 2^{AP}$ *is the labeling function.*

An example of a Kripke structure from of our H-bahn example is provided in Figure 4.1.6. It shows an execution path that leads to a state where both cabins occupy the opposite parts of Track-2 and end up in a deadlock state.

In fact, there is a vast variety of modeling formalisms that one finds in the model checking literature. Some widely used ones are: Labeled automata (Bérard *et al.* 2001), Kripke transition systems (Müller-Olm, Schmidt, & Steffen 1999), etc. These formalisms mainly differ from each other by an additional labeling function that assigns labels to transitions. While discussing techniques that reason on the actual *transition*, like partial order reduction (Godefroid 1991) or data-flow analysis (Steffen 1991), these formalisms provide the necessary foundations.

Execution: An execution or a path in a Kripke structure is an sequence of states denoted as $\pi := s_0, s_1, s_2, \ldots$, where $s_i \in \mathcal{S}$, $s_0 \in \mathcal{I}$, and $(s_i, s_{i+1}) \in \mathcal{R}$ for all $i \geq 0$. The execution itself can either be infinite or finite. Note that, in our case, although Kripke structures are finite, the executions can still be infinite.

4.2 Linear Temporal Logic for Specification Properties

Temporal logics are used to describe and reason on the order of events in time. Originally, temporal logics were developed by philosophers to use the notion of time in natural language arguments. These logics do not explicitly define time, but instead utilize the relative ordering of events for reasoning. The use of temporal logics for specifying a software/hardware system's behavior is due to Pnueli (1977). Temporal logics are particularly useful when it comes to defining properties of reactive systems, which are often designed for *infinite* behavior. For example, a dispensing machine, an operating system kernel, etc.

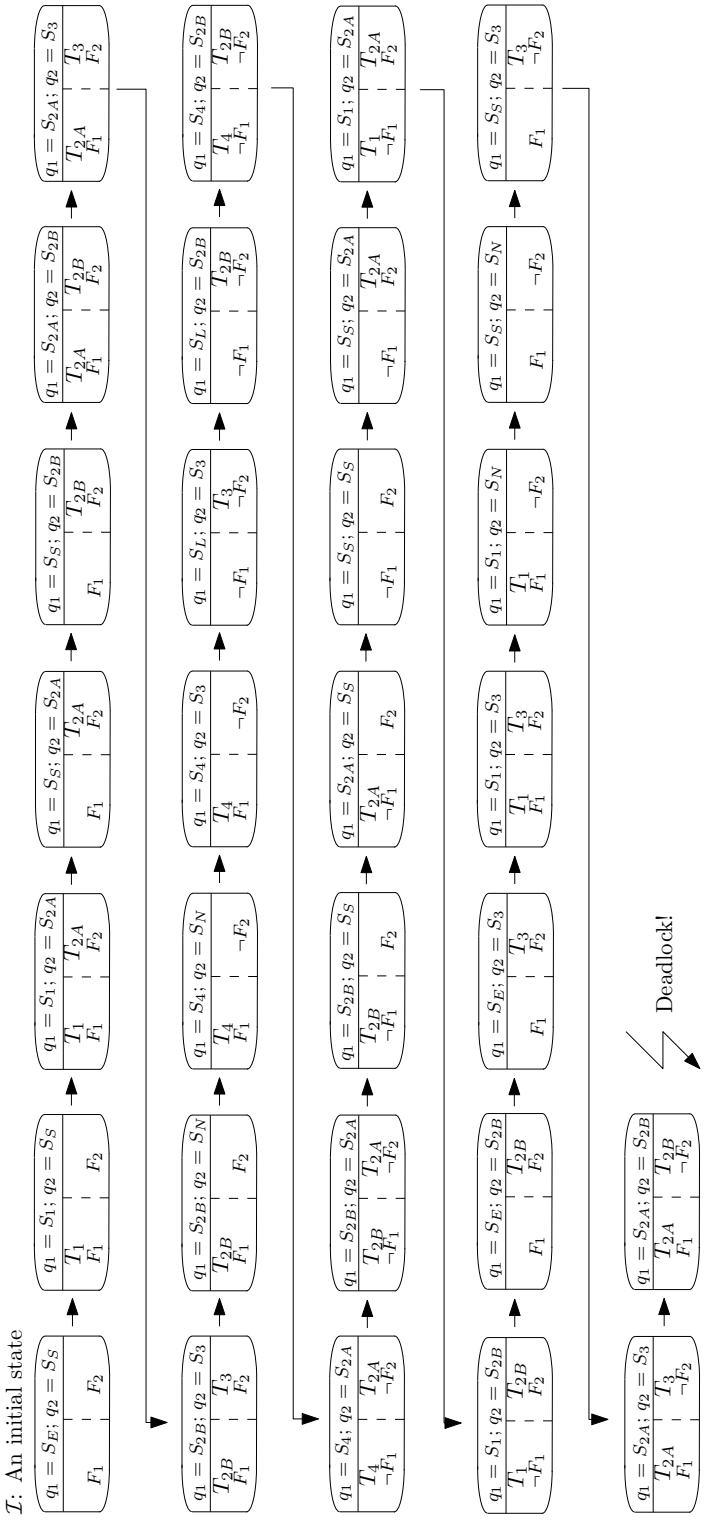

Figure 4.3: Kripke structure of an error path in the H-Bahn model. The path leads to a deadlock state where Cabin-1 tries to acquire Track-2B and Cabin-2 Track-2A simultaneously while going in opposite directions. q_i represents the control state in the EFSM of Cabin-i.

Temporal logics are categorized as either *branching time logics* or *linear time logics*. A linear time logic views time as infinitely increasing in a straight line and places events on that line. Linear Temporal Logic (LTL) is an example of one such logic. On the other hand, a branching time logic assumes several time lines in parallel. Computational Tree Logic (CTL) is a branching time logic and clearly distinguishes itself from LTL, based on the expressive power of temporal statements.

An LTL formula either holds on all paths starting from a state or it does not hold at all. LTL formulae have the form 'Always ϕ', where ϕ is an *LTL formula* constructed from a combination of logical connectives and modal operators. LTL does not provide support for any path quantification such as '$\forall$' and '$\exists$' – a feature provided by CTL. Both CTL and LTL are combined in a more expressive logic called CTL*, which, in turn, is less expressive than the more general μ-calculus. The readers are referred to (Clarke, Grumberg, & Peled 1999) for a thorough exposition of different temporal logics.

Syntax of LTL: If p is an atomic proposition then p is a path formula. If ϕ and ψ are path formulae so are $\neg\phi, \phi \vee \psi, \phi \Rightarrow \psi, \phi \wedge \psi$, $\mathbf{X}\phi$ (next), $\mathbf{F}\phi$ (eventually/finally), $\mathbf{G}\phi$ (globally), ψ **U** ϕ (until), and ψ **R** ϕ (release).

Semantics of LTL Let σ be the labeling function of the Kripke structure that assigns atomic propositions from the set *AP* to the states of the model satisfying them. Moreover, let π be an infinite execution in the Kripke structure and π^i denotes the suffix of π rooted at s_i. The semantics of the well-formed formulae expressible in LTL can be defined as follows:

1.	$\mathcal{M} \models \phi$	$\Leftrightarrow$	$\forall\pi \in \mathcal{K}(\mathcal{M}), \pi \models \phi$	ϕ is satisfied in all the executions
2.	$\pi \models p$	$\Leftrightarrow$	$p \in \sigma(s_0)$	p holds in the first state of π
3.	$\pi \models \neg\phi$	$\Leftrightarrow$	$\pi \not\models \phi$	ϕ does not hold in π
4.	$\pi \models \phi \vee \psi$	$\Leftrightarrow$	$(\pi \models \phi) \vee (\pi \models \psi)$	Either ϕ or ψ or both hold in π
5.	$\pi \models \phi \wedge \psi$	$\Leftrightarrow$	$(\pi \models \phi) \wedge (\pi \models \psi)$	Conjunction
6.	$\pi \models \mathbf{X}\phi$	$\Leftrightarrow$	$\pi^1 \models \phi$	ϕ holds in the i-th suffix of π
7.	$\pi \models \mathbf{F}\phi$	$\Leftrightarrow$	$\exists k \geq 0$, such that $\pi^k \models \phi$	Eventually ϕ will hold
8.	$\pi \models \mathbf{G}\phi$	$\Leftrightarrow$	$\forall k \geq 0, \pi^k \models \phi$	ϕ holds globally
9.	$\pi \models \psi\mathbf{U}\phi$	$\Leftrightarrow$	$\exists k \geq 0$, such that $\forall 0 \leq i < k, \pi^i \models \phi$, and $\pi^k \models \psi$	ψ will hold, until then ϕ holds
10.	$\pi \models \psi\mathbf{R}\phi$	$\Leftrightarrow$	$\forall i, \pi^i \models \phi$ or $\exists k\ \pi^k \models \psi$ and $\forall i \leq j, \pi^j \models \phi$	ϕ holds everywhere, or if ψ holds somewhere then ϕ must always holds at the prefix path

The release **R** operator is a dual of **U** (until) and is equivalent to $\neg(\neg\phi\mathbf{U}\neg\psi)$. Some authors have also used another modal operator: **W** (Weak-until). It is defined similarly to the **U** (until) accept that the ϕ condition does not have to hold necessarily. In Figure 4.4, the interpretation of different LTL formulas can be seen. Different paths $\pi_1, \pi_2, \pi_3, \pi_4$ are shown along with the LTL formula they satisfy.

In model checking, one is mostly interested in checking for three classes of properties: safety properties, liveness properties and deadlock freedom. Such a distinction allows us to use different efficient algorithms specifically designed to verify a particular class.

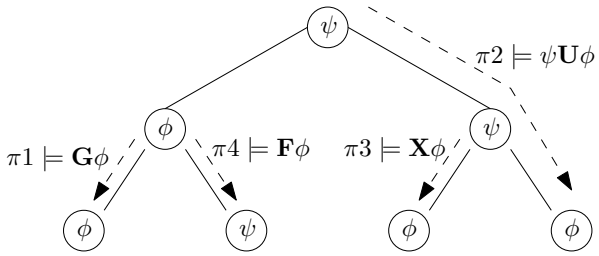

Figure 4.4: Paths satisfying different LTL properties.

4.2.1 Safety Properties

Safety properties state that in a system "*nothing bad will happen*". A typical example of a safety property is checking for mutual exclusion: no two processes enter a critical section at the same time. Bérard *et al.* (2001) characterize a property ϕ as a safety property, if the satisfaction of ϕ in an execution π guarantees that any *prefix* of π also satisfies ϕ. Hence, it is sufficient to check for safety properties on finite executions only, since if they do not hold on finite prefixes of the infinite execution, then they also do not hold on the infinite execution. A witness for the violation of a safety property is an execution that ends in a state where the property is violated. Checking for safety errors constitutes a major portion of model checking usage in practice. In the following we present some of the typical specifications that can be modeled as safety properties.

Invariants: Invariants define the properties that a system must satisfy throughout its life time. They can be written in LTL as $\mathbf{G}p$, interpreted as: *p must hold globally in all the reachable states*. In the example of h-bahn, an invariant can be defined as:

"it should never happen that Track-2a is shared by both cabins at the same time".

Given that pc_i denotes the control state in the EFSM of cabin i, the LTL equivalent of this invariant can be written as

$$\mathbf{G}\ (\neg(pc_1 = \textit{Track-2a}\ \wedge pc_2 = \textit{Track-2a})).$$

Assertions: Assertions describe a particular property that should be satisfied by a given type of states. An example assertion property in our h-bahn model is:

"whenever Cabin-2 reaches the Library stop, Track-4 should be freed".

The LTL formula can be given as:

$$\mathbf{G}\ (pc_2 = \textit{At-Library} \wedge \neg \textit{Track-4}).$$

4.2.2 Liveness Properties

Liveness properties state that in a system "*something good will eventually happen*". A typical liveness property is *program termination*: given certain inputs, the program will eventually terminate. Finding a finite witness for the violation of a liveness property is not sufficient –

it is possible that by extending the execution to an infinite one, the property will eventually be satisfied. In our H-Bahn example, we might require that:

> *"whenever Cabin-2 reaches the Library stop, it should arrive, at some time in the future, at the Eichlinghofen stop"*.

This property can be written in LTL as:

$$\mathbf{G}\ (pc_2 = \textit{At-Library} \rightarrow \mathbf{F}(pc_2 = \textit{At-Eichlinghofen})).$$

The example property is a typical *response* property where one wants some event to happen in response to another event. In communication protocols, it models the acknowledgments of messages: whenever a message is sent, eventually an acknowledgment will be received.

4.2.3 Deadlock Freedom

A system is said to be in a deadlock state, if no transition is possible in any of the processes. The deadlock in our h-bahn model appears when neither of the cabins can move forward. This is possible, when, while going in opposite directions, each cabin acquires the control over one part of the Track-2 and wants to acquire the next part. A deadlock can also be defined as an invariant property, though the transformation is not always straightforward. For example the deadlock in the h-bahn can be described as:

$$\mathbf{G}\neg\left(\textit{Track-2b} \wedge \textit{Track-2a} \wedge (\neg\textit{Forward-1} \wedge \textit{Forward-2}) \vee (\textit{Forward-1} \wedge \neg\textit{Forward-2})\right).$$

A solution to this deadlock problem in our case, would be to always acquire a full exclusive access to both tracks at the same time. It can be implemented by changing the guards at *At Eichlinghofen* and after *At-Track-4* and *At-Track-3*.

4.3 Automata-based LTL Model Checking

Automata-based LTL model checking (Vardi & Wolper 1986) is one of the most successful model checking procedures in practice. It is based on transforming both the model $\mathcal{M}$ and the specification property ϕ into a special kind of automaton called *Büchi automaton*. The special acceptance condition of Büchi automaton, is then used to establish the correctness of ϕ in $\mathcal{M}$. Büchi automata are finite-state automata defined over infinite words. Since reactive systems are usually run forever, Büchi automata provide a concise mathematical formalism to capture their behaviors. Formally, a Büchi automaton can be described as follows:

Definition 4.11 (Büchi Automaton) *A Büchi Automaton is a 6-Tuple* $\mathcal{B} = \langle \mathcal{S}_\mathcal{B}, \mathcal{I}_B, AP, \Sigma, \mathcal{R}_\mathcal{B}, \mathcal{T} \rangle$*, where*

- $\mathcal{S}_\mathcal{B}$ *is a set of states,*
- $\mathcal{I}_B \in \mathcal{S}_\mathcal{B}$ *is the set of initial states,*
- AP *is a set of atomic propositions,*
- $\Sigma = 2^{AP}$ *is a set of transition labels,*

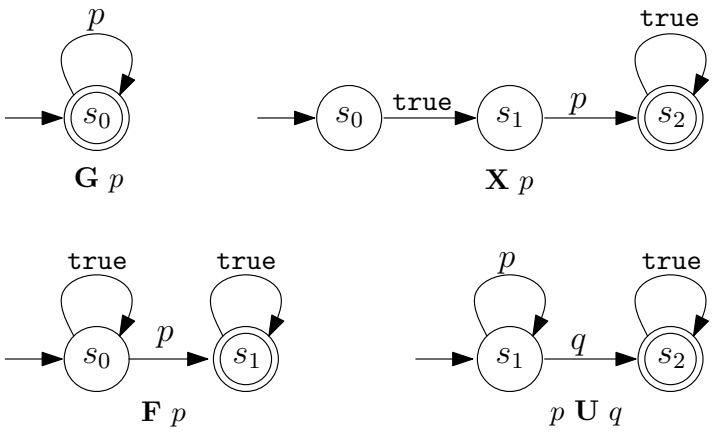

Figure 4.5: Büchi automata of basic LTL formulas

- $\mathcal{R}_\mathcal{B} \subseteq \mathcal{S}_\mathcal{B} \times \Sigma \times 2^{\mathcal{S}_\mathcal{B}}$ *is a set of transitions, and*
- $\mathcal{T} \subseteq \mathcal{S}_\mathcal{B}$ *is a set of accepting states.*

The acceptance condition of a Büchi automaton is defined for ω-words (infinite words) belonging to an ω-regular language. Let ρ be an infinite word. A run of ρ on a finite-state Büchi automaton would visit some of the states only finitely many times and some infinitely often. Let ρ^ω denote the set of states that appears infinitely often in ρ. We can then define Büchi acceptance as follows.

Definition 4.12 (Büchi Acceptance) *Given an ω-word ρ, a Büchi automaton $\mathcal{B}$ accepts ρ, if and only if, $\rho^\omega \cap \mathcal{T} \neq \emptyset$*

Given an LTL property ϕ, an equivalent non-deterministic Büchi automaton $\mathcal{B}(\phi)$ can be constructed that has at most $O(2^{|\phi|})$ states. The converse is not always possible, since Büchi automata are more expressive than LTL expressions (Wolper 1983).

4.3.1 Product of the Büchi Automaton and the Model

The second step is to transform a Kripke structure $\mathcal{K}(\mathcal{M})$ in to an equivalent Büchi automaton $\mathcal{B}(\mathcal{M})$. Recall that a Kripke structure has a labeling function that is defined on the states. In an automaton, transitions, rather than states, are labeled. Hence a Büchi automaton $\mathcal{B}(\mathcal{M})$ of the model $\mathcal{M}$ can be defined as:

Definition 4.13 (Büchi Automaton of the Model) *A Büchi automaton of a Kripke structure $\mathcal{K}(\mathcal{M})$ = $\langle \mathcal{S}, AP, \sigma, \mathcal{R}, \mathcal{I} \rangle$, representing a model $\mathcal{M}$, is a 6-tuple $\mathcal{B}(\mathcal{M}) = \langle \mathcal{S}, AP, \Sigma, \mathcal{R}, \mathcal{I}, \mathcal{T} \rangle$, where*

- $\mathcal{S}$ *(the set of states), AP (atomic propositions), and $\mathcal{I}$ (the set of initial states) are defined as in $\mathcal{K}(\mathcal{M})$, additionally,*
- $\Sigma = 2^{AP}$ *is a set of transition labels,*
- $\mathcal{R} \subseteq \mathcal{S} \times \Sigma \times \mathcal{S}$ *is a set of transitions, such that, $(s, p, s') \in \mathcal{R}$, iff, $p \in \sigma(s')$ and $(s, s') \in \mathcal{R}$ and*

- $\mathcal{T} = \mathcal{S}$ *is a set of accepting states.*

Let $Lang(\mathcal{B}(\mathcal{M}))$ be the ω-language of the Büchi automaton of the model, i.e., all possible runs, and $Lang(\mathcal{B}(\phi))$ the ω-language of the Büchi automaton representing the property ϕ. Then, the model satisfies the property, written as $\mathcal{M} \models \phi$, if and only if, the language accepted by the model is included in that of the specification, i.e.,

$$\mathcal{M} \models \phi \quad \textit{iff} \quad Lang(\mathcal{B}(\mathcal{M})) \subseteq Lang(\mathcal{B}(\phi)) \tag{4.1}$$

This question can be reformulated as: the model satisfies the property ϕ, if no run of the complemented language of $\mathcal{B}(\phi)$ is an accepting run in the model, i.e.,

$$Lang(\mathcal{B}(\mathcal{M})) \subseteq Lang(\mathcal{B}(\phi)) \quad \textit{iff} \quad Lang(\mathcal{B}(\mathcal{M})) \cap \overline{Lang(\mathcal{B}(\phi))} = \emptyset \tag{4.2}$$

It is possible to complement a Büchi automaton equivalent to an LTL formula, *but* the worst-case running time of such a construction is double-exponential in the size of the formula. The alternative approach that preserves the languages is to construct a *complemented* Büchi automaton from the negation of an LTL property, i.e., of $\neg\phi$ instead of ϕ. We can now reformulate the LTL model checking problem as:

$$Lang(\mathcal{B}(\mathcal{M})) \cap \overline{Lang(\mathcal{B}(\phi))} = \emptyset \quad \textit{iff} \quad Lang(\mathcal{B}(\mathcal{M})) \cap Lang(\mathcal{B}(\neg\phi)) = \emptyset \tag{4.3}$$

The language intersection can be checked by an intersection of the Büchi automaton of the model $\mathcal{B}(\mathcal{M})$ and the Büchi automaton of the negated property $\mathcal{B}(\neg\phi)$. Such an intersection is possible because Büchi automata are closed under intersection. Recall that a Büchi automaton accepts the infinite words that visit accepting states infinitely often. Hence, the language $Lang(\mathcal{B}(\mathcal{M}) \cap \mathcal{B}(\neg\phi))$ corresponds to the accepting cycles that are *legal cycles* in the model and where the property specification ϕ *does not hold*. Consequently, we can say that a model $\mathcal{M}$ satisfies a specification property ϕ if the language of the intersected Büchi automaton is empty, i.e.,

$$\mathcal{M} \models \phi \quad \textit{iff} \quad Lang(\mathcal{B}(\mathcal{M}) \cap \mathcal{B}(\neg\phi)) = \emptyset \tag{4.4}$$

Figure 4.6 illustrates the automata-based model checking procedure. A model $\mathcal{M}$ and a property ϕ are fed into a model checker. The property ϕ is first converted into its negated form and then transformed into a Büchi automaton. Both $\mathcal{B}(\mathcal{M})$ and $\mathcal{B}(\neg\phi)$ are then intersected to find any common word that is accepted by both of them. The existence of such a word proves that there are some *undesired* or *bad* behaviors in the model. The word itself gives us an *error trace* representing the erroneous path in the model where ϕ is not satisfied.

4.3.2 Reduction of Model Checking Problem to Search in State Spaces

Both LTL safety and liveness checking problems can be transformed into state space exploration problems. Recall that liveness requires that "something good will eventually happen". Liveness checking problems can be reduced to the checking for an *accepting cycle* that satisfies the Büchi acceptance condition. In other words, we check for the presence of a word of the form uv^ω, where u and v are finite words.

Definition 4.14 (Liveness Checking) *A model* $\mathcal{M}$ does not *satisfy an LTL safety property* ϕ*, if*

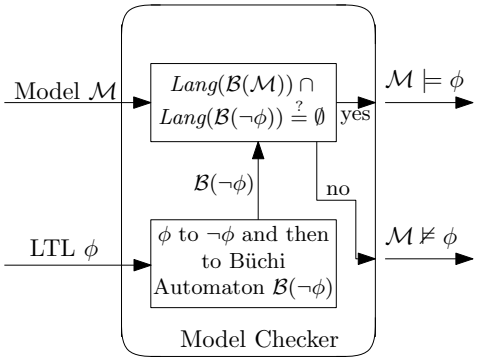

Figure 4.6: LTL Model Checking

and only if, there exists a lasso-shaped path

$$\pi := s_1 \in \mathcal{I}, s_2, \ldots, s_l, s_{l+1}, \ldots, s_m, \ldots, s_l$$

such that $s_m \in \mathcal{T}$ *in the product automaton of the model and the negated LTL property.*

On the other hand, safety checking require that "nothing bad will never happen", which means that the state space traversal for a falsification can be stopped once "something bad happens". Consequently, safety checking can be reduced to a simple path search problem requiring a path from $\mathcal{I}$ to any state in $\mathcal{T}$. Another important observation is that since we are no more interested in infinite paths while checking for safety properties, the acceptance condition for Büchi automaton is also not required. In other words, safety checking can be done directly on the Kripke structure of model $\mathcal{M}$ or, at most by, synchronizing it with a finite automaton.

Definition 4.15 (Safety Checking) *A model* $\mathcal{M}$ does not *satisfy a LTL safety property* ϕ*, if and only if, there exists a path*

$$\pi := s_1 \in \mathcal{I}, s_2, s_3, \ldots, s_m \in \mathcal{T}$$

in the product automaton of the model and the negated LTL property.

4.3.3 Space and Time Complexity

The time complexity of LTL model checking is $|\mathcal{K}(\mathcal{M})| \times 2^{|\phi|}$, where the second term corresponds to the size of the Büchi automaton obtained through the conversion from an LTL formula ϕ. Hence, we can say that LTL model checking is exponential in the number of temporal operators used in the LTL formula and linear in the size of the model. It belongs to the class co-NP-Hard as the Hamiltonian path problem can be reduced to the complement of LTL model checking problem (Clarke, Grumberg, & Peled 1999).

4.4 On-The-Fly LTL Model Checking

On-the-fly model checking (Courcoubetis *et al.* 1992; Fernandez *et al.* 1992) is an efficient way of performing model checking. Thus we can avoid the construction of the complete synchronous product of the model and the specification. The rationale behind on-the-fly model

checking is that, while searching for an error (or falsification), it is sufficient to generate the part of the state space that contains the error state.

4.4.1 Nested Depth-First Search

For checking the synchronous product graph of the model and the specification for accepting cycles on-the-fly, *Nested Depth-First Search* has been proposed (Holzmann, Peled, & Yannakakis 1996). Nested DFS extends the Double DFS algorithm by (Courcoubetis *et al.* 1992). Double DFS collects all the reachable accepting states through a DFS and keeps them in a queue in the order in which they were visited first. The second DFS expands each of the final states one-by-one and tries to reach the same states again. The reachability of a final state from itself guarantees the presence of an accepting cycle.

Nested DFS also explores the state space in a depth-first manner. The visited states are stored in a visited list and the states which reside on the current search stack are *marked*. It then invokes a secondary DFS starting at accepting states after they have been fully explored in the primary DFS and when the primary DFS backtracks to an accepting state. The secondary DFS explores states already visited by the primary search but not by any secondary search; states visited by the secondary search are *flagged* and if a state is found on the stack of the first search, an accepting cycle is found. Typical implementations use 2 bits per state: one for marking and one for flagging.

Complexity The complexity of Nested DFS is $O(|\mathcal{S}| + |\mathcal{R}|)$, in a state space with $|\mathcal{S}|$ states and $|\mathcal{R}|$ transitions. In the worst case, the algorithm would have to look at the whole state space – a case that can happen if no accepting cycle exists in the product automaton.

4.4.2 Property-Driven Nested Depth-First Search

Nested Depth-First Search starts a secondary search for every accepting state. An improvement has been suggested (independently) by Lluch Lafuente (2002) and Barnat, Brim, & Černá (2002) (the latter one was suggested in the context of distributed model checking). The main observation is that a cycle in the product automaton is accepting if and only if the corresponding cycle in $\mathcal{B}(\neg\phi)$ is accepting. Therefore, these approaches use Tarjan's algorithm to analyze $\mathcal{B}(\neg\phi)$ and to find out all the strongly connected components (SCCs). An SCC in $\mathcal{B}(\neg\phi)$ is called *non-accepting*, if none of its states are accepting; *fully-accepting*, if each cycle formed by states of the SCC is accepting, and *partially-accepting* otherwise. Improved nested DFS partitions the negated property automaton into SCCs and applies secondary search only in case of partially accepting cycles. In case of arriving at a *fully-accepting* SCC, the accepting cycle detection problem is reduced to a cycle detection problem. For the case of *non-accepting* SCC, we can be sure that no accepting cycle exists. In Algorithm 4.1, we see the pseudo-code of the algorithm. For every new state encountered during the search, its associated SCC is tested. The algorithm INDFS-LASSO is invoked for the secondary search (cf. Algorithm 4.2), when a state that belongs to a partially accepting SCC is visited. The variable *flagged* is to avoid restarting the algorithm more than once on the same state. The check in line 6 restricts the scope of the INDFS-LASSO to only one SCC. This particular feature of the algorithm makes it highly suitable for a distributed version, where different SCCs can be assigned to different processes.

Algorithm 4.1 Improved Nested Depth-First Search: INDFS

Input: s: a state.

Output: A lasso-shaped counter example from s that visits at least a state $s \in \mathcal{T}_{\mathcal{B}}$, if one exists, $\emptyset$ otherwise.

```
1: push(Open, s)      //Open IS IMPLEMENTED AS A STACK
2: Closed ← Closed ∪ {s}
3: if s ∈ Open and full-accepting(SCC(s)) then
4:    return Construct-Solution(s)      //s IS VISITED AGAIN AND IT BELONGS TO A FULL-ACCEPTING
      SCC OF B(¬φ)
5: for all successor s' ∈ Succ(s) do
6:    if s' ∉ Closed then
7:       π ← INDFS(s')      //RECURSIVELY START A NEW DFS FOR S'
8:       if π ≠ ∅ return π
9: end for
10: if accepting(s) and partial-accepting(SCC(s)) then
11:    π ← INDFS-LASSO(s)
12:    if π ≠ ∅ return π
13: pop(Open, s)
14: return ∅
```

Algorithm 4.2 Sub-procedure of INDFS to search for the lasso: INDFS-LASSO

Input: s: a state.

```
1: flagged(s) ← true
2: for all successor s' ∈ Succ(s) do
3:    if s' ∈ Open then
4:       return Construct-Solution(s')
5:    if not flagged(s) and SCC(s) = SCC(s') then
6:       π ← INDFS-LASSO(s')
7:       if π ≠ ∅ return π
8: end for
9: return ∅
```

SCCs of type *partially-accepting* are rare in practice. Černá & Pelánek (2003b) argued that based on their study of property specification patterns by Dwyer, Avrunin, & Corbett (1999), around 54% of the specification properties that appear in practice, can be transformed into Büchi automata whose SCCs can only be divided into *fully-accepting* and *non-accepting* SCCs. Such automata are referred to as *weak* automata.

4.4.3 LTL Model Checking with Breadth-First Search

One line of research aims at avoiding Nested DFS by studying variants of Breadth-First Search (Barnat, Brim, & Chaloupka 2005; Barnat, Brim, & Chaloupka 2003; Brim *et al.* 2004). The approach presented in (Barnat, Brim, & Chaloupka 2005; Barnat, Brim, & Chaloupka 2003) invokes a secondary search for detecting cycles from BFS *backward edges*, i.e., transitions encountered in the overall state space that link states in larger, together with (already explored) states in smaller depth. Those backward edges may potentially spawn cycles and

are searched individually. If no accepting cycle is found, the depth bound is increased. The number of backward edges is reduced by similar observations as for improved Nested DFS. The worst case time complexity is $O(|\mathcal{R}| \cdot (|\mathcal{S}| + |\mathcal{R}|))$. The approach allows *on-the-fly* model checking and is compatible with a limited form of partial order reduction. In (Brim *et al.* 2004), instead of backward edges, predecessor acceptance is chosen for an $O(|\mathcal{R}|^2 + |\mathcal{S}|)$ algorithm.

4.5 Other Approaches for LTL Model Checking

On-the-fly model checking is not the only way to check for language emptiness. Other research groups have worked on alternative methods of first generating the whole state space and then filtering out the accepting cycles in it. In the following we see two such approaches.

4.5.1 Tarjan's Algorithm

Analyzing the structure of the state space graph can help in localizing the accepting cycles. More precisely, if the strongly connected components can be isolated in a graph, then the search for accepting cycles in the whole state space can be reduced to the search for accepting cycles in the strongly connected components. An algorithm to compute all such components of a graph in linear time is Tarjan's algorithm (Tarjan 1972). The algorithm is divided into four stages. In the first stage, a DFS starting at the initial state computes the discovery and finishing times $t_d(u)$ and $t_f(u)$ for each visited state u, which corresponds to the time, when node u is entered and left. The second stage computes the inverse of the graph. In the third stage, a series of DFSs considers the nodes in order of decreasing t_f-value. The fourth and last stage outputs the nodes of each tree in the DFS forest of the third stage as a strongly connected component. The advantage of Tarjan's algorithm is that we will have all the system runs that violate the property. The disadvantage is the amount of memory required to store the whole state space. The algorithm, though, is particularly useful when one targets *verification* and not *falsification*.

4.5.2 OWCTY

The OWCTY (One Way Catch Them Young) algorithm (Fisler *et al.* 2001) is another off-line LTL model checking algorithm. It was proposed in the context of *symbolic model checking* (Burch *et al.* 1992), where a set of states is represented as a Boolean formula called as Binary Decision Diagram, or BDD for short. BDDs utilize the similarities in the bit-vectors of states to maintain a Trie-like data structure. In the best case, BDDs can compress the state set to only a logarithmic amount. The membership query takes a linear time in the size of the state vector.

The approach is similar to Tarjan's algorithm. It computes the entire reachability set before extracting the cycle. But, unlike Tarjan's algorithm, the order of the exploration is irrelevant. Once the whole state space is generated a loop alternates between a *reachability phase* and an *elimination phase* until a fixpoint is reached. In the first phase, accepting states are checked if they can be reached again. In the second phase, states with a determined accepting status are eliminated from the search. The worst case number of iterations is bounded by the diameter d of the search space. The explicit state conversion of the approach runs

in $O(d \cdot (|\mathcal{R}| + |\mathcal{S}|))$ time. Cycle extraction for counter-example generation runs in linear time. Černá & Pelánek (2003a) proposed an explicit state version of the symbolic OWCTY algorithm designed for a distributed setting.

4.5.3 Maximally Accepting Predecessors

Brim *et al.* (2004) put forward another on-the-fly accepting cycle detection algorithm that they termed as *Maximally Accepting Predecessors* (MAP). The underlying idea revolves around analyzing the set of accepting predecessors of a state. Here, *predecessors* of a state v corresponds to the set of all states u such that a directed path from u to v exists in the graph; if $u \in \mathcal{T}$, it is an *accepting predecessor*. They observed that each accepting state lying on an accepting cycle is its own accepting predecessor. Since, it is very expensive in terms of space to keep all the predecessors, it was suggested to keep only the *maximally accepting predecessor*, where the *maximum* is defined through a linear total ordering on the state space. An accepting cycle detection exists if for an accepting state v, the maximal accepting state of v is v itself. The inverse implication does not necessarily hold. Hence, several iterations of the state space are performed until a fix-point is reached.

4.6 Directed Model Checking

Among the techniques to overcome the *state space explosion* problem, *directed model checking* has been established as one of the key technologies to lessen the burden of finding short counterexamples for design bugs quickly. Driven by the success of guided state-space exploration in AI, model checking algorithms exploit the property specification and the system to orient the search towards their falsification. The roots of validation with guided search are quite old. Some basic ideas have been present since the very first days of the automated verification of concurrent systems. For instance, Approver (Jan 1978), probably the first tool for the automated verification of communication protocols, used a directed search of the state space. The *SpotLight* (Yang & Dill 1998) system already applied the basic AI search algorithm A* (Pearl 1985) to combat the state explosion problem.

The term *directed model checking* was coined by (Edelkamp, Leue, & Lluch Lafuente 2004) who implemented the guided explicit-state model checker HSF-SPIN on top of SPIN (Holzmann 2004). Both in theory and practice, this reference is the basis of our work. In HSF-SPIN, safety violation checking is handled by replacing standard DFS (or BFS) by A*. Given that the provided heuristic estimate is *admissible*, i.e., it never over-estimates the optimal distance to the goal, A* is guaranteed to return a shortest (step-minimal) counterexample.

There are two important advantages of directed model checking:

1. **Memory savings:** Search guidance results in exploration of less states which, in turn, implies saving less states in the hash table.

2. **Shorter error trails:** Given admissible heuristics, directed model checking, is guaranteed to find the optimal path from the start state to the error state. For system engineers that want to fix the model based on this error path, it is a crucial advantage. The shorter the path, the easier it is to fix it.

For a detailed exposition of directed model checking, the readers are referred to (Lluch Lafuente 2003a).

4.7 Heuristics for Directed Model Checking

For defining heuristics for safety model checking, we assume that the global state space is generated based on the asynchronous compositions of local state spaces $\mathcal{P}_i$, $i \in \{1, \ldots, N\}$, called as *processes* (same as EFSMs). In other words, each global system state is partitioned into N local states. The state of a local process $\mathcal{P}_i$ is called its *program counter*, $i \in \{1, \ldots, N\}$, pc_i for short. In the following, we enumerate some of the heuristics from (Edelkamp, Lluch Lafuente, & Leue 2001). For a detailed analysis and description, the readers are strongly encouraged to see the orginal source.

4.7.1 Heuristics for Safety Properties

Formula-based Heuristic: The *formula-based heuristic* h_f is recursively defined on (safety) property specifications. In case of a property formula $\phi \vee \psi$, the heuristic estimate is defined as $\min\{h_f(\phi), h_f(\psi)\}$. If $\phi \wedge \psi$, the heuristic is given by $h_f(\phi) + h_f(\psi)$.

Number of Active Processes Heuristic: This heuristic mainly targets deadlock searches. It is based on the fact that a system is in deadlock, if progress is not possible in any of the subsystems. Hence, in order to arrive at a deadlock, states that have many blocked processes should be preferred over the states with few blocked processes. For most of the models, the heuristic is admissible: if each transition blocks only one process, the total number of active processes denotes a lower bound on the transitions required to reach a deadlock. The heuristic is also consistent (Lluch Lafuente 2003a).

FSM Distance Heuristic: The *FSM distance heuristic* is defined as the sum for each p_i of the distance between the local state of p_i in s and the local state of p_i in s', i.e.,

$$h_M(\langle s, s' \rangle) = \sum_{i=1}^{n} \delta_i(pc_i(s), pc_i(s')),$$

where $\delta_i(pc_i(s), pc_i(s'))$ denotes the shortest path from $pc_i(s)$ to $pc_i(s')$ in the automaton representation of $\mathcal{P}_i$. The values for δ_i are computed prior to the search.

4.7.2 Heuristic for Liveness Properties

Trail-directed Heuristic The FSM distance heuristic assumes that both states s and s' are known to the exploration module. It has mainly been used in *trail-directed search*, where a counter-example to an existing error state is to be shortened. It has also been applied to the verification of liveness properties where the prefix path to the start of the cycle and the accepting cycle itself are shortened in sequence. For this case the distance in the never-claim automaton $\mathcal{B}(\neg\phi)$ is included as follows

$$h'_M(\langle s, s' \rangle) = \max \left\{ h_M(\langle s, s' \rangle), \delta_{\mathcal{B}(\neg\phi)}(pc_{\mathcal{B}(\neg\phi)}(s), pc_{\mathcal{B}(\neg\phi)}(s')) \right\}.$$

As the product of different processes is asynchronous, it is not difficult to see (Lluch Lafuente 2003a) that the FSM distance is *monotone*, i.e., $h_M(s) - h_M(s') \leq 1$ for each pair (s, s') with

s' being the direct successor of s. Monotone heuristics guarantee the optimality of counterexample paths in heuristic search exploration algorithms like A* (Pearl 1985). Since the maximum of two monotone heuristics is monotone, $h'_M(s, s')$ is also a monotone heuristic for shortening liveness trails.

For liveness properties, HSF-SPIN contributes an improved nested-DFS algorithm based on the classification of the automata representation of the property (the *never claim*) in strongly-connected components. Later on, trail-improvement and partial-order reduction were integrated into the system (Lluch Lafuente, Leue, & Edelkamp 2002).

4.8 Summary

We have seen a brief introduction to the field of model checking in general and to on-the-fly automata-based model checking in particular. One of the most important steps in model checking is to actually build a formal model of the system. In this regard, We discussed Extended Finite State Machines, for which decidability is guaranteed due to the finiteness of the variable domains. In this dissertation, we only concentrate on the specification properties that can be written as Linear Temporal Logic (LTL) formulae. For LTL model checking, we utilize an automata-based approach where both the model and the property specification are transformed into Büchi automata and are checked for paths (for safety) or accepting cycles (for liveness).

The safety and liveness checking can be accelerated by using *directed model checking*. It utilizes heuristic guidance to reduce the search efforts by directing the search to the error. The advantages are two-fold: reduced search efforts implying less memory consumption, and shorter error trails. These advantages are especially welcomed by the engineers to easily localize the causes of the error and fix the model.

The non-exhaustive error-detection model checking algorithms are able to analyze much larger models, but, even for guided search space traversal, it is crucial to bound the resources in computation time and main memory. Directed model checking provides us the link to the work in the previous chapter on External A*. In the subsequent chapters, we will extend integrate EM heuristic search for directed LTL model checking.

Chapter 5

Safety Properties

In the previous chapter, we studied a reduction of an LTL model checking problems to a problem of path search (for safety) and to an accepting cycle search (for liveness) in an implicit graph. The main difficulty faced while dealing with model checking graphs is the so-called *state space explosion problem*. *Directed model checking* (Edelkamp, Leue, & Lluch Lafuente 2004) provides a solution to mitigate this problem by directing the search toward the erroneous states of the system. It has shown great potential in dealing with large systems, but again, the space and time requirements of the model checking procedure itself can easily push directed model checking off its limits. Extending the work of Edelkamp, Leue, & Lluch Lafuente (2004), this chapter integrates external heuristic search into directed model checking.

So far, we have looked at undirected graphs with uniform weights only. However, in model checking, the transition systems are often *directed*, i.e., it is not always possible to invert a transition. This, in turn, prevents a trivial application of External A* designed for undirected graphs. Moreover, designs of softwares and communication protocols often contain atomic regions. An atomic region corresponds to a block of statements that should be executed without the intervention of any other process. A typical example is the *test-and-set* instruction set which is assumed to be indivisible. Atomic regions are represented in the general state graph as arcs with weights equal to the number of instructions in the atomic region. This condition motivates the extension of External A* to weighted and directed implicit graphs.

Structure of the chapter: We first discuss the effects of directed and weighted edges on the workings of External A*. A method to determine the number of previous layers necessary for duplicate detection for external model checking, is presented next. In the context of model checking, we present a theoretical upper-bound on the number of previous layers that can be computed without generating the whole implicit graph. We then shift our attention to weighted edges, and their influence on undirected and directed graphs. Later, an extension of the Spin model checker for external guided exploration is studied along with the experimental results that validate the approach in a practical setting. Finally, a discussion on related work in the context of external memory model checking, is presented.

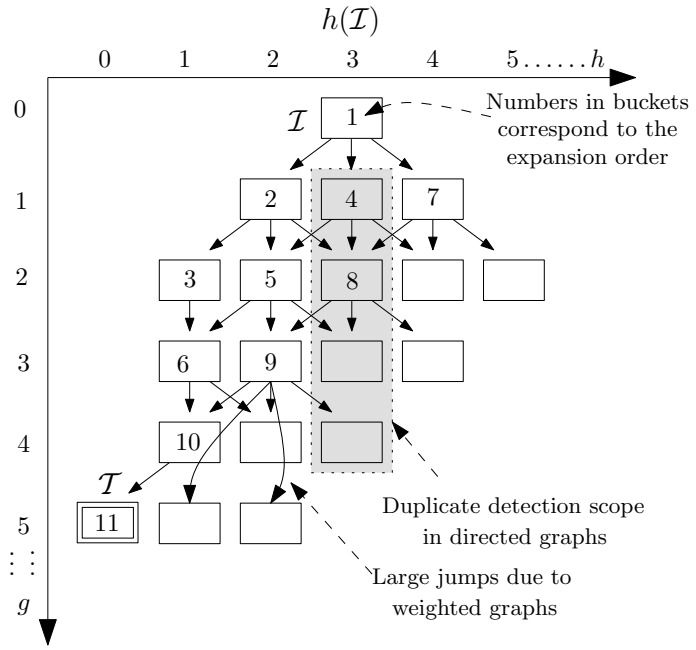

Figure 5.1: Effects of directed and weighted graphs on the workings of External A*. The shaded area corresponds to the increased duplicate detection scope due to directed edges. Curved arrows show the large jumps due to weighted edges. The numbers in the buckets represent the expansion order.

5.1 Effects of Directed and Weighted Graphs on External A*

Directed edges result in an increase in the duplicate detection scope. Duplicate detection scope dictates the number of previous layers, one has to look at to remove all the duplicate states. Recall that for undirected graphs, it was enough to subtract the previous two layers to guarantee that no node will be expanded twice. This was possible because the whole previous search frontier (the layer $l-1$) was also generated while expanding the layer l. Such is not the case for directed graphs. Directed edges result in longer back edges, i.e., it is possible that the states that were seen in layer $l-k$, for some $0 \leq k \leq l$, are generated again through l. Hence, in order to collect all the duplicate states, we need to look at k previous layers.

On the other hand, weighted edges spread the successors of a state with a depth value of l, not only in $l+1$ layer, but in layers $l+1, l+2, \ldots$. The changes that directed and weighted edges bring on the successor generation and duplicate detection is best depicted in Figure 5.1. The shaded area shows the increased duplicate detection scope, while the effect of weighted graphs is shown with the curved arrows.

5.2 Notations

To represent the weighted Kripke structures, we extend the definition of Kripke structures with a weight function $w : \mathcal{R} \rightarrow \{1, 2, \ldots, C\}$ that maps a transition to a positive integer in

range $[1, C]$. The *weight* of a path $\pi = s_0, s_1, \ldots, s_k$ can then be defined as

$$w(\pi) = \sum_{i=1}^{k} w(s_{i-1}, s_i).$$

A path π is an *optimal* path from a state u to a state v, if its weight is minimal among all paths between u and v; in this case, its accumulated weight is called the *shortest path weight*, written as $\delta(u, v)$. The shortest path weight to a set of target/accepting states $\mathcal{T}$ starting from a set of initial states $\mathcal{I}$ is abbreviated as

$$\delta(\mathcal{I}, \mathcal{T}) = \min_{i \in \mathcal{I}} \left\{ \min_{t \in \mathcal{T}} \{\delta(i, t)\} \right\}.$$

A Kripke structure or a state-transition system is obtained by repetitive applications of the legal transitions on the states of an EFSM. This process is usually called *unfolding*. The topology of the underlying implicit graph thus obtained, can be characterized based on the particular unfolding method.

Definition 5.1 (Breadth-First Search Graph) *A* Breadth-First Search graph *refers to the topology or the arrangement of a Kripke structure obtained by:*

1. *unfolding an EFSM in a breadth-wise manner, i.e., expanding the states closest to the initial state first, and*

2. *in case of duplicate states, keeping only the copy of the state with the lowest BFS number.*

It immediately follows from the definition that in Breadth-First Search graphs, the shortest paths are preserved. And in a system where positive integral weights are involved, a BFS layer is never visited twice.

We also see a direct connection between the arrangement of nodes obtained with External A* and the Breadth-First Search graphs. Since External A* preserves the shortest path distances and keeps only the copy of the state closest to the initial states, each row of the matrix of External A* can be viewed as a layer of the Breadth-First Search graph. Each layer, though, is now disjointly partitioned by a total heuristic function into buckets. Let $\mathit{Open}'(l)$ represents the l-th layer of the Breadth-First Search graph, and $\mathit{Open}(l, h)$ represents the bucket of External A* in the l-th row with a heuristic estimate h. Moreover, let H be the maximum heuristic value encountered during the exploration. If f^* is the length of the optimal solution between $\mathcal{I}$ and $\mathcal{T}$, then we can say that,

$$\forall l \leq f^*, \bigcup_{h=0}^{H} \mathit{Open}(l, h) \subseteq \mathit{Open}'(l).$$

In other words, the buckets of External A* lying on the optimal solution diagonal f^* and to the left of it, can also be found in the same layer of the Breadth-First Search graph.

On the contrary, a Depth-First traversal policy strictly violates both conditions of Definition 5.1. The implications of the second condition are severe: once a state is visited at a higher depth, its duplicate at a lower depth is simply pruned away – disrupting the shortest path distances.

5.3 Longest Back-Edge in Directed Graphs

The efficiency of external Breadth-First Search and External A* is dependent on their *duplicate detection scope*. The duplicate detection scope dictates the number of layers that have to be looked at in order to guarantee that no state is expanded twice. The actual number of previous layers depends on the longest back-edge in the graph.

Definition 5.2 (Back-Edge) *A transition* $(u, v) \in \mathcal{R}$ *is a* back-edge, *if* $\delta(\mathcal{I}, v) \leq \delta(\mathcal{I}, u)$.

Definition 5.3 (Longest Back-Edge) *A transition* $(x, y) \in \mathcal{R}$ *is the* longest back-edge, *if*

$$(x, y) = \arg\max_{(u,v)\in\mathcal{R}} \{\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v)\}.$$

The concept of longest back-edge is referred to as the *locality* of a graph by Zhou & Hansen (2006a). In the subsequent discussion, we will use their formulation.

Definition 5.4 (Locality) *The* locality *of a directed and unweighted graph* $\mathcal{G}$ *is defined as*[*]

$$\mathcal{L}(\mathcal{G}) \quad = \quad \max_{u,v\in\mathcal{S} \mid v\in \mathsf{Succ}(u)} \{\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v)\} + 1. \tag{5.1}$$

Intuitively, assuming a Breadth-First Search, locality corresponds to the maximum number of steps after which v will be re-generated through u.

The following theorem argues that subtracting *locality* many layers guarantees that no re-expansion occurs throughout a Breadth-First generation of the state space[†].

Theorem 5.1 (Duplicate Detection in Breadth-First Search Graphs) *Given a directed and unweighted Breadth-First Search graph, the total previous layers of the graph that need to be subtracted to prevent duplicate search efforts are equal to the locality of the state space graph.*

Proof We prove by induction. Let u be a node with v as one of its successors. Assume that u is in layer l and for all the nodes in layers $0 \ldots l$, no node has appeared twice. Moreover, we assume that the shortest distance values $\delta(\mathcal{I}, u)$ for every node u are known.

Expanding u generates the successor v. There can be three cases:

- Case I: $\delta(\mathcal{I}, u) < \delta(\mathcal{I}, v)$: In this case, v has never appeared before. Substituting $\delta(\mathcal{I}, v)$ with $\delta(\mathcal{I}, u) + 1$ in Definition 5.4 gives a locality of zero. Hence, no previous layer has to be looked at, making v available in layer $l + 1$ for expansion.
- Case II: $\delta(\mathcal{I}, u) = \delta(\mathcal{I}, v)$: This case appears when both u and v are generated in the layer l for the first time. Substituting $\delta(\mathcal{I}, v)$ with $\delta(\mathcal{I}, u)$ in Definition 5.4 gives us a locality of at least 1 and subtracting just the layer l from layer $l + 1$ will remove v.
- Case III: $\delta(\mathcal{I}, u) > \delta(\mathcal{I}, v)$: In this case, v must have appeared $\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v)$ layers earlier. Since the new v is in $\delta(\mathcal{I}, u) + 1$ layer, subtracting $\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + 1$ many layers will remove v from layer $l + 1$.

Taking a *maximum* of $\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v)$ over all transitions $(u, v) \in \mathcal{R}$ to compute the value of *locality* completes our proof. ■

[*]Our definition slightly varies from that by (Zhou & Hansen 2006a), as we include $+1$ in order to monitor the transition costs from u to v and to fit the implementation of External A*.

[†]The paper (Zhou & Hansen 2006a) provides an incomplete proof.

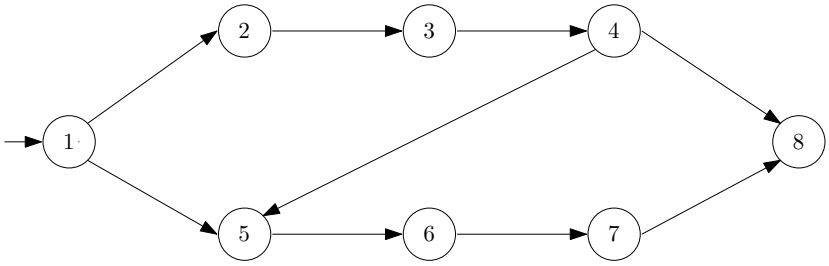

Figure 5.2: An acyclic graph with non-zero locality.

We immediately obtain the following corollary for undirected graphs:

Corollary 5.2 *In an undirected and unweighted Breadth-First Search graph $\mathcal{G}$, the number of previous layers that have to be subtracted to guarantee that no state is expanded twice, is* two.

Proof Since, in undirected graphs, the maximum difference between the shortest path distances from a start state $\mathcal{I}$ to a state u and to its successor v is 1, Equation 5.1 gives us

$$\mathcal{L}(\mathcal{G}) = \max_{u,v \in \mathcal{S} \mid v \in Succ(u)} \{\delta(\mathcal{I},u) - \delta(\mathcal{I},v)\} + 1 \quad = \quad 2$$

Using Theorem 5.1 on the resulting locality completes the proof. ■

This concept is closely related to the cycles in the graph. However, the concept of locality is more general than the longest cycle. As locality corresponds to the depth difference of a back-edge in the exploration, it is possible that in an acyclic graph, locality is non-zero. An example for such a graph is provided in Fig 5.2. The back edge from state 4 to 5 induces a locality of 3. The BFS-layers are as follows: Level 0 consists of state 1, Level 1 consists of 2 and 5, Level 2 of 3 and 6, Level 3 of 4 and 7, while Level 4 consists of state 8 only.

5.4 Duplicate Detection in Concurrent Systems

5.4.1 Asynchronous Concurrent Systems

Concurrent systems, as we have seen in the previous chapter, often consist of multiple asynchronously combined sub-systems. In the formulation of Extended Finite State Machines, a concurrent system consists of the *asynchronous product* of the sub-automata, denoted as $\oplus$. The most important issue that arises here, is that the cross product $\oplus$ is *not available* until it is generated. Hence, we lack information about the structure of the graph, and consequently, about the longest back-edge. The question that we will try to answer now is: '*Can the sub-automata of the components be used to infer the longest back-edge in the product automaton?*'.

At first glance, one might get the impression, that the longest back-edge or the locality in this composite graph would be equal to the product of the localities in the sub-automata. Fortunately, for Breadth-First Search graphs, and hence also for graphs generated through External A*, one can achieve a much *tighter* bound. For the time being, we shall only consider the structural characteristics of the sub-automata – viewing them as pure directed graphs. The graph of an EFSM p is denoted as $\mathcal{G}(p)$. It is obtained from p by simply removing all the guards and actions from the transitions. A formal definition is presented later in Definition 5.7, where it will be used more extensively.

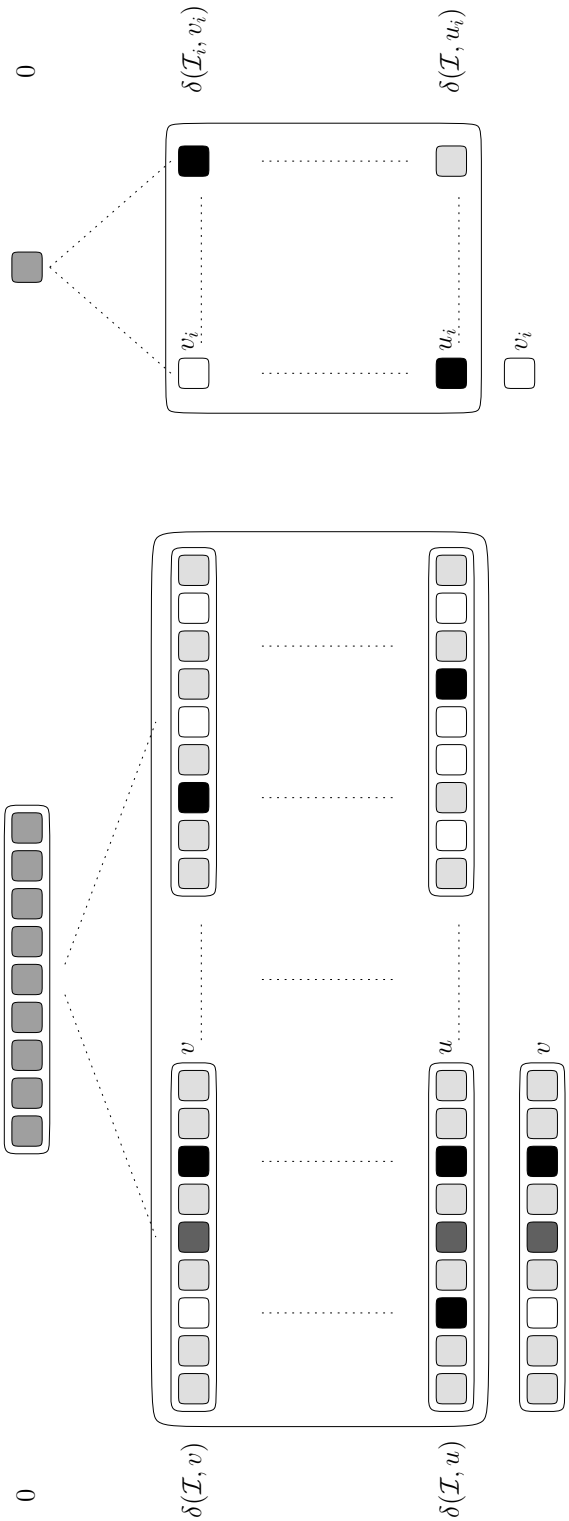

Figure 5.3: Locality of global BFS space (left), local BFS space (right)

Theorem 5.3 (Locality in Asynchronously Combined Concurrent Systems) *Let $p_1, \ldots, p_n$ be the processes in the concurrent system. Let $\mathcal{L}(\mathcal{G}(\oplus))$ be the locality in the composite system when restricted to a graph and $\mathcal{L}(\mathcal{G}(p_i))$, $i \in \{1, \ldots, n\}$, be the locality of each individual process p_i when restricted to graphs. Then, the locality in the asynchronously combined Breadth-First search graph is bounded by the maximum of the process localities, i.e.,*

$$\mathcal{L}(\mathcal{G}(\oplus)) \quad \leq \quad \max\{\mathcal{L}(\mathcal{G}(p_1)) \ldots, \mathcal{L}(\mathcal{G}(p_n))\} \tag{5.2}$$

Proof Let u be the state to expand and v be one of its successors in the global BFS space as shown in Fig. 5.3 (left). BFS layers are ordered from top to bottom with the layer number denoted to the left. The state vector $\mathcal{I}$ at the root consists of individual start states $\mathcal{I}_i$ for each process p_i, $i \in \{1, \ldots, n\}$.

We will prove by induction. Assume that for all expanded BFS layers $0, \ldots, \delta(\mathcal{I}, u)$ the states are correctly placed. As we have no control on the transition t from u to v by the asynchronous product of the system processes, t may corresponds to a transition in any process. Let p_i be the process of transition t, having its BFS space depicted in Fig. 5.3 (right). Let $\mathcal{I}_i$ (u_i, and v_i) be the projection of the state vector $\mathcal{I}$ (u and v) to the process p_i (In the global BFS space to the left of the figure, i is the third component). The only difference between u and v is in the i-th component, i.e., $u_j = v_j$ for all $j \neq i$. The locality areas for both graphs are shown as shaded rectangles in Fig. 5.3.

Because of additional transitions in the j-th component, $j \neq i$, it is immediate that $\delta(\mathcal{I}_i, v_i) \leq \delta(\mathcal{I}, v)$, and that $\delta(\mathcal{I}_i, u_i) \leq \delta(\mathcal{I}, u)$. Moreover, the differences $\delta(\mathcal{I}, v) - \delta(\mathcal{I}_i, v_i)$ and $\delta(\mathcal{I}, u) - \delta(\mathcal{I}_i, u_i)$ are the same (the same transitions in the j-th component, $j \neq i$, apply). We have

$$\begin{aligned} \delta(\mathcal{I}, u) - \delta(\mathcal{I}_i, u_i) &= \delta(\mathcal{I}, v) - \delta(\mathcal{I}_i, v_i) \\ \delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) &= \delta(\mathcal{I}_i, u_i) - \delta(\mathcal{I}_i, v_i) \\ \delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + 1 &= \delta(\mathcal{I}_i, u_i) - \delta(\mathcal{I}_i, v_i) + 1. \end{aligned} \tag{5.3}$$

By the definition of Locality, in the i-th component,

$$\delta(\mathcal{I}_i, u_i) - \delta(\mathcal{I}_i, v_i) + 1 \leq \mathcal{L}(\mathcal{G}(p_i)). \tag{5.4}$$

Combining Equations 5.3 and 5.4, we have

$$\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + 1 \quad = \quad \delta(\mathcal{I}_i, u_i) - \delta(\mathcal{I}_i, v_i) + 1 \leq \mathcal{L}(\mathcal{G}(p_i)). \tag{5.5}$$

Equation 5.5 states that for any two arbitrary chosen u and v with a transition t from u_i to v_i in the i-th component, $\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + 1$ is bounded by $\mathcal{L}(\mathcal{G}(p_i))$ (the locality in the i-th sub-automaton). Taking a maximum for all $1 \leq i \leq n$ gives

$$\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + 1 \quad \leq \quad \max\{\mathcal{L}(\mathcal{G}(p_1)), \ldots, \mathcal{L}(\mathcal{G}(p_n))\}. \tag{5.6}$$

Since the locality of the product automaton is also defined as the maximum over all the back-edges, we get,

$$\mathcal{L}(\mathcal{G}(\oplus)) \quad \leq \quad \max\{\mathcal{L}(\mathcal{G}(p_1)), \ldots, \mathcal{L}(\mathcal{G}(p_n))\}, \tag{5.7}$$

which completes the proof. ■

Example Figure 5.4 provides a simple example illustrating the result of Theorem 5.3. On top of the figure two individual process graphs are shown, while the graph at the bottom depicts the asynchronous composition of the two. It is easy to check that the locality of the first graph is 3, the locality of the second graph is 2, and the locality of the composition is 3. The BFS-layers are as follows: Level 0 consists of 1a, Level 1 consists of 2a and 1b, Level 2 consists of 3a and 2b, Level 3 consists of 4a and 3b, while Level 4 consist of 4b only. The longest back-edge is from node 4a to 2a and from 4b to 2b, implying a locality of 3.

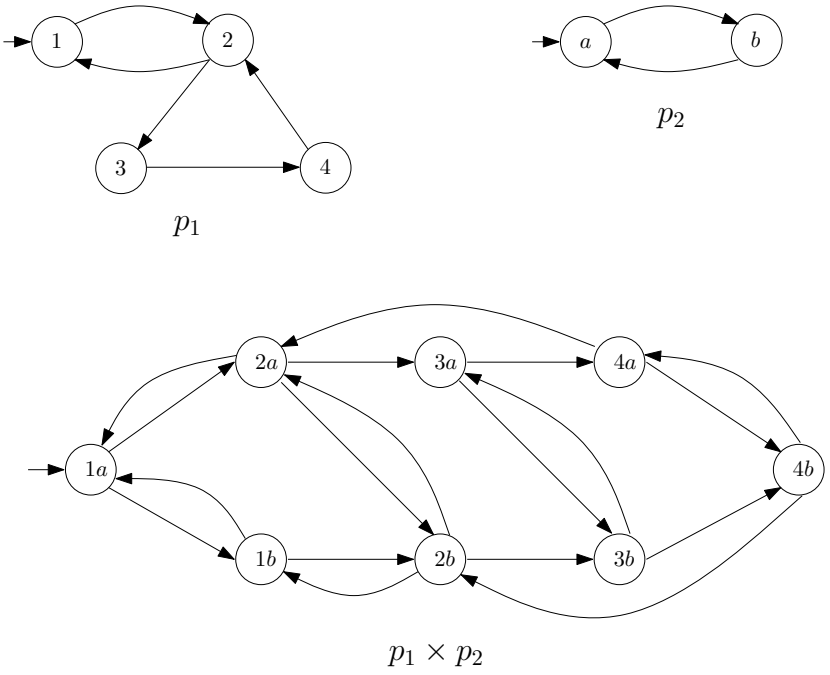

Figure 5.4: Locality in asynchronous concurrent systems. Local state spaces (top left and top right), asynchronously combined global state space (bottom).

Theorem 5.3 provides us an important result for bounding the I/O complexity in model checking graphs. It uses only the sub-automata – which are provided as the system description – to reason on the locality in the composite graph. The proof depends on two main assumptions: a) all the successors of a state are generated at the same time, and b) the shortest path distances are correct. Both external Breadth-First search and External A* satisfy these requirements that allow us to use the result on both the algorithms.

Even if the number of stored layers is less than the locality of the graph, the number of times a node can be re-opened in Breadth-First search is only linear in the depth of the search (Zhou & Hansen 2006a). This contrasts the exponential number of re-openings for linear-space Depth-First Search strategies.

5.4.2 Synchronous Concurrent Systems

We have just seen the effects of back-edges on asynchronous composition and proved that the longest back-edge in the composite graph is bounded by the maximum of all the longest

back-edges in each individual automata. Even though we restrict to model checking of asynchronous systems only as many software systems are synchronized to a universal clock, we need some bounds on the synchronous composition to prove the general case of EFSMs.

Recall that in synchronous composition, one global transition require one transition in each of the automata. This, in turn, introduces a certain form of *dependency* in the global automata. In the subsequent discussion, we will use the symbol $\otimes$ to denote a synchronous composition, as opposed to $\oplus$ for asynchronous composition.

Example Let us take the two automata from Figure 5.4 and synchronously combine them. The result is shown in Figure 5.5. The longest back-edge now goes from $4a$ to $2b$ giving rise to a locality as large as 6. This locality in the composite product is the product of the localities in the sub-automata $p_1[\mathcal{L}(p_1) = 3]$ and $p_2[\mathcal{L}(p_2) = 2]$.

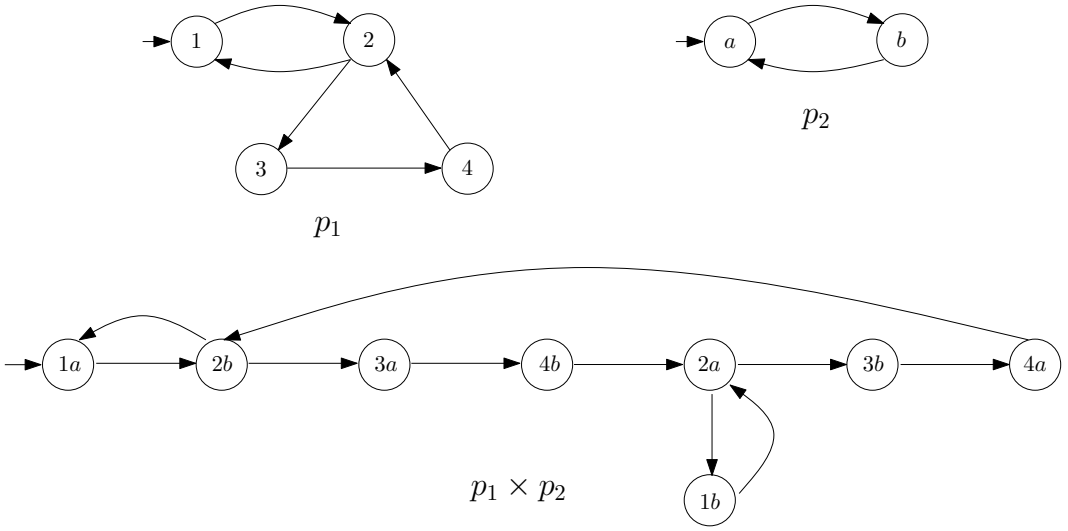

Figure 5.5: Locality in synchronous concurrent systems. Local state spaces (top left and top right); synchronously combined global state space (bottom).

In the following, we will formally prove that in a synchronous composition the locality is bounded by the product of the localities in the sub-automata.

Theorem 5.4 (Locality in the Synchronously Combined Concurrent Systems) *Let $p_1, \ldots, p_n$ be the processes in the concurrent system. Let $\mathcal{L}(\mathcal{G}(\otimes))$ be the locality in the synchronously combined composite system when restricted to a graph and $\mathcal{L}(\mathcal{G}(p_i))$, $i \in \{1, \ldots, n\}$, be the locality of each individual process p_i when restricted to graphs. Then, the locality in the synchronously combined Breadth-First Search graph is bounded by the product of the process localities, i.e.,*

$$\mathcal{L}(\mathcal{G}(\otimes)) \quad \leq \quad \mathcal{L}(\mathcal{G}(p_1)) \cdot \mathcal{L}(\mathcal{G}(p_2)) \cdot \ldots \cdot \mathcal{L}(\mathcal{G}(p_n)) \tag{5.8}$$

Proof The prove is by induction. For the base case, let us first prove the statement for two automaton p_1 and p_2 synchronously combined into an automaton $\otimes$. Let u be the state to expand and v be one of its successors in the global BFS space as shown in Figure 5.6 (left). BFS layers in the figure are ordered from top to bottom with the layer number denoted to the left. A state vector in the global state space consists of individual states from each of the automata. Let $\mathcal{I}_i$ (u_i, and v_i) be the projection of the state vector $\mathcal{I}$ (u and v) to the process p_i,

for $i = 1, 2$. Let the back-edge from u_1 to v_1 and u_2 to v_2 be the two back-edges in automata p_1 and p_2, respectively (as shown in Figure 5.6 (right)). Assume that for all expanded BFS layers $0, \ldots, \delta(\mathcal{I}, u)$ the states are correctly placed.

The node v has appeared before in the search and will be visited again by a back-edge rooted at $u = \langle u_1, u_2 \rangle$. Since the composition is synchronous, both u_i's have to agree in u to generate v (v_i in the automata). They all have to take their back-edge transitions together to generate the duplicate state v in the global space. There can be two cases:

Case I: $\delta(\mathcal{I}_1, u_1) - \delta(\mathcal{I}_1, v_1) + 1 = \delta(\mathcal{I}_2, u_2) - \delta(\mathcal{I}_2, v_2) + 1$. In this case, the *first appearance* of u_1 in the product automaton will match the first appearance of u_2. Consequently, both automata will execute their back-edge transitions together in u to generate v. The back-edge $u \rightarrow v$, in the global automaton is then clearly bounded by the same back-edge length as in p_1 or in p_2, i.e.,

$$\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + 1 = \delta(\mathcal{I}_1, u_1) - \delta(\mathcal{I}_1, v_1) + 1 = \delta(\mathcal{I}_2, u_2) - \delta(\mathcal{I}_2, v_2) + 1.$$

or,

$$\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + 1 \leq (\delta(\mathcal{I}_1, u_1) - \delta(\mathcal{I}_1, v_1) + 1) \cdot (\delta(\mathcal{I}_2, u_2) - \delta(\mathcal{I}_2, v_2) + 1).$$

Since, the locality defines the longest back-edge in each sub automaton, we can rewrite the above equation as:

$$\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + 1) \leq \mathcal{L}\left(\mathcal{G}(p_1)\right) \cdot \mathcal{L}\left(\mathcal{G}(p_2)\right).$$

The above equation states that for any arbitrary chosen back-edge $u \rightarrow v$, its duplicate detection scope is bounded by the product of the sub-automata's localities. If such an edge is also the longest back-edge, the left side can be replaced by $\mathcal{L}\left(\mathcal{G}(\otimes)\right)$, which gives

$$\mathcal{L}\left(\mathcal{G}(\otimes)\right) \leq \mathcal{L}\left(\mathcal{G}(p_1)\right) \cdot \mathcal{L}\left(\mathcal{G}(p_2)\right),$$

as the desired result.

Case II: $\delta(\mathcal{I}_1, u_1) - \delta(\mathcal{I}_1, v_1) + 1 \neq \delta(\mathcal{I}_2, u_2) - \delta(\mathcal{I}_2, v_2) + 1$. Without loss of generality, let us assume that $\delta(\mathcal{I}_1, u_1) - \delta(\mathcal{I}_1, v_1) + 1 < \delta(\mathcal{I}_2, u_2) - \delta(\mathcal{I}_2, v_2) + 1$, as shown in Figure 5.6 (right). As a result, the back-edge in p_1 will fire before the back-edge in p_2. This would result in a *delay* in the global state combination $u = \langle u_1, u_2 \rangle$ to appear. Instead of u another state, say u', will be generated. Consequently, instead of v, a new state $v' = \langle v_1, x_2 \rangle$ will be generated through u'. Since v_2 is still reachable in the BFS, it will appear next synchronized with a state, x_1 of p_1. Let $v'' = \langle x_1, v_2 \rangle$ be such state.

It is important to note here that only the states that lie on the longest back-edge $u_1 \rightarrow v_1$ and $u_2 \rightarrow v_2$ are synchronized during this delay for u. In the worst case, for $i = 1, 2$, all x_i's are bounded by $\delta(\mathcal{I}, u_i) - \delta(\mathcal{I}, v_i) + 1$. The length of the delay in the generation of u is bounded by *all possible* state combinations that can appear by synchronization of states lying on the back-edges. Hence, the difference of the BFS layers between u and v, or the length of the back-edge $u \rightarrow v$ is given as,

$$\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + 1 \leq (\delta(\mathcal{I}_1, u_1) - \delta(\mathcal{I}_1, v_1) + 1) \cdot (\delta(\mathcal{I}_2, u_2) - \delta(\mathcal{I}_2, v_2) + 1).$$

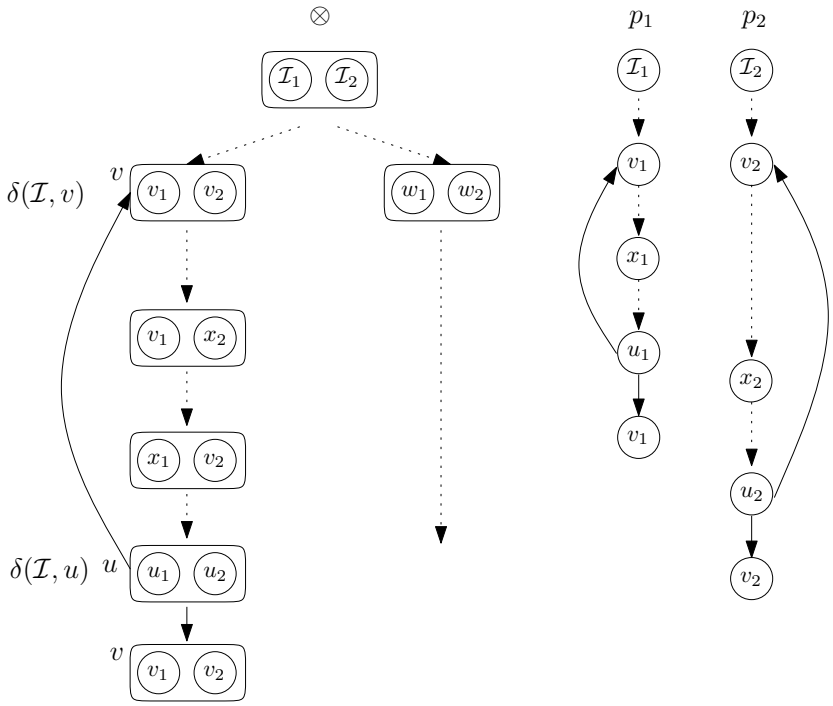

Figure 5.6: Locality in the BFS space of a synchronously composed automaton ($\otimes$) (left). BFS spaces of the sub-automata p_1 and p_2 (right). Dotted lines show sequence of transitions. Curved arrows show the back-edges.

The same steps as in Case I can now be applied to get

$$\mathcal{L}(\mathcal{G}(\otimes)) \leq \mathcal{L}\left(\mathcal{G}(p_1)\right) \cdot \mathcal{L}\left(\mathcal{G}(p_2)\right),$$

as the desired result.

For the induction step, let us assume that for $n-1$ sub-automata synchronously combined in an automaton $\mathcal{G}(\otimes_{n-1})$,

$$\mathcal{L}(\mathcal{G}(\otimes_{n-1})) \leq \mathcal{L}\left(\mathcal{G}(p_1)\right) \cdot \mathcal{L}\left(\mathcal{G}(p_2)\right) \cdot \ldots \cdot \mathcal{L}\left(\mathcal{G}(p_{n-1})\right)$$

holds. Moreover, for sake of brevity, assume that $\otimes_{n-1}$ is completely provided before-hand. Due to the assumption, the length of a back-edge $\hat{u} \rightarrow \hat{v}$ in $\mathcal{G}(\otimes_{n-1})$ is bounded by the product of the individual back-edges in the sub-automata, i.e.,

$$\delta(\mathcal{I}, \hat{u}) - \delta(\mathcal{I}, \hat{v}) + 1 \leq (\delta(\mathcal{I}_1, u_1) - \delta(\mathcal{I}_1, v_1) + 1) \cdot \ldots \cdot (\delta(\mathcal{I}_{n-1}, u_{n-1}) - \delta(\mathcal{I}_{n-1}, v_{n-1}) + 1). \quad (5.9)$$

Let us now add one more automaton p_n to the system and synchronously combine it with $\mathcal{G}(\otimes_{n-1})$. Let $\mathcal{G}(\otimes)$ be the new composite automaton. Let $u \rightarrow v$ be a back-edge obtained in $\mathcal{G}(\otimes)$ due to the back-edge $\hat{u} \rightarrow \hat{v}$ and the new back-edge $u_n \rightarrow v_n$ in p_n. The earlier proof for the base case $n = 2$ can now be applied directly to the new composition to get the length of the back-edge $u \rightarrow v$ as:

$$\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + 1 \leq \delta(\mathcal{I}, \hat{u}) - \delta(\mathcal{I}, \hat{v}) + 1 \cdot (\delta(\mathcal{I}_n, u_n) - \delta(\mathcal{I}_1, v_n) + 1).$$

Using the Inequality 5.9 on the above, we get

$$\begin{aligned}\delta(\mathcal{I},u)-\delta(\mathcal{I},v)+1 \quad &\leq \quad (\delta(\mathcal{I}_1,u_1)-\delta(\mathcal{I}_1,v_1)+1)\cdot\ldots\cdot(\delta(\mathcal{I}_{n-1},u_{n-1})-\delta(\mathcal{I}_{n-1},v_{n-1})+1)\cdot\\ &\quad\ (\delta(\mathcal{I}_n,u_n)-\delta(\mathcal{I}_n,v_n)+1)\end{aligned}$$

Since locality is defined as the longest back-edge, we get,

$$\delta(\mathcal{I},u)-\delta(\mathcal{I},v)+1 \quad \leq \quad \mathcal{L}\left(\mathcal{G}(p_1)\right)\cdot\ldots\cdot\mathcal{L}\left(\mathcal{G}(p_{n-1})\right)\cdot\mathcal{L}\left(\mathcal{G}(p_n)\right). \tag{5.10}$$

Equation 5.10 gives the bound for an arbitrarily chosen back-edge (u,v) in a synchronously combined automaton. Taking a maximum over all such edges gives

$$\mathcal{L}(\mathcal{G}(\otimes)) \quad \leq \quad \mathcal{L}\left(\mathcal{G}(p_1)\right)\cdot\ldots\cdot\mathcal{L}\left(\mathcal{G}(p_{n-1})\right)\cdot\mathcal{L}\left(\mathcal{G}(p_n)\right).$$

which completes the proof. ■

5.5 Duplicate Detection in Communicating Concurrent Systems

In the previous section, we presented an upper bound on the duplicate detection scope of concurrent systems, where the variables and guards were not taken into account. Variables and guards provide a means for concurrent systems to communicate with each other. In their presence, the behavior of the overall system can change drastically. Recall that the proof of Theorem 5.3 exploits the observation that all transitions are enabled at all times. The guards, on the other hand, can block the execution of a transition based on the value of a variable. The implications are severe. Due to the blocking of a transition, at a state, say u, the generation of u's successor, say v, would be delayed. This, in turn, will effect the shortest path distance $\delta(u,v)$, and subsequently, the longest back-edge and the locality. We denote this phenomenon as *edge stretching* in the following discussion.

Variables as Automata Given a variable from a finite domain, its behavior in an EFSM can be modeled as a finite automaton as follows.

Definition 5.5 (Variable Automaton) *Let x be a variable from a finite domain $D(x)$. A* variable automaton *$\mathcal{G}(x)$ of an EFSM $p=\langle Q, Var=\{x\}, Guards, Actions, \Delta, \mathcal{I}\rangle$, is a finite-state automaton $\mathcal{G}(x)=\langle \mathcal{S}_x, \mathcal{R}_x, \mathcal{I}_x\rangle$, where*

- *$\mathcal{S}_x$ is a finite set of states with each state corresponding to a unique variable value, i.e., $|\mathcal{S}_x| = |D(x)|$, and*
- *$\mathcal{R}_x \subseteq \mathcal{S}_x \times \mathcal{S}_x$ is the transition relation. A transition (s,s') exists, if a transition $(q_i, guard, action, q_j) \in \Delta$ exists such that $guard(s) =$ `true` and $action(s) = s'$.*
- *$\mathcal{I}_x$ is the initial value of the variable.*

If the EFSM involves more than one variable, say k, a single automaton $\mathcal{G}(Var)$ modeling the combined behavior of all the variables can be obtained as follows:

Definition 5.6 (Multi-Variables Automaton) *Let $Var = \{x_1 \ldots x_k\}$ be a set of variables from finite domains $D(x_1)\ldots D(x_k)$. A* multi-variables automaton *$\mathcal{G}(Var)$ of an EFSM $p=\langle Q, Var, Guards, Actions, \Delta, \mathcal{I}\rangle$, is a finite-state automaton $\mathcal{G}(Var)=\langle \mathcal{S}_{Var}, \mathcal{R}_{Var}, \mathcal{I}_{Var}\rangle$, where*

- $\mathcal{S}_{Var}$ *is a finite set of states with each state corresponding to a unique valuation vector* $\gamma \in \Gamma$, *i.e.,* $|\mathcal{S}_{Var}| = |\Gamma| = |D(x_1)| \cdot \ldots \cdot |D(x_k)|$, *and*
- $\mathcal{R}_x \subseteq \mathcal{S}_x \times \mathcal{S}_x$ *is the transition relation. A transition* (γ, γ') *exists, if a transition* $(q_i$, *guard, action,* $q_j) \in \Delta$ *exists such that guard*$(\gamma) =$ `true` *and action*$(\gamma) = \gamma'$.
- $\mathcal{I}_{Var}$ *is the initial state consisting of the initial valuation vector.*

EFSM as a Graph: Assume that an *oracle* exists that has provided the multi-variables automaton of an EFSM p in advance. We now relax the definition of EFSM to only contain the automaton structure and no further information on guards and actions.

Definition 5.7 (Graph of an EFSM) *The* graph of an EFSM $p = \langle Q, Var, Guards, Actions, \Delta, \mathcal{I} \rangle$ *is a 3-tuple* $\mathcal{G}(p) = \langle Q, \Delta_g, \mathcal{I} \rangle$, *where*

- Q *is the set of states defined as in* p,
- $\mathcal{I} \subseteq Q$ *is the set of initial states defined as in* p,
- $\Delta_g \subseteq Q \times Q$ *is a transition relation. A transition* $(q, q') \in \Delta_g$ *exists, if and only if,* $(q'$, *guard, action,* $q') \in \Delta$ *exists for* p.

We now synchronously combine an EFSM, given as a graph (cf. Definition 5.7), and the multi-variable automaton to obtain a state-transition system.

Definition 5.8 (State-Transition System as a Composition of $\mathcal{G}(Var)$ and $\mathcal{G}(p)$) *Let* p *be an Extended Finite State Machine. The synchronous composition of a multi-variables automaton* $\mathcal{G}(Var) = \langle \mathcal{S}_{Var}, \mathcal{R}_{Var}, \mathcal{I}_{Var} \rangle$ *and the graph of the EFSM* $\mathcal{G}(p) = \langle Q, \Delta_g, \mathcal{I} \rangle$ *results in a state-transition system, defined as* $ST(\mathcal{M}) = \langle \mathcal{S}, \mathcal{R}, \mathcal{I} \rangle$, *where*

- $\mathcal{S} \subseteq Q \times \mathcal{S}_{Var}$ *is a set of states,*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ *is the transition relation. A transition* $((q_i, \gamma_i), (q_j, \gamma_j)) \in \mathcal{R}$ *exists, if and only if,* $(q_i, q_j) \in \Delta_g$ *and* $(\gamma_i, \gamma_j) \in \mathcal{R}_{Var}$.
- $\mathcal{I} \subseteq \mathcal{S}$ *is the set of initial states defined exactly as in the EFSM.*

Equivalence of the two state-transition systems: Are the two state-transition systems, i.e., the one defined through the unfolding of an EFSM p and the one defined by the synchronization of $\mathcal{G}(Var)$ and $\mathcal{G}(p)$, equal? The question can be reformulated as: Does the synchronization guarantee that by taking a transition in $\mathcal{G}(p)$, only the legal change in values of the variables occur in $\mathcal{G}(Var)$? To answer this question, we first need to make the synchronization as defined in Definition 5.8 more conservative. This can be done by only minor changes in both the $\mathcal{G}(Var)$ and $\mathcal{G}(p)$. To achieve a correct synchronization, we need to equip both the automata with same transition labels. These transition labels can be defined at the construction time of $\mathcal{G}(Var)$, i.e., for every transition between two valuations, a unique label is attached to the edge in $\mathcal{G}(Var)$. The same label is then also assigned to the corresponding transition in $\mathcal{G}(p)$. Let A be the set of labels. The next step is to redefine the transition relation in Definition 5.8 as: A transition $((q_i, \gamma_i), (q_j, \gamma_j)) \in \mathcal{R}$ exists, if and only if, $(q_i, a, q_j) \in \Delta_g$ and $(\gamma_i, a, \gamma_j) \in \mathcal{R}_{Var}$, where $a \in A$ is the common transition label. This change guarantees that the state-transition system correctly simulates the unfolding of an EFSM.

Having necessary formalisms in hand, we now proceed to define an upper-bound on the duplicate detection scope of an EFSM.

Theorem 5.5 (Duplicate Detection Scope of an EFSM) *In a Breadth-First Search graph obtained from the unfolding of an EFSM p, the duplicate detection scope is bounded by the product of its structural locality and the cardinalities of the variable domains.*

$$\mathcal{L}(p) \quad \leq \quad \mathcal{L}(\mathcal{G}(p)) \cdot (|D(x_1)| \cdot |D(x_2)| \cdot \ldots |D(x_k)|) \tag{5.11}$$

Proof An EFSM $p = \langle Q, \mathit{Var}, \text{Guards}, \text{Actions}, \Delta, \mathcal{I} \rangle$ can be sub-divided into a multi-variables automaton $\mathcal{G}(\mathit{Var})$ (cf. Definition 5.6) and a corresponding graph without actions and guards $\mathcal{G}(p)$ (cf. Definition 5.7). Both these automata can be synchronously combined to correctly simulate the behavior of p by using the composition defined in Definition 5.8 and an extra action labels set.

Theorem 5.4 bounds the locality or the longest back-edge of a synchronous composition of two automaton p_1 and p_2 by the product of individual longest back-edges in the two. In our case, we have the full knowledge of the first automaton $\mathcal{G}(p)$ and the longest back-edge $\mathcal{L}(\mathcal{G}(p))$ that appears in it. Unfortunately, the second automaton, $\mathcal{G}(\mathit{Var})$ is not provided before-hand and has to be generated on-the-fly. The only speculation we can do about the longest back-edge in $\mathcal{G}(\mathit{Var})$ is that, in the worst-case (graph is a large cycle), it is bounded by the total number of states. Since each state in $\mathcal{G}(\mathit{Var})$ is basically a valuation vector, given finite domains, there can be at most $|D(x_1)| \cdot |D(x_2)| \cdot \ldots |D(x_k)|$ states.

Combining both the bounds on longest back-edges in $\mathcal{G}(p)$ and $\mathcal{G}(\mathit{Var})$ using Theorem 5.4 results in

$$\mathcal{L}(p) \quad \leq \quad \mathcal{L}(\mathcal{G}(p)) \cdot (|D(x_1)| \cdot |D(x_2)| \cdot \ldots |D(x_k)|) \tag{5.12}$$

as an upper-bound on the locality of an EFSM p. ■

5.6 A Worst-case Example

In the following, we will give a worst-case example to show that the bound given by Theorem 5.5 is tight, i.e., it is possible to create an EFSM, where the longest back-edge is bounded by the value given by the Inequality 5.11 We will first compute an upper bound on the duplicate detection scope of an EFSM including the variables and guards.

Example Assume an EFSM with just one control state q_0 and one transition r_0 as a self-loop. The transition r_0 contains a guard which states that if $x_0 < |D(x_0)|$, r_0 can be taken, while executing the statement $x_0 = x_0 + 1$. By the definition of a Kripke structure, a state is a tuple (q, γ), where q is the control state and $\gamma \in \Gamma$ is the valuation vector, which in our case consists of only one element, namely the value of x_0. A state (q, γ) is a duplicate state, if both components have already appeared together in another state. Hence, the total number of unique states in the unfolded Kripke structure would be equal to $|D(x_0)|$ – there is just one control state q_0 and γ changes for every transition.

Now let us assume that the EFSM has k variables, each from a finite domain. Let $\mathcal{L}(\mathcal{G}(p))$ be the structural locality of the EFSM. A representative EFSM is shown in Figure 5.7 (a),

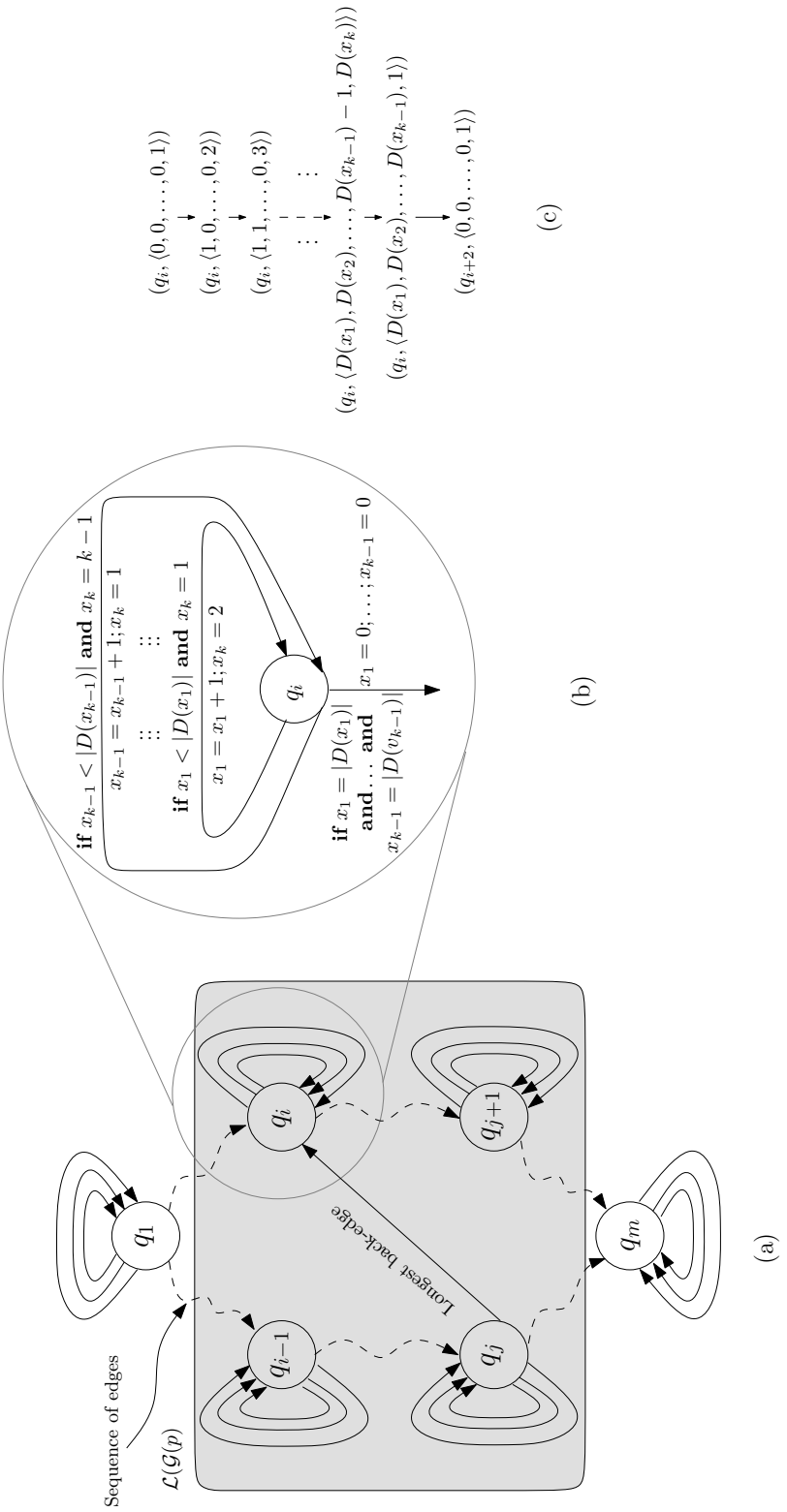

Figure 5.7: (a)An EFSM; (b) An EFSM state with details on guards and actions. All states have exactly the same guards and actions structure; (c) The unfolded Kripke structure at q_i.

where the longest back-edge is due to the edge from q_j to q_i. Furthermore, let us assume that each control state has $k-1$ self-loop transitions and one or more outgoing transitions. Each of the self-loops increments the value of a variable (if it is still within its bounds) and then makes the next self-loop executable. In Figure 5.7 (b), the zoomed section of an EFSM state is shown. The alternation between different self-loops is synchronized with the help of the variable x_k that can take its values from the domain $D(x_k) = \{1, 2, \ldots, k-1\}$ ($k-1$ is the total number of self-loops).

Since the self-loops are defined in such a way that only one of them is executable at any given time, unfolding them at a control state q_i, would generate the following sequence of states:

$$(q_i, \langle 0, 0, \ldots, 0, 1\rangle), (q_i, \langle 1, 0, \ldots, 0, 2\rangle), \ldots, (q_i, \langle |D(x_1)|, |D(x_2)|, \ldots, |D(x_{k-1})|, 1\rangle)$$

with each state appearing in a different layer. This gives a total of $|D(x_1)| \cdot |D(x_2)| \cdot \ldots \cdot |D(x_{k-1})|$ layers required for unfolding the self-loops on an EFSM state. In the worst case, the number of layers required after entering in an EFSM state and reaching another state is bounded by

$$|D(x_1)| \cdot |D(x_2)| \cdot \ldots \cdot |D(x_{k-1})| \cdot |D(x_k)|.$$

If $\delta(u, v)$ represents the shortest path distance in the EFSM between two control states u and v, the shortest path in the corresponding unfolded Kripke structure from the start state $\mathcal{I} = q_1$ to q_i would be equal to

$$\delta\left((\mathcal{I}, \langle 0, 0, \ldots 0, 1\rangle), (q_i, \langle 0, 0, \ldots 0, 1\rangle)\right) = \delta(\mathcal{I}, q_i) \cdot (|D(x_1)| \cdot |D(x_2)| \cdot \ldots \cdot |D(x_{k-1})| \cdot |D(x_k)|)$$

According to Definition 5.4, the duplicate detection scope due to a back-edge (u, v) is given by $\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + 1$. Here the last term $+1$, is due to the weight of the transition from u to v that brought the search to a lower BFS level. In our case, the weight of this edge would stretch to $(|D(x_1)| \cdot |D(x_2)| \cdot \ldots \cdot |D(x_{k-1})| \cdot |D(x_k)|)$. Hence, the duplicate detection scope due to some back-edge (q_j, q_i) would be:

$$\begin{aligned}
& \delta((\mathcal{I}, \langle 0, 0, \ldots 0, 1\rangle), (q_j, \langle 0, 0, \ldots 0, 1\rangle)) \\
& - \delta((\mathcal{I}, \langle 0, 0, \ldots 0, 1\rangle), (q_i, \langle 0, 0, \ldots 0, 1\rangle)) + 1 \\
= \; & \delta(\mathcal{I}, q_j) \cdot (|D(x_1)| \cdot |D(x_2)| \cdot \ldots \cdot |D(x_{k-1})| \cdot |D(x_k)|) \\
& - \delta(\mathcal{I}, q_i) \cdot (|D(x_1)| \cdot |D(x_2)| \cdot \ldots \cdot |D(x_{k-1})| \cdot |D(x_k)|) \\
& + (|D(x_1)| \cdot |D(x_2)| \cdot \ldots \cdot |D(x_{k-1})| \cdot |D(x_k)|) \\
= \; & (\delta(\mathcal{I}, q_j) - \delta(\mathcal{I}, q_i) + 1) \cdot (|D(x_1)| \cdot |D(x_2)| \cdot \ldots \cdot |D(x_{k-1})| \cdot |D(x_k)|)
\end{aligned}$$

Taking the maximum over all q_i, q_j pairs with $q_i \in \mathit{Succ}(q_j)$ on the above equation we have,

$$\begin{aligned}
\leq \; & \max_{q_i, q_j | q_i \in \mathit{Succ}(q_j)} \{\delta(\mathcal{I}, q_j) - \delta(\mathcal{I}, q_i) + 1\} \cdot (|D(x_1)| \cdot |D(x_2)| \cdot \ldots \cdot |D(x_{k-1})| \cdot |D(x_k)|) \\
= \; & \mathcal{L}\left(\mathcal{G}(p)\right) \cdot (|D(x_1)| \cdot |D(x_2)| \cdot \ldots \cdot |D(x_{k-1})| \cdot |D(x_k)|)
\end{aligned}$$

which verifies our claim that the bound provided by Theorem 5.5 is tight.

It is straightforward to generalize the result for n EFSMs using an extended worst-case example. In the new construction, the set of self-loops previously defined on one EFSM state,

has to be divided on multiple control states, each belonging to a different EFSM.

5.7 Dynamic Analysis for Duplicate Detection Scope

Finding out the exact value of the locality, or the size of the longest back-edge in communicating concurrent systems is, in fact, *as hard as the problem of model checking for safety properties* itself. It is easy to reduce the problem of model checking for safety properties to the problem of finding the longest back-edge. If the unfolded Kripke structure is just a single cycle, Breadth-First Search is forced to subtract all previous layers to remove the duplicates.

In the last section, we have demonstrated a worst-case upper bound on the longest back-edge that can appear while unfolding an EFSM. One of the main assumptions was that the variables, during the unfolding, are assigned *all* the values from their domains. In real-world scenarios, such a situation hardly appears. Most of the values are never reached and hence, many valuation vectors never appear. The question that arises now is: *how can we exploit only the* legal *values of the variables?* In the following, we will use the notation $\oplus$ to denote the final product of the *asynchronously combined* EFSMs.

Let us first introduce a more refined notion of locality of an implicit graph explored until a layer l.

Definition 5.9 (Locality of a BFS-Layer) *The locality of a Breadth-First Search graph resulted due to an unfolding of an EFSM until a layer l is given as:*

$$\mathcal{L}(p)_l \quad \leq \quad \mathcal{L}(\mathcal{G}(p)) \cdot \left(|\hat{D}_{l-1}(x_1)| \cdot |\hat{D}_{l-1}(x_2)| \cdot \ldots \cdot |\hat{D}_{l-1}(x_k)| \right), \tag{5.13}$$

where k is the number of variables, and for all $1 \leq i \leq k$, $\hat{D}_l(x_i) \subseteq D(x_i)$, is the the set of values taken by the variable x_i until layer l.

Algorithm 5.1 depicts an External Breadth-First Search on model checking graphs involving variables and guards. A state in the Kripke structure consists of two parts: a control state from the EFSM and a valuation vector containing the values of all the variables. A duplicate state will appear only if *both* parts match. The algorithm starts by first analyzing the structure of the n EFSMs and computing the longest back-edge (or the localities) in all the EFSMs. The upper-bound of locality for each layer i is maintained in the variable DDS_l and is updated based on the dynamic behavior of the graph in dealing with the variables. The dynamic growth of the unfolded graph or the Kripke structure is monitored by maintaining all the values assigned to the variables. These values are kept in a set $\hat{D}(x_i)$ for a variable x_i. The first step in this process is to save the variable values in the initial state. We restrict to only a single initial state for the sake of brevity, the extension to multiple initial states is analogous. The loop in Line 2 initializes a domain set for each variable x_i in the layer 0, denoted as $\hat{D}_0(x_i)$.

The Breadth-First exploration starts from the *while* loop in Line 6 by expanding *Open*$(l = 0)$. All the previously seen values of the variables are used to initialize the domain sets in the next layer $\hat{D}_{l+1}(x_i)$. We then iterate on all the states u in layer *Open*(l) to generate the successors v for the next layer *Open*$(l + 1)$. For each successor v, the algorithm individually checks all the k variables for any change. In case the value of the variable x_i in u is different than that in v, i.e., if $u[x_i] \neq v[x_i]$, and $v[x_i]$ has never been seen earlier, $v[x_i]$ is added to the layer domain $\hat{D}_{l+1}(x_i)$ (cf. Line 13). We will refer to this process as *widening the layer*

Algorithm 5.1 External Breadth-First Search with Dynamic Locality Computation

Input: n EFSMs $\{p_1, p_2, \ldots p_n\}$ with $\mathcal{L}(\mathcal{G}(p_1)), \mathcal{L}(\mathcal{G}(p_2)), \ldots \mathcal{L}(\mathcal{G}(p_n))$, as their associated localities. All the EFSMs share a set of k global variables $\{x_1, x_2, \ldots x_k\}$ that can take their values from finite domains $D(x_1), D(x_2), \ldots, D(x_k)$, respectively.

Output: A path from a state in $\mathcal{I}$ to a state in $\mathcal{T}$, if exists, $\emptyset$ otherwise.

1: $DDS_1 \leftarrow \mathcal{L}(\mathcal{G}(\oplus)) \leftarrow \max\{\mathcal{L}(\mathcal{G}(p_1)), \ldots, \mathcal{L}(\mathcal{G}(p_n))\}$
2: **for all** $i \leftarrow 1$ **to** $|k|$ **do** //FOR ALL k VARIABLES
3: $\hat{D}_0(x_i) \leftarrow \{\mathcal{I}[x_i]\}$ //INITIALIZE VARIABLE DOMAINS IN LAYER *Open*(0)
4: **end for**
5: $l \leftarrow 0$ //LAYER INDEX. LAYER *Open*(l) IS EXPANDED INTO *Open*$(l+1)$
6: **while** *Open*$(l) \neq \emptyset$ **do** //LAYER *Open*(l) IS NOT EMPTY
7: $\hat{D}_{l+1}(x_i) \leftarrow \hat{D}_l$ //SAVE THE DOMAINS FROM THE PREVIOUS ITERATION
8: **for all** States $u \in$ *Open*(l) **do** //ITERATE ON ALL STATES IN THE LAYER *Open*(l)
9: **if** $u \in \mathcal{T}$ **return** *Construct-Solution*(u)
10: **for all** States $v \in$ *Succ*(u) **do** //FOR ALL SUCCESSORS v OF u
11: **for all** $i \leftarrow 1$ **to** $|k|$ **do**
12: **if** $u[x_i] \neq v[x_i]$ **and** $v[x_i] \notin \hat{D}_{l+1}(x_i)$ **then** // x_i HAS A NEW VALUE IN v
13: $\hat{D}_{l+1}(x_i) \leftarrow \hat{D}_{l+1}(x_i) \cup \{v[x_i]\}$ //WIDEN THE DOMAIN OF THE VARIABLES x_i
14: **end for**
15: *Open*$(l+1) \leftarrow$ *Open*$(l+1) \cup \{v\}$ //SAVE THE NEW STATE v
16: **end for**
17: **end for**
18: *Open*$(l+1) \leftarrow$ *External-Sort-and-Remove-Duplicates*(*Open*$(l+1)$)
19: **for all** $i \leftarrow 1$ **to** DDS_l **do** //SUBTRACT DDS_l MANY PREVIOUS LAYERS
20: **if** $i - l < 0$ **then** break //SUBTRACT ONLY LEGAL LAYERS
21: *Open*$(l+1) \leftarrow$ *Open*$(l+1) \setminus$ *Open*$(i-l)$
22: **end for**
23: $DDS_{l+2} \leftarrow \mathcal{L}(\mathcal{G}(\oplus)) \cdot (|\hat{D}_{l+1}(x_1)| \cdot |\hat{D}_{l+1}(x_2)| \cdot \ldots \cdot |\hat{D}_{l+1}(x_k)|)$ //NEW LOCALITY FOR *Open*$(l+2)$
24: $l \leftarrow l+1$ //ANOTHER LAYER IS FINISHED
25: **end while**
26: **return** $\emptyset$

domain in the following discussion. After this analysis, the successor v is added to the layer *Open*$(l+1)$ as a potential candidate for expansion.

The process of delayed duplicate detection starts by first using an external sorting routine to sort the layer *Open*$(l+1)$ and simultaneously removing the duplicate states within the layer (cf. Line 18). The locality computed for the last layer is then used for set subtraction. In Line 21, we subtract DDS_l many previous layers from *Open*$(l+1)$. Given that the DDS_l represents the correct upper-bound on the longest back-edge, this phase will generate a duplicate-free sorted sequence consisting of the states that have never been seen in any of the previous layers. The new value for the locality is then computed as a product of the domains' cardinalities for each variable and the structural locality of the product graph $\mathcal{L}(\mathcal{G}(\oplus))$ (cf. Equation 5.13). The new locality will be used during the next iteration of the main *while* loop to restrict the duplicate detection scope.

During the life-time of the algorithm, it is easy to observe that two invariants are maintained:

Invariant 1 *For all variables x_i and layers l*

$$\hat{D}_0(x_i) \subseteq \hat{D}_1(x_i) \subseteq \hat{D}_2(x_i) \subseteq \ldots \subseteq \hat{D}_l(x_i) \subseteq D(x_i),$$

which follows from the loop in Line 11.

Invariant 2 *For all layers l, if $\mathcal{L}(\mathcal{G}(\oplus)) > 0$*

$$\mathcal{L}(\mathcal{G}(\oplus)) \leq DDS_0 \leq DDS_1 \leq \ldots \leq DDS_l \leq \ldots \leq \mathcal{L}(p),$$

which follows from Line 23 and Invariant 1.

With the following theorem, we prove the correctness of Algorithm 5.1:

Theorem 5.6 *For a Breadth-First layer l resulted from the unfolding of an EFSM into a Kripke structure, removing*

$$\mathcal{L}(\mathcal{G}(\oplus)) \cdot \Big(|\hat{D}_{l-1}(x_1)| \cdot |\hat{D}_{l-1}(x_2)| \cdot \ldots \cdot |\hat{D}_{l-1}(x_k)| + 1\Big)$$

many layers guarantees that all states seen in previous layers have been removed from the layer l.

Proof Assume the same worst case unfolding of an EFSM as in the proof of Theorem 5.5, with the exception that only the legal values assigned to the variables through the domains $\hat{D}_{l-1}(x_i)$ for $1 \leq i \leq k$ are considered in the unfolding. Let the product of the partial domains is represented by $\hat{d}_l = \Big(|\hat{D}_{l-1}(x_1)| \cdot |\hat{D}_{l-1}(x_2)| \cdot \ldots \cdot |\hat{D}_{l-1}(x_k)| + 1\Big)$. The edge from q_j to q_i in the Figure 5.7 can only *stretch* by $\hat{d}_l$ in the worst case. This in turn, will generate the new q_i through q_j in a layer that is $\hat{d}_l$ levels below the layer where the control state q_j appeared for the first time. Using the same construction as in the proof of Theorem 5.5, it is easy to show that the duplicate detection scope of layer l is bounded by the product of the structural locality and the product of the partial domains $\hat{d}_l$.

The new values that are added to the variable domains in the layer l, will not effect the locality of the layer l – the new combinations have never appeared before and hence can be completely ignored while calculating $\hat{d}_l$. ■

So far, we have included the full structural locality of the graph in the product. In fact, for a more refined duplicate detection scope, we can dynamically analyze the structure of individual EFSMs too. Let $\mathcal{L}\left(\mathcal{G}(p_i)\right)_{l-1}$ define the locality of the EFSM p_i restricted to only l BFS layers.

Proposition 1 *For a Breadth-First layer l of a Kripke structure resulted from the unfolding of an asynchronous product of n EFSMs $p_1 \ldots p_n$, removing*

$$\max\{\mathcal{L}\left(\mathcal{G}(p_1)\right)_{l-1}, \ldots, \mathcal{L}\left(\mathcal{G}(p_n)\right)_{l-1}\} \cdot \left(|\hat{D}_{l-1}(x_1)| \cdot |\hat{D}_{l-1}(x_2)| \cdot \ldots \cdot |\hat{D}_{l-1}(x_k)| + 1\right)$$

many layers guarantees that all states seen in previous layers have been removed from the layer l.

Proof (Sketch:) None of the guards on the transitions of an EFSM i depend on the active control state of another EFSM j. This observation follows from the construction of the EFSM, where only the variables are allowed to appear in guards. While unfolding a back-edge in the Kripke structure will appear, if and only if, a back-edge in one of the EFSM appears. Hence by keeping track of the longest back-edge encountered so far, and using

$$\max\left\{\mathcal{L}\left(\mathcal{G}(p_1)\right)_{l-1} \ldots \mathcal{L}\left(\mathcal{G}(p_n)\right)_{l-1}\right\}$$

in place of $\mathcal{L}\left(\mathcal{G}(\oplus)\right)$, guarantees that no duplicates will be expanded. ■

Advantages There are two significant advantages for the dynamic locality computation.

1. If the locality found by the structural analysis is already zero, we do not even need to maintain any of the variable domains. Consequently, External BFS or External A* *does not* have to look at previous layers for subtraction during duplicates removal.

2. It is a common observation that not all allowable values to a variable are actually assigned to it. Mostly, only a small subset of values are reachable during the execution of a program. By keeping the partial domains for all the variables and using them for the computation of locality for each layer, we can avoid subtracting a large number of previous layers.

5.8 Weighted Graphs

Up to this point, we have ignored the weights of the graphs. In this section, we generalize our approach to small integer weights w in $\{1, \ldots, C\}$. Note that, in practice, atomic regions, the motivation for using weighted graphs, are few steps that have to be executed without interruption, such as a *test-and-set* assumption. Hence the condition of *bounded integer weights* does not compromise the applicability of External A* on model checking graphs. We will consider the cases of undirected and directed graphs individually in the following subsections.

5.8.1 Weighted and Undirected Graphs

The following lemma gives an upper bound on the duplicate detection scope in the Breadth-First weighted undirected graphs as follows:

Lemma 5.7 (Duplicate Detection in Weighted and Undirected Graphs) *The duplicate detection scope for Breadth-First weighted and undirected graphs is $2C+1$, where C is the maximum edge weight.*

Proof We will prove on the same direction as the proof of correctness of Munagala and Ranade. Assume a state u in layer $g(u)$ and till layer $g(u)$ no state has been expanded twice. Let v be a successor of u such that $w(u,v) = C$, then there can be one of the following three cases:

- Case I: v has appeared in the band of layers $g(u)+1, g(u)+2, \ldots, g(u)+C-1$. This gives a duplicate detection scope of at least C.
- Case II: v appears in the layer $g(u)$, subtracting the layer $g(u)$ should remove the second occurrence of v.
- Case III: v has already appeared in the sequence layers above u, i.e., in $g(u)-1, g(u)-2 \ldots, g(u)-C$. In this case subtracting the band of layer above $g(u)$, which amounts to C.

Summing all of them gives us a duplicate detection scope of $2C$ in undirected Breadth-First search graphs with bounded integer weights. ■

The behavior of weighted and undirected graphs on External A* is similar. Recall the definition of consistent heuristic (c.f. Definition 3.2), which says that a heuristic h is consistent, if and only if, for all $u, v \in \mathcal{S}$ with $v \in \mathit{Succ}(u)$,

$$h(u) - h(v) \leq w(u,v)$$

Since the graph is undirected, considering the inverse graph, we equally have

$$h(v) - h(u) \leq w(v,u).$$

Combining both of the equations give us

$$|h(u) - h(v)| \leq w(u,v). \tag{5.14}$$

Equation 5.14 gives the *range* of the differences between the heuristic value of a state and its successors. It states that the successors of the bucket $\mathit{Open}(g,h)$ are no longer spread across the three buckets: $\mathit{Open}(g+1, h-1)$, $\mathit{Open}(g+1,h)$, and $\mathit{Open}(g+1,h+1)$, but over $3+5+\cdots+2C+1 = C\cdot(C+2)$ buckets in layers $g+1, g+2, \ldots, g+C$, respectively. In Figure 5.8, the range of predecessors and successors of a bucket (grey-shaded) are depicted. The lower triangle covers the buckets where the potential successors can fall. The upper triangle, on the other hand, corresponds to the potential predecessors of the bucket.

The derivation of the I/O complexity is similar to the uniform case; the difference is that each bucket is referred to at most $2C+1$ times for bucket subtraction and expansion.

Theorem 5.8 (I/O Complexity of External A* for Weighted and Undirected Graphs) *Given an implicit, undirected, and a weighted graph with weights being in the integer range $[1, C]$, and a consistent estimate, the I/O complexity of External A* is bounded by*

$$O\left(\mathit{sort}(|\mathcal{R}|) + C \cdot \mathit{scan}(|\mathcal{S}|)\right) \tag{5.15}$$

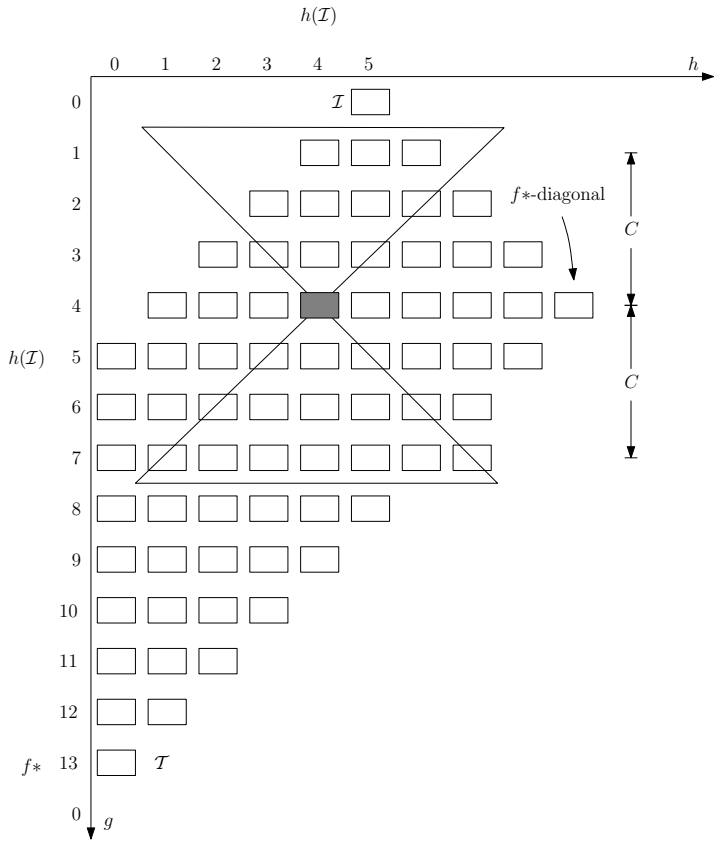

Figure 5.8: Effects of weighted and undirected graphs on the working of External A*. The lower triangle correspond to the range of successors from the grey-shaded bucket. The upper triangle shows the range of predecessors of the same bucket.

Proof Theorem 3.5 bounds the asymptotic number of I/Os of External A* on a uniformly weighted undirected graph to

$$O(sort(|\mathcal{R}|) + scan(|\mathcal{S}|)).$$

There the second term was due to the scanning for expansion and subtraction of previous layers; which were just two, due to the undirected nature of the graph. Lemma 5.7 bounds the duplicate detection scope to $2C$ in weighted undirected graphs. This amounts to

$$O(sort(|\mathcal{R}|) + C \cdot scan(|\mathcal{S}|))$$

as the overall I/O complexity of External A* on weighted undirected graphs. ■

Large Edge Weights: If the range of unique edge weights, the maximum edge weight C, and the f^* value are very large, buckets could become sparse and hence should be handled more carefully. If a bucket has fewer than B states, wasted I/Os might result. This motivates to look at the internal space complexity of our algorithm. Let M be the size of main memory/RAM available in terms of number of states. Assuming that $(f^*)^2 \cdot B = O(M)$, i.e.,

enough main memory is available to keep a small buffer representation of every bucket in the main memory, unnecessary I/Os can be clearly avoided.

The internal memory requirement can be further reduced to $O(C \cdot f^* \cdot B)$ states, by saving only the C layers that change between successive active buckets. The worst case internal space complexity of External A* for weighted and undirected graphs can be bounded to $O(C^2/2 \cdot B)$ states. This space is required to store the buffers of $C^2/2$ buckets in RAM, to which a new successor can belong to.

5.8.2 Weighted and Directed Graphs

We now relax our restriction on bounded weights and extend the duplicate detection scope to unbounded integral weighted and directed graphs. The notion of *weighted locality* is introduced in the following to refer to the locality of weighted Breadth-First Search graphs.

Definition 5.10 (Weighted Locality) *The* weighted locality $\mathcal{L}_c$ *of a directed graph* $\mathcal{G}$ *with a weight function* $c : (u, v) \rightarrow \mathbb{Z}^+$ *is defined as*

$$\mathcal{L}_c(\mathcal{G}) = \max_{u,v \ s.t. v \in \mathrm{Succ}(u)} \{\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + c(u, v)\} \tag{5.16}$$

Lemma 5.9 (Duplicate Detection Scope in Weighted and Directed Graphs) *Given a directed and weighted graph* $\mathcal{G}$*, the number of previous layers that need to be subtracted in a Breadth-First Search to prevent duplicate search effort is equal to the* $\mathcal{L}_c(\mathcal{G})$*.*

Proof Let us consider two nodes u and v, with $v \in \mathit{Succ}(u)$. Assume that u has been expanded for the first time, generating the successor v which has already appeared in the layers $0, \ldots, \delta(\mathcal{I}, u) - \mathcal{L}_c(\mathcal{G})$ implying $\delta(\mathcal{I}, v) \leq \delta(\mathcal{I}, u) - \mathcal{L}_c(\mathcal{G})$. We have

$$\begin{aligned} \mathcal{L}_c(\mathcal{G}) &\geq \delta(\mathcal{I}, u) - \delta(\mathcal{I}, v) + c(u, v) \\ &\geq \delta(\mathcal{I}, u) - (\delta(\mathcal{I}, u) - \mathcal{L}_c(\mathcal{G})) + c(u, v) \\ &= \mathcal{L}_c(\mathcal{G}) + c(u, v) \end{aligned}$$

This is a contradiction to $c(u, v) > 0$. ■

For bounded weights with $C = \max_{u,v \ s.t. v \in \mathit{Succ}(u)}\{c(u, v)\}$, the value of $\mathcal{L}_c(\mathcal{G})$ can be computed to:

$$\mathcal{L}_c(\mathcal{G}) = \max_{u,v \ s.t. v \in \mathit{Succ}(u)} \{\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v)\} + C$$

An upper bound on the weighted locality can be obtained for the case of bounded edge weights, with C being the maximum edge weight.

Theorem 5.10 (Upper-Bound on Locality) *The locality of a weighted graph for breadth-first search can be bounded by the minimal distance to get back from a successor node* v *to* u *(if such a path exists), maximized over all* u*, plus* C*. In other words, with* $v \in \mathrm{Succ}(u)$*, we have*

$$\max_{u \in \mathcal{S}}\{\delta(v, u)\} + C \geq \max_{u \in \mathcal{S}}\{\delta(\mathcal{I}, u) - \delta(\mathcal{I}, v)\} + C$$

Proof For any state $\mathcal{I}, u, v$ in a graph, the triangular property of shortest path states that

$$\delta(\mathcal{I}, u) \leq \delta(\mathcal{I}, v) + \delta(v, u).$$

In particular, for $v \in \mathit{Succ}(u)$, we have

$$\begin{aligned} \delta(v,u) &\geq \delta(\mathcal{I},u) - \delta(\mathcal{I},v), \text{ and} \\ \max_{u\in\mathcal{S}}\{\delta(v,u)\} &\geq \max_{u\in\mathcal{S}}\{\delta(\mathcal{I},u) - \delta(\mathcal{I},v)\} \\ \max_{u\in\mathcal{S}}\{\delta(v,u)\} + C &\geq \max_{u\in\mathcal{S}}\{\delta(\mathcal{I},u) - \delta(\mathcal{I},v)\} + C \end{aligned}$$

■

Another result, we can obtain from the above theorem is about the duplicate detection scope of undirected and weighted graphs:.

Corollary 5.11 *For undirected but weighted graphs, $2 \times C$ is an upper bound on the locality.*

Proof Using Theorem 5.10, the locality $\mathcal{L}_c(\mathcal{G})$ is bounded by

$$\begin{aligned} \mathcal{L}_c(\mathcal{G}) &= C + \max_{u,v\in\mathcal{S}\ s.t. v\in\mathit{Succ}(u)} \delta(v,u) \\ &= C + \max_{u,v\in\mathcal{S}\ s.t. v\in\mathit{Succ}(u)} \delta(u,v) \qquad \textit{(due to the undirected nature of the graphs)} \\ &= 2 \times C \end{aligned}$$

■

Which validates our earlier result on the locality of weighted and undirected graphs.

5.9 Model Checking in Practice

SPIN (Holzmann 1990) is, probably, the most prominent explicit state (enumerative) modelchecking tool. Models are specified in its input language Promela. The language is well-suited to specify communication protocols, but has also been used for a wide range of other verification tasks. The model checker transforms the input into an internal automata representation, which, in turn, is enumerated by its exploration engine. Several efficiency aspects ranging from partial-order reduction to bit-state hashing enhance the exploration process. The parser produces sources that encode states and state transitions in native C code. These are linked together with the validation module to allow exploration of the model. The graphical user interface XSPIN allows to code the model, run the validator, show the internal automata representation, and simulate traces with message sequence charts.

SPIN takes as input a protocol description written in its own specification language called Promela and generates a C language program. This program integrates both the model and the verification code into a single module. In SPIN's terminology these files are referred to as '*pan*' files, that stems from 'protocol analyzer'.

In the following, we discuss some important details on the syntax and semantics of the Promela and how a state in SPIN looks like.

5.9.1 Promela

Promela appears very much like C, but contains many extensions to facilitate the modeling of communication among different processes. A *process* in Promela corresponds to a unique automaton of the system having its own sets of states and transitions. A process is also

allowed to possess a finite set of *local variables*, each of them having a finite domain. Processes communicate with each other through *global variables* and *channels*. A *channel* is an ordered FIFO buffer and can hold a finite number of *messages*, where a *message* is an ordered set of variables. All global variables and channels can be accessed by any process for reading or writing, while the local variables are only accessible by the corresponding process. Promela also provides support for non-determinism through an extended `if-then` clause.

Since each global variable can take its value from a finite domain and all channels have a bounded capacity, the cross-product of all the processes gives us a global finite transition system. The aggregation of states of all processes and the contents of global variables and channels compose the global system state s.

5.9.2 Model Checking in SPIN

The verification of a Promela model is achieved by simulating all possible paths in the model and checking for the error conditions at every state (path, in case of liveness checking). Promela contains six basic statements: assignments, assertions, print statements, send or receive statements and Promela's expression statement. Promela's expressions statements are special form of expressions that are translated into statements that are executable if the expression evaluates to *true*. Each basic executable statement of Promela corresponds to a transition in the system. The execution engine of SPIN takes a tuple (p, t), where p is a non-blocked process and t is an enabled transition in p, and executes the statement corresponding to t to generate a new state s'. This new state is then checked for any kind of property violation such as assertion violations and deadlocks. In case of an error state, SPIN returns an *error trail* that gives us the sequence of statements that led SPIN to the error state starting from the initial state.

SPIN supports many state space optimization techniques including the bit-state hashing (the so-called *super-trace*) and partial-order reduction, where the commutivity of transitions are exploited to prune some part of the search space. SPIN's default search algorithm is depth-first search that prefers the states encountered deeper in the search over the ones encountered earlier. The main drawback of depth-first search is that the length of the error trails is usually very large for human comprehension.

5.9.3 Directed Model Checking in HSF-SPIN

The experimental model checker HSF-SPIN (Edelkamp, Leue, & Lluch Lafuente 2004) is a compatible extension to SPIN. Additionally, it incorporates directed search in explicit state model checking. The tool has been designed to allow different search algorithms by providing a general state expanding subroutine. In its current implementation it provides depth-first and breadth-first search as well as heuristic search algorithms like best-first search, A* and IDA*, and local search algorithms like hill-climbing and genetic algorithms. Partial order and symmetry reduction have been successfully combined with this heuristic search (Lluch Lafuente, Leue, & Edelkamp 2002; Lluch Lafuente 2003b). HSF-SPIN can handle a significant fraction of Promela and deals with the same input and output formats as SPIN. Heuristic search in HSF-SPIN combines positively with automated abstractions in form of abstraction databases.

The Promela language scope of HSF-SPIN is not as large as the current version of SPIN[‡]

[‡]The SPIN code we started with was SPIN 3.4

as it lacks some features like fully dynamic process creation and embedded c-code, but sufficiently strong even for larger models that we have in our benchmark set. For the comparison with our approach we have used the most recent version of SPIN available at the time of this writing, namely SPIN v4.2.4. For SPIN, the reported results corresponds to SPIN's invocation with iterative depth-first search where a longer error path is shortened iteratively. This step was necessary as SPIN tends to return very long counterexamples, due to Depth-First Search.

All heuristic functions return a positive integer value for a given state. Some of them profit from information gathered before the verification starts. For example, the FSM distance estimate requires to run the all-pairs shortest path algorithm on the state transition graph of each process type. On the other hand, the deadlock inferring approach allows the user to determine explicitly which states have to be considered as potentially blocking by labeling statements in the model.

5.9.4 External Model Checker

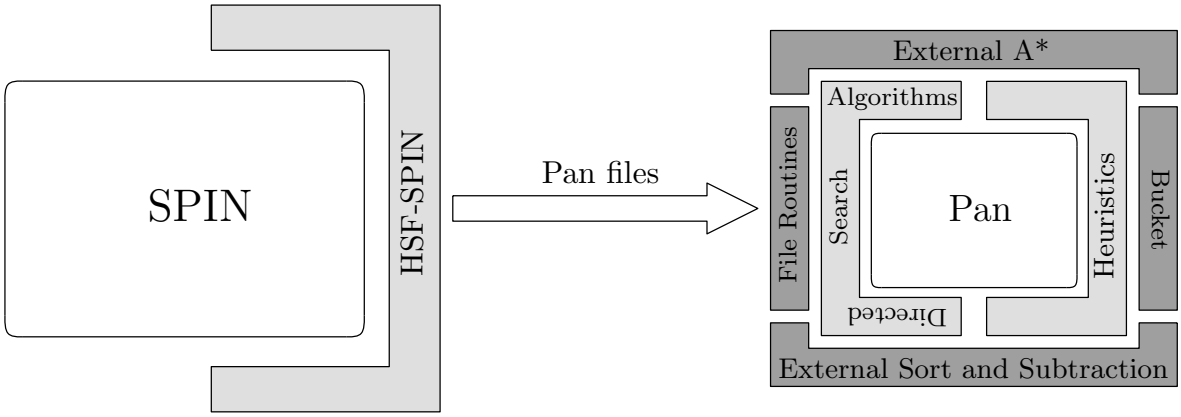

Figure 5.9: Architecture of IO-HSF-SPIN. Dark-grey area depicts the IO-HSF-SPIN and Light-grey area depicts the HSF-SPIN.

The external exploration algorithms, except the locality computation routines, have been implemented in the experimental model checker IO-HSF-SPIN. As an extension of HSF-SPIN, it allows us to use the same heuristics implementations as were available in HSF-SPIN. A block diagram is shown in Figure 5.9. HSF-SPIN works as a wrapper on SPIN. Given a promela model, the *pan* files are generated along with special functions for HSF-SPIN. These files are then compiled with the heuristic functions and special search routines for Directed model checking. The implementation of External A* and file handling functions is also done as a wrapper on HSF-SPIN.

An external approach requires that if a state is flushed to the disk and then read again, the algorithm should be able to work on it in the same way as on a state that was never removed from the memory. Since SPIN is highly optimized for speed, it uses many global variables. Consequently, not only the state itself but much of the meta information about the global variables also had to be flushed to the disk. The result is an implementation that provides the facility of *pause-and-resume*, i.e., the exploration can be canceled at any point of

time and be resumed from the same point where it was canceled. An important advantage of this facility is when the hard disk gets full and the system has to be shutdown to add more hard disks or to restart the exploration on a bigger machine.

5.10 Experiments

The experiments are conducted on a dual processor AMD Opteron DP 250 (2.4 GHz) system with 4 Gigabytes of main memory running Linux operating system. The storage space is shared through a File server connected via NFS. The compiler used is GCC 3.3.6 with optimizations for 64-Bit system.

For each experiment we report, the solution depth, number of stored nodes and number of transitions. Time is an important resource and we report two different kinds of it. The first one, which is the *CPU* time corresponds to the total amount of CPU time consumed by the process, while the second one *real* time corresponds to the total *wall time* that includes waiting for the resources to become available and all I/O operations. Space consumption is also divided into two types: *RAM*, which corresponds to the maximum process size in the internal memory, and *Hard disk*, which reports the total space consumed on the hard disk.

Note that the results in this section may slightly differ to the ones presented in the earlier publications (Jabbar & Edelkamp 2005; Edelkamp & Jabbar 2006c). We have made several refinements to the external exploration. One of the major changes is the replacement of the internal buffers corresponding to buckets with small hash tables. The generated states are inserted into the hash table, and when it gets full, it is flushed to the hard disk. The advantage is that we can perform some earlier internal memory duplicate detection on a smaller set of states. The sorting order is defined on the lexicographical ordering of the hash values and the state vector. Hence changing the hash table size changes the order of a state in the final sorted order. The main advantage of having hash tables in buckets is that it produces small files as apposed to the naïve approach where nodes are added to a linear array and are sorted and flushed to the disk when it is full.

For analyzing safety properties, we chose three classical and challenging protocol models namely, optical telegraph, CORBA-GIOP, and dining philosophers. The property to search for is the *deadlock* property. We employed the number of active processes as the heuristics to guide the exploration.

N	M	Depth	Stored	Transitions	Time		Space in Gigabytes	
					CPU	Real	RAM	Hard disk
2	1	58	52,417	130,886	1s	2s	0.113	0.0283
3	1	70	893,393	2,484,427	33s	3m 10s	0.129	0.551
4	1	75	7,929,712	23,395,633	7m 39s	51m 33s	0.209	5.103
2	2	64	203,546	511,323	6s	16s	0.125	0.137
3	2	76	3,431,615	9,431,435	2m 52s	14m 10s	0.166	2.453
4	2	81	30,504,622	87,905,795	36m 49s	1h 59m 31s	0.682	22.11

Table 5.1: Deadlock Detection in CORBA - GIOP. Solvable with SPIN v4.2.4. N represents the number of users, and M the number of servers in the model.

CORBA - GIOP (Kamel & Leue 2000) model takes two main parameters namely: the

number of users and the number of servers with a range restriction of 1 to 4 on users and 1 to 2 on servers. We have been able to solve all the instances of the GIOP model especially the configuration with 4 users and 2 servers which requires a storage space of 20.7 gigabytes. The results are shown in Table 5.1. The algorithm scales up very fast due to the high branching factor in the state space graph. All those instances are also solvable with iterative Nested DFS in SPIN v4.2.4.

Philo.	Depth	Stored	Transitions	Time		Space in Gigabytes	
				CPU	Real	RAM	Hard disk
50	202	120,154	124,905	1s	17s	0.0956	0.126
100	402	980,304	999,805	29s	6m 11s	0.233	2.082
150	603	3,330,454	3,374,705	2m 32s	24m 17s	0.428	10.686
200	802	7,920,604	7,999,605	9m 52s	37m 39s	0.692	34.138
250	1002	15,500,754	15,624,505	31m 09s	1h 28m 43s	1.129	83.345

Table 5.2: Deadlock Detection in Dining Philosophers with IO-HSF-SPIN. NOT solvable with SPIN v4.2.4.

Table 5.2 presents the results for the deadlock detection in different instances of dining philosophers problem. The first column denotes the number of philosophers used for each instance. We have been able to solve 250 philosophers having a space requirement of 83.345 gigabytes of hard disk space while utilizing a mere 1.129 gigabytes of RAM. SPIN v4.2.4 could not solve any of the problem in 1 hour time and 4 gigabyte process size constraints. It consumed the whole memory in the first few minutes and then started to swap to the virtual memory.

The bottleneck in the dining philosopher's problem is not only the combinatorial explosion in the number of states but also the size of the states. As can be observed in the last column depicting the space requirement, the space requirement grows with a significant rate easily reaching even the address limits of most modern micro computers.

Sta.	Depth	Stored	Transitions	Time		Space in Gigabytes	
				CPU	Real	RAM	Hard disk
6	38	19,039	38,344	1s	2s	0.0556	0.00495
7	45	333,877	820,348	9s	29s	0.0784	0.133
8	50	420,531	917,044	10s	35s	0.693	0.193
9	57	9,186,611	23,582,956	14m 58s	47m 52s	0.333	4.76
14	86	3,901,184,954	5,076,172,614	173h 08m	479h 46m	3.073	3,006.335

Table 5.3: Deadlock Detection in Optical Telegraph.

The third domain is the optical telegraph model. We conducted experiments with different number of stations. The results are presented in Table 5.3. For all the runs in this table, we fixed the size of the internal buffer to 1003 elements. We have been able to solve the optimal deadlock problem for upto 14 stations. For 14 stations, the total memory consumed is about *3 Terabytes*. SPIN v4.2.4, with iterative depth-first search, can also solve problems with 6, 7, 8, and 9 stations in a couple of minutes, but with 14 stations, it ran out of memory.

5.11 Related Work in External Memory Safety Model Checking

In the last few years, there has been a growing interest in using secondary storage mediums for model checking. In the following, we discuss the efforts done by other groups in this field.

5.11.1 In FDR

One of the first approaches toward using disk-based search algorithms in model checking was proposed by Roscoe (1994) in the context of model checking of communicating sequential processes through FDR. It employed a disk-based Breadth-First Search similar to the ones we have seen earlier. A disk-based queue *Open* is maintained, where new successors are saved. Once a BFS level is completely expanded, the *Open* is externally sorted and subtracted from the *Closed* set, which itself is maintained as a sorted set. The previous BFS level that is just expanded, is then merged with the *Closed* set through an external merge procedure. The authors did not provide any I/O complexity results. Based on the analyses techniques developed in this dissertation, we can bound the I/O complexity of the algorithm by

$$O\left(sort(|\mathcal{R}|) + d \cdot scan(|\mathcal{S}|)\right) \text{ I/Os},$$

where d is the length of the counter example.

5.11.2 In Murϕ

Stern & Dill (1998) put forward another disk-based search algorithm in the context of the Murϕ validation tool. They employed a disk-based search that follows a Breadth-First order for state expansion, but the method to save the *Closed* list was different than what we have seen so far. The algorithm maintains three sets: *Open* as a FIFO queue (in RAM), a hash table HT (in RAM), and a set on visited states *Visited* (on disk). The *Visited* state set is different to the *Closed* state set. It contains all the states that have reached during the exploration, as apposed to the expanded states in *Closed* set. New states are inserted in HT, until it is full. Once full, a full scan over the entire *Visited* set is performed for duplicates removal. It removes all the states from HT that exist on the disk. The rest of the states are appended to the *Visited* set and enqueued in the *Open* queue.

Using our techniques for the analysis of implicit graphs, we can say that the disk-based search in Murϕ can produce

$$O\left(\frac{|\mathcal{R}|}{M} \cdot scan(|\mathcal{S}|)\right) \text{ I/Os}$$

in the worst case. The algorithm can be regarded as doing an '*Early disk-based Duplicate Detection*', instead of a *Delayed Duplicate Detection*. Moreover, it cannot handle the *Open* queue larger than the main memory capacity.

This Early Duplicate Detection approach is also employed by Barnat, Brim, & Šimeček(2007) in external liveness checking.

Another effort toward externalization of Murϕ verifier is due to Penna *et al.* (2002). They extended the disk-based Breadth-First search approach by Stern & Dill (1998) through a more restricted duplicate detection scope. Their proposal is based on a similar concept of locality, as we have seen earlier in this chapter. The authors claimed that with high probability, the

locality is bounded, i.e., the successor states do not fall beyond *few* layers – a result that is due to Tronci *et al.*(2001), where it was claimed that in most of the network protocols, it is sufficient to use only 75% of the *Visited* list for duplicate detection. A novel aspect in this disk-based BFS approach is that the duplicate detection scope can be changed on-the-fly if it appears that only few states are deleted from the *Open* queue.

The algorithm is not optimal-efficient – states can be expanded several times until the error state is reached. Moreover, the algorithm also faces a non-termination problem, when the search leaks back to the layers that were not used for the duplicate detection.

Penna *et al.* (2003), have combined the internal BFS and disk-based BFS in Murϕ. The model checker can then be started to perform an internal BFS. Only when the memory requirement increases beyond the available RAM, the *Visited* set, the *Open* queue, and the status of the internal BFS are written to the disk. An external BFS then takes control of the subsequent exploration.

Bao & Jones (2005) presented another variant of Murϕ Verifier with magnetic disk. They also targeted to reduce the duplicate detection efforts. The proposal was to use a partitioning function to divide the state space into memory sized partitions.

Each partition is mapped to a hash table in the main memory and an external FIFO queue that is flushed to the disk when full. Each successor is fed into the partition function that decides the queue to which it should be added. Duplicate detection is delayed when the queue is selected for expansion. At that time, the hash table is read into the main memory and the the states from both the disk queue and the memory queue are filtered for duplicates through the hash table. Once all states from a particular queue are read, checked for duplicates and if necessary expanded, the next longest queue is selected for expansion.

The algorithm of (Bao & Jones 2005) requires a partitioning function that can divide the state space in memory-sized partitions. Coming up with such a partitioning function is a difficult problem itself. On the other hand, a fine grained function that divides the state space into many small partitions can significantly degrade the performance of the algorithm due to excessive I/Os.

5.11.3 In Petri nets

Christensen, Kristensen, & Mailund (2001) presented a memory efficient approach for the analysis of Petri nets. They proposed to arrange the states in a plane wrt. a *progress measure*, which is induced from a given partial order over the states. Repeated scans are performed over the entire search space using a sweep-line method. The scan over the entire state space is memory efficient, as it only needs to store the states that participate in the transitions that cross the current sweep-line position. These states are marked visited and the scan over the entire state space is repeated to cope with states that are not reachable with respect to earlier scans.

Since the *Closed* states are not saved, solution reconstruction once an error has been established, is impossible. In (Kristensen & Mailund 2003), the authors suggest to keep the *Closed* set on the hard disk as a spanning tree. Solution reconstruction then reduces to climbing up the tree saved on the hard disk. Their dependency on a good *progress measure* hinders its applicability to model checking in general. They have applied it mainly to Petri nets based model checking where the notion of *time* is used as a *progress measure*. Moreover, the approach is very sensitive to the back edges that can lead a search to leak-back in the *Closed* states. All the back edges encountered during the scan are inserted into a queue. Once the

scan is finished, for each back edge, a new scan is performed. For k back edges, the time complexity can then be bounded by $k.|\mathcal{S}|$. Since the disk is only used to keep the states for solution reconstruction, and not for any kind of duplicate detection, the issue of I/O complexity does not arise here.

5.11.4 On Promela Models

In the approach of Hammer & Weber (2006), a scheme based on an under-approximation of the hash table is proposed. During the search, the *Closed* list is inserted into an internal memory chained hash table. When the hash table reaches a specified capacity, some of the states are flushed to the disk freeing empty space for new states. Since, it is very I/O extensive to read/write single states, a *reclaim* set amounting to about 10% of the states in the hash table is maintained. Each state is assigned a value that indicates how likely a revisit to this state is. Calculating the k-median and selecting all elements lesser than the k-median compose the reclaim set.

A new state is first checked in a Bloom filter cache. A bloom filter is a bit array of size m with l hash functions $h_1, h_2, \ldots, h_l$. A state s is considered marked if the bits

$$\langle h_1(s) \text{mod } m, h_2(s) \text{mod } m, \ldots h_l(s) \text{mod } m \rangle$$

are set. Since it provides an over-approximation of the state space, a *miss* in the Bloom filter is sufficient to declare a state as unvisited, while a *hit* has to be checked against the hash table.

The second step to catch a duplicate state is to check it in the hash table. There, a *hit* is sufficient to say that the state is already visited, but in case of a *miss* we are not sure if it is really visited or not as the state might have been flushed to the disk in the memory reclaiming process. Again, to avoid I/Os for single states, a candidate set is maintained in the form of a small hash table. The states from the *reclaim* set that are flushed to the disk are kept in partitions that are loaded one-by-one and the candidate set is checked against them, removing duplicates.

The algorithm differs from the earlier approaches in the regard that the size of the *Open* set can also grow beyond the main memory capacity. The authors suggest to keep only a small portion in the memory and to flush the rest to the disk. The algorithm does not follow any strict exploration strategy.

The algorithm has been implemented in the model check CMC (Component-based Model Checker). CMC utilizes a Promela interpreter and state space generator called NIPS (Weber 2007). The I/O time has been further reduced by using Lempel-Ziv LZ77 algorithm through ZLIB library. For states with large sizes, the compression resulted in considerable savings. No I/O complexity results were provided by the authors.

5.12 Summary

With this chapter, we have contributed the first theoretical and analytical study of I/O efficient on-the-fly directed model checking for LTL safety properties. External A* is extended to directed and weighted graphs to cope with the model checking graphs. Through a combination of static analysis of the user-provided concurrent system model and a dynamic

analysis of the on-the-fly generated state space structure, a novel method for finding out the duplicate detection scope of model checking graphs is presented.

We have also seen a very first attempt to bound the number of previous layers that need to be checked for duplicates removal in the context of model checking graphs. The bound is changed dynamically by analyzing the values assigned to the variables.

We have integrated External A* for directed and weighted graphs, through a non-trivial extension of state-of-the-art model checker SPIN. The results on challenging benchmark protocols show that the external algorithms reduce the memory consumption, are sufficiently fast in practice, and have some further advantages by using a different node expansion order. The largest exploration we have been able to conduct took 3 Terabytes on the hard disk while running for 20 days.

Earlier disk-based approaches are mostly based on doing a simple Breadth-First Search, where all previous layers are merged into one single layer or where a small percentage of the last layers are checked for duplicates removal. They do not allow one to use any form of guidance to search errors faster. By using a heuristic function that always return zero, External A* can simulate External BFS I/O efficiently. Another contribution of this chapter is the I/O complexity analysis of some of the earlier disk-based model checking approaches.

There are two time expensive operations in external model checking: delayed duplicate detection and generation of states. In (Edelkamp & Jabbar 2005), we propose a *semi-external algorithm* in which internal memory bit-state hashing (supertrace in SPIN) (Holzmann 1998) is combined with External A*. The experiments have shown a significant reduction in some models where duplicate detection was the major bottleneck. For other models, where time was mostly consumed in the generation of states, it has shown only a marginal or no improvement at all. In Chapter 7, we will study a distributed version of External A* that has shown to be successful in cutting down the overall time, especially the state generation time, by utilizing a network of workstations.

Chapter 6

Liveness Properties

In this chapter, we alleviate the scope of external search from reachability analysis to *accepting cycle detection* to verify liveness properties. Unlike *safety* properties that require the reachability of a particular error state, falsification of *liveness* properties require an infinite path in the form of a 'lasso' that violates the property.

The best known internal memory algorithm is the Nested Depth-First Search (cf. Chapter 4). The main challenge for external on-the-fly LTL model checking is that the Depth-First traversal of the global state space graph and similarly Tarjan's algorithm (Tarjan 1972) is not efficient. All attempts to solve this problem via variants of Breadth-First Search (Brim *et al.* 2004; Barnat, Brim, & Chaloupka 2003; Černá & Pelánek 2003a) arrive at a time complexity that is non-linear in the size of the model. The approach we propose in this chapter is based on a translation procedure of liveness problems into safety problems (Schuppan & Biere 2004). The translation also includes a rich state description which allows the expression of lower bounds for cost-optimal guided search.

Structure of the chapter: First, we refresh some of the notations and conventions that are either new or borrowed from the previous chapters. This is followed by a discussion about infinite paths in the context of liveness checking. Then, we discuss the problems with externalization of Nested Depth-First Search. Our solution is based on the reduction of liveness to safety approach by Schuppan and Biere. We will take a closer look at their approach in the next section. To extend the internal memory algorithm of Schuppan and Biere for external memory, we will start with External Breadth-First Search and analyze its I/O complexity. For guidance and to be able to apply External A* for checking liveness properties, we then propose a set of heuristic functions. Utilizing such estimates, External A* is then extended to a cycle detection algorithm.

Notations: For the sake of brevity, we use $\mathcal{G}$ as a symbol denoting the underlying cross product of the model $\mathcal{M}$ and the negated LTL property $\neg\phi$, both represented as Büchi automata. The implicit graph $\mathcal{G}$ consists of states and transitions represented by $\mathcal{S}$ and $\mathcal{R}$, respectively, and has $\mathcal{I}$ and $\mathcal{T}$ representing the set of initial states and the set of final/accepting states, respectively. Following Spin's standard, we use the term *never-claim* synonymously for the Büchi automaton of the negated property and denote it by $\mathcal{B}(\neg\phi)$. An extended finite-state machine (EFSM) is a FSM annotated with guards and actions defined on a set of finite

domain variables, which makes liveness checking decidable. We furthermore assume that N EFSMs are combined *asynchronously* to form the model. The Büchi automaton of the model is then *synchronized* with the $\mathcal{B}(\neg\phi)$. The set of accepting states $\mathcal{T}$ consists of all the states in the cross product where a state of the model is synchronized with an accepting state of $\mathcal{B}(\neg\phi)$.

Falsification of a given liveness property ϕ requires a lasso-shaped path that starts from one of the initial states $i \in \mathcal{I}$ and visits at least one accepting state $t \in \mathcal{T}$ within the loop of the lasso.

In the context of external heuristic search with External A*, a *bucket* is a set of states identified by $Open(g, h)$, where g and h respectively represent the depth and the heuristic estimate common to all the states in the bucket. For Depth-First Search, the acronym DFS is utilized; BFS is used for Breadth-First Search.

6.1 Explicit-State Model Checking

In this section, we do a brief review of automata-based LTL model checking. In automata-based model checking for LTL properties, both the model to be analyzed and the specification to be checked are modeled as non-deterministic *Büchi automata*. Syntactically, Büchi automata are finite-state automata, but semantically they differ from finite-state automata when it comes to the acceptance condition. Büchi automata are defined on infinite words or runs. Let π be a run and $inf(\pi)$ be the set of states reached infinitely often in π. A Büchi automaton accepts π, if the intersection between $inf(\pi)$ and the set of final states $\mathcal{F}$ is not empty. In the context of LTL model checking, it identifies an accepting cycle in the cross product and hence corresponds to the violation of a liveness property.

The best known on-the-fly algorithm for finding such *accepting cycles* is the Nested Depth-First Search (cf. Chapter 4). The algorithm employs DFS to first look at an accepting state and once it is found, starts another DFS to look for a state on the *stack* of the first search. The resulting path is a lasso that starts from an initial state, visits at least one accepting state and closes the loop by visiting a state on the path from the initial state to the first accepting state.

6.2 Problems with Externalizing Depth-First Search

DFS relies on a stack data structure. For performing a DFS on external memory, we need an external stack data structure. A stack can be simulated on external memory by using an internal memory array of size $2B$ elements (Chiang *et al.* 1995). At any time, the array contains the $k < 2B$ elements most recently inserted. We assume that the stack contents are bounded by at most N elements. A *pop* operation incurs no I/O, except for the case where the buffer has run empty, where $O(1)$ I/Os to retrieve a block of B elements are sufficient. A *push* operation incurs no I/O, except for the case where the buffer has run full, where $O(1)$ I/Os are required to write a block of B elements. The rationale behind a stack of size $2B$ is apparent as we can thus avoid the oscillation in I/Os for alternating push and pop operations.

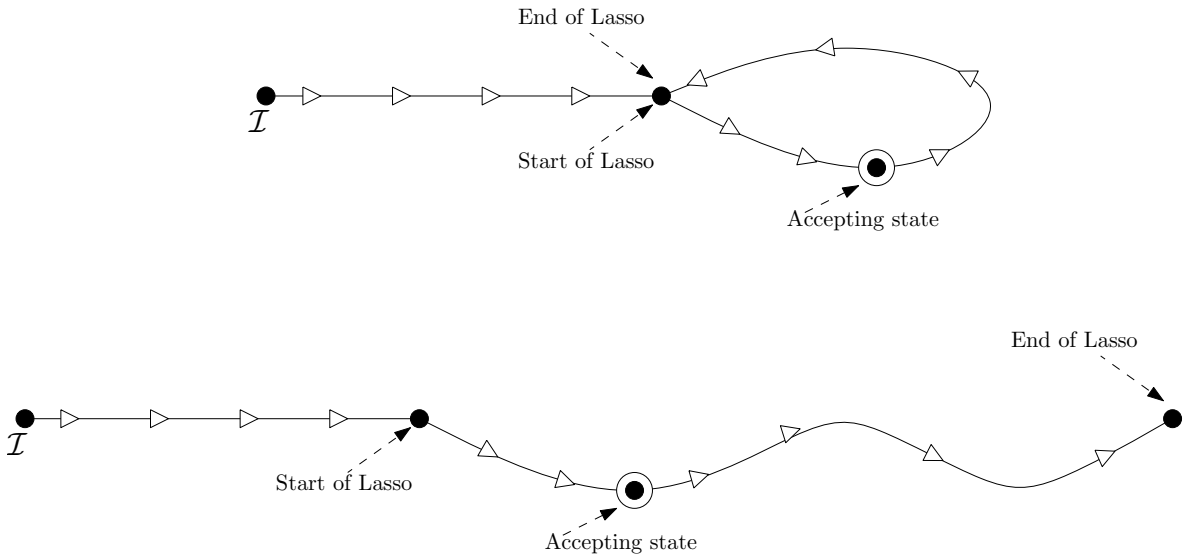

Figure 6.1: Accepting cycle detection. A typical lasso-shaped path (top) and an unfolded lasso due to the reduction of liveness to safety analysis.

I/O Complexity of External Depth-First Search: The I/O complexity for external DFS for explicit graphs has been shown to be bounded by

$$O\left(|\mathcal{S}| + \frac{|\mathcal{S}|}{M} \cdot \mathit{scan}(|\mathcal{R}|)\right)$$

by (Chiang *et al.* 1995). Even in undirected graphs, this is considerably more expensive than external BFS. The reason lies in the fact that DFS exploits less locality during the search. The run-time analysis is as follows. There are $|\mathcal{S}|/M$ stages where the internal buffer for the visited states' set becomes full. The buffer is then flushed and duplicates are eliminated from the external adjacency list representation by a file scan. Visited successors in the unexplored adjacency lists are marked not to be generated again, so that all such states in the internal visited list can be eliminated. As with external BFS in explicit graphs, $O(|\mathcal{S}|)$ I/Os in the above complexity result are due to the unstructured access to the external adjacency list. Computing strongly connected components (SCCs) in explicit graphs has the same I/O complexity as DFS, i.e., $O(|\mathcal{S}| + \frac{|\mathcal{S}|}{M} \cdot \mathit{scan}(|\mathcal{R}|))$.

For implicit graphs, as generated for model checking, no access to an external adjacency list is needed, which gives some hopes of finding a better complexity result for these graphs. Removing the term $O(|\mathcal{S}|)$ as with external BFS, however, is a challenge. The major problem for external DFS exploration in implicit graphs is that unseen adjacencies cannot be modeled and there is no time for performing delayed duplicate detection.

6.3 Liveness as Safety

We decided to adapt the *liveness as safety* model checking approach by Schuppan & Biere (2004). Cycle detection for the falsification of a liveness property aims at searching for a

lasso-shaped path that visits at least one accepting state of the Büchi automaton representing the negation of the property. Figure 6.1 (top) depicts a lasso-shaped path. Starting from an initial state, it visits a accepting state and then closes the loop by visiting a state along the path to the accepting state. The state where the loop meets the stem of the lasso, is referred to as a *seed* state.

The liveness as safety (LaS) approach, instead, reduces the problem to a reachability analysis problem, where the head of the lasso is looked for again. The bottom part of Figure 6.1 illustrates the main concept in LaS. The reduction requires roughly doubling the state vector and *guessing* the seed of a liveness cycle. More precisely, a richer state description is used during the search:

$$\langle u, \hat{u}, lo, live \rangle.$$

With every state u, a previously seen state $\hat{u}$ together with two flags *lo* (lasso) and *live* are saved. The first flag is set to prevent future overwriting of the stored state. It can take three values: *st* (stem) indicating that a copy is not saved, *lb* indicating that a loop has begun, and *lc* when the loop is closed. The second flag *live*, a boolean, indicates whether an accepting state is visited (`true`) or not (`false`).

The extended state space consisting of the search state, the copied state and the two flags is referred to as $\mathcal{S}'$. Schuppan & Biere (2005) showed that for fairness constraints of the form $\mathbf{F}\phi$,

$$\pi = (u_0 \dots u_{l-1})(u_l \dots u_{k-1})^\omega$$

is a run in the state space $\mathcal{S}$, if and only if,

$$\pi' = (u_0, \hat{u}_0, 0, 0) \dots (u_{l-1}, \hat{u}_{l-1}, 0, 0)((u_l, \hat{u}_l, 1, 0) \dots (u_{k-1}, \hat{u}_l, 1, 0))^\omega (u_l, \hat{u}_l, 1, 1)$$

is a run in the extended state space $\mathcal{S}'$.

Theorem 6.1 (Time and Space Complexity of Liveness as Safety (Schuppan 2006)) *The accepting cycle can be found by using liveness as safety reduction with a time complexity of* $O(|\mathcal{S}| \cdot (|\mathcal{S}| + |\mathcal{R}|))$ *while using* $O(|\mathcal{S}^2|)$ *space.*

Proof If $\mathcal{S}$ and $\mathcal{R}$ are the sets of states and transitions of the synchronous product of the model and the automaton of the negated property, then $\mathcal{S}$ is searched at most $|\mathcal{S}|$ times, yielding a time complexity of

$$O(|\mathcal{S}| \cdot (|\mathcal{S}| + |\mathcal{R}|)).$$

Since the approach relies on *guessing* the beginning of the lasso, which in the worst case could be every state, the extended state space $\mathcal{S}'$ can be quadratic in the worst case. Moreover, the flag *lasso* can take 3 values while *live* flag can take 2 values. This gives us an accumulated space complexity of LaS:

$$|\mathcal{S}'| \quad = \quad 3 \cdot 2 \cdot |\mathcal{S}| \cdot |\mathcal{S}| = O(|\mathcal{S}^2|).$$

■

An important observation is that, based on this extension, the exploration algorithms themselves do not (or only in a minor way) have to be changed. For example, in (Schuppan & Biere 2004) the authors show how to extend models using so-called observers and applying the same model checkers designed for safety checking.

6.4 External Breadth-First Search For Liveness Checking

Instead of having any state as a candidate for being a *seed* state, we suggest to use only *accepting* states as the seed states. Furthermore, we work with a slightly different state representation. The whole search for a lasso-shaped path is divided into two phases: Primary and Secondary. In the *primary* phase, a path to an accepting state (seed state) is sought after, while in the *secondary* phase, we try to close the lasso by searching for the same accepting state again.

Encoding: We define a state in the extended state space as a tuple

$$\langle u, \hat{u}, \mathit{stage} \rangle.$$

The first component is a state in the *original state* space generated through an expansion. The second component is the *accepting state* that we try to reach again in order to close the cycle. The third component *stage* is a boolean flag that takes on two values. When the tuple belongs to *primary search*, *stage* is set to `false`; for *secondary search* it is set to `true`. Moreover, for primary search, the first two components are always equal.

The Algorithm: The External Breadth-First Search for accepting cycle detection is shown in Algorithm 6.1. The algorithm proceeds by expanding the layer $\mathit{Open}(l)$ and generating the successors in the layer $\mathit{Open}(l+1)$. While expanding a state $\langle u, \hat{u}, \mathit{stage} \rangle$ (cf. Line 9) , it checks whether the tuple belongs to the primary search or the secondary search. If it belongs to the secondary search, i.e., it is trying to close the lasso, the new successor v is checked against the saved accepting state $\hat{u}$. In case of a match, the solution path is reconstructed and returned. On the other hand, if $v \neq \hat{u}$, the search is continued by inserting a new extended state $\langle v, \hat{u}, \texttt{true} \rangle$ in the next layer $\mathit{Open}(l+1)$.

In case of a primary search, i.e., the search for an accepting state, the new successor v is checked for potential inclusion in the set of accepting states $\mathcal{T}$. An affirmative inclusion test, spawns two children: $\langle v, v, \texttt{true} \rangle$ for starting a secondary search, and $\langle v, v, \texttt{false} \rangle$ for keeping the primary search continued. The rationale behind keeping the primary search continued is that v might not belong to the accepting cycle and we would then have to find another accepting state on the same path.

Once all the states of layer $\mathit{Open}(l)$ are expanded, the duplicate removal for layer $\mathit{Open}(l+1)$ is started. The first step is to sort the layer with an external sorting routine and remove all the states that appear twice in that layer. In the second step, $\mathcal{L}(\mathcal{G})$ previous layers are subtracted from $\mathit{Open}(l+1)$ to obtain a file that contains only the states that have never been expanded before. Here $\mathcal{L}(\mathcal{G})$ represents the locality of the graph $\mathcal{G}$ obtained by the cross product. The cycle of expansion, sorting, and subtraction continues, either until an acceptance cycle has been found or until the state space is exhausted, i.e., $\mathit{Open}(l) = \emptyset$.

I/O Complexity The I/O complexity of External Breadth-First search for liveness checking depends largely on the number of accepting states in the product automaton of the model and the negated LTL property. For each of the accepting states, we spawn a secondary search.

Algorithm 6.1 External Breadth-First Search for Accepting Cycle Detection in LTL Model Checking

Input: $\mathcal{I}$: The set of initial states
Output: A lasso shaped path from $\mathcal{I}$ that visits a state $s \in \mathcal{T}$, if one exists, $\emptyset$ otherwise.

1: **for all** $u \in \mathcal{I}$ **do**
2: $\mathit{Open}(0) \leftarrow \{\langle u, u, \texttt{false}\rangle\}$ //ALL STATES MUST BE INSERTED FOR PRIMARY SEARCH
3: **if** $u \in \mathcal{T}$ **then** //IF THE SEARCH IS STARTING FROM AN ACCEPTING STATE
4: $\mathit{Open}(0) \leftarrow \{\langle u, \hat{u}, \texttt{true}\rangle\}$ //A STATE FOR THE SECONDARY SEARCH
5: **end for**
6: $l \leftarrow 0$
7: **while** $(l \neq \infty)$ **do** //FOR ALL NON-EMPTY LAYERS l
8: **for all** states $\langle u, \hat{u}, \mathit{stage}\rangle \in \mathit{Open}(l)$ **do**
9: **for all** States $v \in \mathit{Succ}(u)$ **do** //FOR ALL SUCCESSORS v OF u
10: **if** $\mathit{stage} = \texttt{true}$ **then** //– SECONDARY SEARCH –
11: **if** $v = \hat{u}$ **then** //LOOP CLOSED
12: **return** *Construct-Solution*$(\langle v, \hat{u}, \mathit{stage}\rangle)$
13: **else**
14: $\hat{v} \leftarrow \hat{u}$ //SAVE THE ACCEPTING STATE FROM THE PREVIOUS STATE
15: $\mathit{Open}(l+1) \leftarrow \mathit{Open}(l+1) \cup \{\langle v, \hat{v}, \texttt{true}\rangle\}$ //CONTINUE THE SECONDARY SEARCH VIA THE NEW STATE
16: **else** //– PRIMARY SEARCH –
17: **if** $v \in \mathcal{T}$ **then** //AN ACCEPTING/FINAL STATE IS REACHED
18: $\mathit{Open}(l+1) \leftarrow \mathit{Open}(l+1) \cup \{\langle v, v, \texttt{true}\rangle, \langle v, v, \texttt{false}\rangle\}$ //SPAWN TWO CHILDREN: ONE FOR CONTINUING THE PRIMARY SEARCH AND ONE FOR THE START OF A SECONDARY SEARCH
19: **else**
20: $\mathit{Open}(l+1) \leftarrow \mathit{Open}(l+1) \cup \{\langle v, v, \texttt{false}\rangle\}$ //CONTINUE THE PRIMARY SEARCH VIA THE NEW STATE
21: **end for**
22: **end for**
23: $\mathit{Open}(l+1) \leftarrow \mathit{sort\text{-}and\text{-}remove\text{-}duplicates}(\mathit{Open}(l+1))$
24: **for** $i \leftarrow 1$ **to** $\mathcal{L}(\mathcal{G})$ **do** //REMOVE DUPLICATES FROM $\mathcal{L}(\mathcal{G})$ LAYERS
25: **if** $i - l < 0$ **then** break //SUBTRACT ONLY LEGAL LAYERS
26: $\mathit{Open}(l+1) \leftarrow \mathit{Open}(l+1) \setminus \mathit{Open}(l+1-i)$
27: **end for**
28: $l \leftarrow l + 1$ //WORK ON THE NEXT LAYER
29: **end while**
30: **return** $\emptyset$

Theorem 6.2 (I/O Complexity of External BFS for Accepting Cycle Detection) *External BFS LTL model checking algorithm finds the shortest counterexample with an accepting seed state. Its I/O complexity is*

$$O(sort(|\mathcal{T}| \cdot |\mathcal{R}|) + \mathcal{L}(\mathcal{G}) \cdot scan(|\mathcal{T}| \cdot |\mathcal{S}|))$$

where $\mathcal{S}, \mathcal{R}$ and $\mathcal{T}$ are, respectively, the sets of states, transitions and accepting states in the Büchi automaton of the cross product of the model and the negated LTL property, and $\mathcal{L}(\mathcal{G})$ is the locality in the Breadth-First Search graph obtained during exploration.

Proof Since each state is expanded at most once, sorting can be done in time $O(sort(|\mathcal{T}| \cdot |\mathcal{R}|))$ I/Os. Filtering, evaluating, and merging are all available in the scanning time of the BFS-levels in consideration. The I/O complexity for predecessor elimination depends on the number of BFS-levels that are referred to during file subtraction/reduction. This number is bounded by the duplicate detection scope, and hence by the locality of the Breadth-First search graph in the extended search space. Consequently, the I/O complexity for large-scale LTL model checking is bounded by $O(sort(|\mathcal{T}| \cdot |\mathcal{R}|) + \mathcal{L}(\mathcal{G}) \cdot scan(|\mathcal{T}| \cdot |\mathcal{S}|))$ I/Os. ■

Bounded semantics: Each iteration of the External Breadth-First search can actually be seen as a snapshot in *bounded automata-based model checking*. Bounded model checking (Biere *et al.* 2003) typically uses a propositional SAT solver for the symbolic exploration of model checking problems. It exploits the SATPLAN exploration idea of (Kautz & Selman 1996) using a rising search horizon k to generate Boolean formulae encoding the overall exploration problem up to the BFS-level k.

In bounded explicit-state automata-based model checking a similar approach is used, but without using binary decision diagrams or SAT-formulae. To avoid traversing the full state space in Tarjan's algorithm, we analyze the cross product graph up to a threshold depth value k. If a counter-example is already found in depth k, the search is terminated, otherwise k is increased for the next iteration. For bounded semantics of LTL, the readers are directed to (Krcal 2003).

6.5 Heuristics in Extended State Space

In the extended search space $\mathcal{S}'$ we search for shortest lasso-shaped counterexamples, without knowing the start of the cycle beforehand. But due to the new LaS encoding, we can, in fact, define heuristic estimates for different stages individually.

6.5.1 Heuristics for the Primary Search

An accepting cycle requires visiting an accepting state in the Büchi automaton $\mathcal{B}(\neg\phi)$which is provided beforehand. On the contrary, the Büchi automaton of the model is constructed on-the-fly from the extended finite-state machine (EFSM) synchronizing with the states of the Büchi automaton. This leaves us with a lack of information on the EFSMs states that will couple with accepting states of the Büchi automaton. Consequently, to have a heuristic estimate, we can only exploit the available information on the Büchi automaton $\mathcal{B}(\neg\phi)$.

Let $\mathcal{T}_\phi$ be the set of accepting states in the Büchi automaton $\mathcal{B}(\neg\phi)$. Moreover, let $pc_i, i = \{1 \dots N, \mathcal{B}(\neg\phi)\}$ be the set of program counter functions that when given a composite state, returns the active control state in the i-th component of the N EFSMs and in $\mathcal{B}(\neg\phi)$. The

shortest path distance in the individual automaton is defined as previously with δ. A heuristic estimate h_{pri} for the primary search can then be defined as the minimum distance to reach an accepting state. Formally,

$$h_{pri}(\langle u, \hat{u}, \texttt{false}\rangle) = \min_{t \in \mathcal{T}_\phi} \left\{ \delta(pc_{\mathcal{B}(\neg\phi)}(u), t) \right\}$$

Lemma 6.3 *h_{pri} is admissible and consistent.*

Proof We will first prove the admissibility of the heuristic estimate and then the consistency. Since we are dealing with primary search, the second component $\hat{u}$ can be ignored in the proof.

Admissibility: A heuristic function is admissible, if it never over-estimates the actual shortest path distances. h_{pri} gives us the minimum shortest path distance to a final state in $\mathcal{B}(\neg\phi)$. In the best case, the search arrives at the closest final state t' by taking only

$$\min_{t \in \mathcal{T}_\phi} \left\{ \delta(pc_{\mathcal{B}(\neg\phi)}(u), t) \right\} = \delta(pc_{\mathcal{B}(\neg\phi)}(u), t')$$

transitions, which completes the proof for admissibility as h_{pri} never over-estimated that distance.

Consistency: Without loss of generality, we assume that all edges have a cost 1. A heuristic h is consistent, if for every two states u and v, with v being a direct successor of u,

$$h(u) \leq h(v) + 1 \tag{6.1}$$

There can be two cases: either v is on the shortest path from u to the closest final state t' or it is not. If v is on the shortest path, then taking the transition from u to v would bring us 1 unit close to t' giving $h(v) + 1 \leq h(v) + 1$, a tautology. On the other hand, if v is not on the shortest path from u to t', taking the transition from u to v can take us arbitrarily far from t'. Consequently, we have $h(u) \leq +\infty + 1$, which preserves the inequality and completes the proof. ■

6.5.2 Heuristics for the Secondary Search

Fortunately, in the secondary search, we are not in such a bad position as with the primary search. Unlike earlier, we now know the exact state that we have to search for. A heuristic function can then be defined based on the state machine distances in both the model $\mathcal{M}$ and the Büchi automaton $\mathcal{B}(\neg\phi)$.

$$h_{sec}(\langle u, \hat{u}, \texttt{true}\rangle) = \max \left\{ \sum_{i=1}^{N} \delta(pc_i(u), pc_i(\hat{u})), \delta(pc_{\mathcal{B}(\neg\phi)}(u), pc_{\mathcal{B}(\neg\phi)}(\hat{u})) \right\}$$

Lemma 6.4 *h_{sec} is admissible and consistent.*

Proof This heuristic actually consists of two different parts: the distances in the model $\mathcal{M}$ and the distances in the Büchi automaton $\mathcal{B}(\neg\phi)$. It is known that the maximum of two

admissible and consistent heuristics comprises an admissible and consistent heuristic. A simple explanation is that since both estimates are lower-bounds, taking the maximum of two still preserves the admissibility and consistency properties. In the following we will prove the admissibility and consistency of the first part only as the second part can be proved conversely.

Admissibility: In an asynchronous composition, each transition in the composite system consists of only *one* transition in an EFSM. This implies that for the state u to reach $\hat{u}$, each individual EFSM has to arrive at the corresponding local state. In the best case, i.e., following the shortest path to the desired state $\hat{u}$, $\sum_{i=1}^{N} \delta(pc_i(u), pc_i(\hat{u}))$ transitions are necessary to transform u into $\hat{u}$ (due to asynchronous composition). Hence, the first part never over-estimates the shortest path distance to $\hat{u}$ – a sufficient and necessary condition for the admissibility of a heuristic.

Consistency: A concise proof for the consistency of distances between two states in a finite state machine is presented in (Lluch Lafuente 2003a) that we reproduce here. Consistency requires that for all states u, v with $v \in Succ(u)$, $h(u) \leq h(v) + 1$. In our case, this would expand to:

$$\sum_{i=1}^{N} \delta(pc_i(u), pc_i(\hat{u})) \leq \sum_{i=1}^{N} \delta(pc_i(v), pc_i(\hat{u})) + 1$$

Due to asynchronous composition only one local process is allowed to take a transition at a time; let j be such a process. We can rewrite the above inequality as:

$$\sum_{i=1, i\neq j}^{N} \delta(pc_i(u), pc_i(\hat{u})) + \delta(pc_j(u), pc_j(\hat{u})) \leq \sum_{i=1, i\neq j}^{N} \delta(pc_i(v), pc_i(\hat{u})) + \delta(pc_j(v), pc_j(\hat{u})) + 1. \tag{6.2}$$

Since no transition has happened in the processes $i = 1 \ldots N, i \neq j$, the distances have also not changed, i.e.,

$$\sum_{i=1, i\neq j}^{N} \delta(pc_i(u), pc_i(\hat{u})) = \sum_{i=1, i\neq j}^{N} \delta(pc_i(v), pc_i(\hat{u})),$$

which cancels out the two terms from Inequality 6.2 and leaves us with:

$$\delta(pc_j(u), pc_j(\hat{u})) \leq \delta(pc_j(v), pc_j(\hat{u})) + 1. \tag{6.3}$$

By the triangular property of shortest paths, for any three nodes $u, v, \hat{u}$, we have that

$$\delta(u, \hat{u}) \leq \delta(u, v) + \delta(v, \hat{u}).$$

Since $\delta(u, v) = 1$, the Inequality 6.3 clearly follows from the triangular property. ■

6.5.3 Accumulated Heuristics for Extended State Space Search

$$h_{LaS}(\langle u, \hat{u}, stage \rangle) = \begin{cases} h_{pri}(\langle u, \hat{u}, \texttt{false} \rangle) & \text{if } stage = \texttt{false} \\ h_{sec}(\langle u, \hat{u}, \texttt{true} \rangle) & \text{if } stage = \texttt{true} \end{cases} \tag{6.4}$$

Lemma 6.5 *h_{LaS} is admissible and consistent.*

Proof We have seen in Lemma 6.3 and Lemma 6.4 that both cases individually form an admissible and consistent heuristic. This leaves us with the need to prove both the properties at the *transition* from a primary search to a secondary search. As earlier, we consider both admissibility and consistency separately.

Admissibility: As each counterexample has to contain at least one accepting state from $\mathcal{B}(\neg\phi)$, for primary states $\langle u, \hat{u}, \texttt{false}\rangle$, we have $h_{LaS} = h_{pri}(\langle u, \hat{u}, \texttt{false}\rangle)$ as a lower bound. For secondary states $\langle u, \hat{u}, \texttt{true}\rangle$, we have $h_{LaS} = h_{sec}(\langle u, \hat{u}, \texttt{true}\rangle)$ as a lower bound to close the cycle and to complete the lasso.

Consistency: As both h_{pri} and h_{sec} are consistent (cf. (Lluch Lafuente 2003a), and Lemma 6.3 and 6.4) and exactly one of them is usable at a time in a state, the only case we have to deal with is when a state in primary search spawns a child in secondary search. For this case, a predecessor $\langle u, \hat{u}, \texttt{false}\rangle$ with an evaluation of

$$h_{LaS}(\langle u, \hat{u}, \texttt{false}\rangle) = h_{pri}(\langle u, \hat{u}, \texttt{false}\rangle) = 0, \tag{6.5}$$

spawns a successor v with an evaluation of

$$h_{LaS}(\langle u, \hat{u}, \texttt{false}\rangle) = h_{sec}(\langle v, \hat{u}, \texttt{true}\rangle) > 0. \tag{6.6}$$

The necessary condition for consistency as given by Inequality 6.1 can be written for h_{LaS} as

$$h_{LaS}(\langle u, \hat{u}, \texttt{false}) \leq h_{LaS}(\langle v, \hat{u}, \texttt{true}\rangle) + 1. \tag{6.7}$$

or,

$$h_{pri}(\langle u, \hat{u}, \texttt{false}) \leq h_{sec}(\langle v, \hat{u}, \texttt{true}\rangle) + 1. \tag{6.8}$$

The above inequality clearly holds, due to the Inequalities 6.5 and 6.6. ■

6.6 External Heuristic Search for Liveness Checking

The cycle detection algorithm for directed external LTL model checking is an extension of External A*. It traverses the bucket files along growing $f = g + h$ diagonals, where g is the depth of the states in a bucket and h is the heuristic estimate shared by all the states in the bucket. Algorithm 6.2 shows the pseudo-code of the proposed algorithm. It starts by iterating on the set of initial states and putting them in the corresponding $Open(0, h)$ bucket. All states are queued for primary search by inserting a tuple $\langle u, u, \texttt{false}\rangle$ for all $u \in \mathcal{I}$ in the initial buckets. If an initial state u is also an accepting state, i.e., $u \in \mathcal{T}$, an extended state $\langle u, \hat{u}, \texttt{true}\rangle$ is queued for the secondary search too.

The algorithm proceeds in increasing $f_{\min}$ order, going over the entire diagonal for increasing g-values. A slight change can be noticed in the algorithm concerning the order of duplicate detection. In External Breadth-First search, it was done just after a layer had been expanded, as we knew that the next layer is going to be expanded next. In External A*, however, the order is decided by the $f_{\min}$ value. Hence it is rational to delay the duplicate removal until the bucket is selected for expansion.

The rest of the algorithm is similar to the External Breadth-First search for accepting cycle detection. For every accepting state reached through primary search, two children are

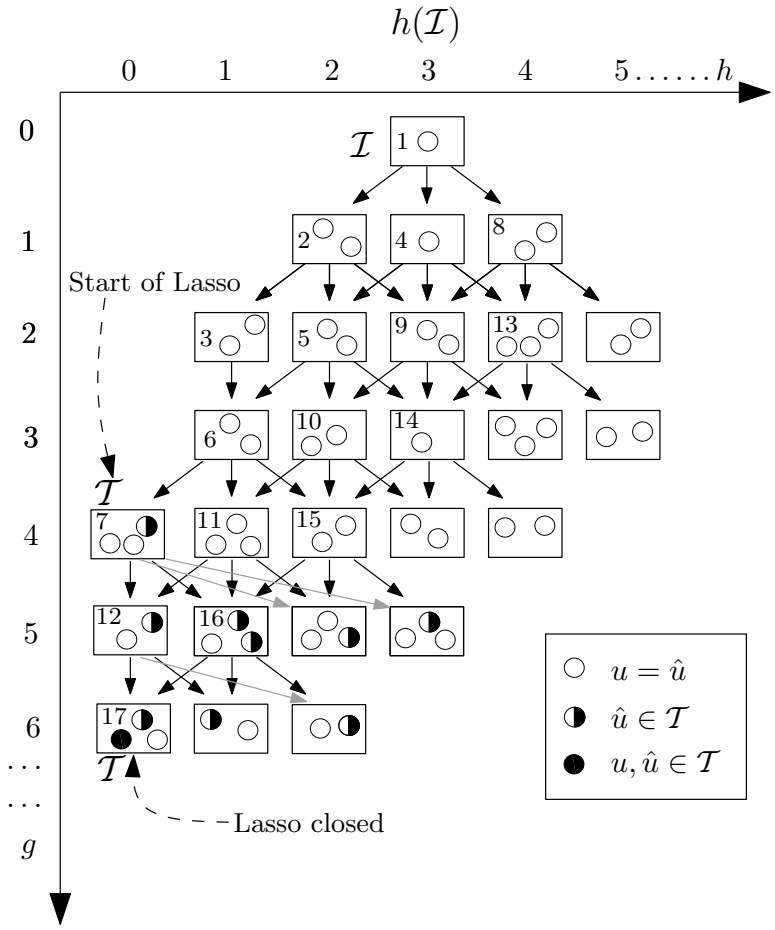

Figure 6.2: Working of External A* for LTL model checking. States in the primary search are denoted with white colored circles, while secondary search states are half black depicting the beginning of a (possible) lasso. Grey arrows show the large jumps when the new heuristic comes into effect. The numbers in the buckets correspond to the execution order.

spawned: One for the primary search and one for the secondary search. If the secondary search reaches the saved state $\hat{u}$ again, the error path is reconstructed and returned.

Figure 6.2 depicts an example execution of the guided exploration for accepting cycle detection as offered by Algorithm 6.2. For primary nodes (illustrated using white circles), we apply the heuristic h_{pri}, while for secondary nodes (half white/half black circles) we apply the estimate h_{sec}. Once a terminal state with $u = \hat{u}$ (black circles) is reached, we have found an accepting cycle.

Given a monotone heuristic estimate such as h_{LaS}, Algorithm 6.2 terminates with a minimal-length counterexample where the lasso seed is accepting. If one allows seed states also to be non-accepting, there are potentially shorter counterexamples. This is possible if the accepting state is reachable only via a non-accepting seed. If one allows seed states only to be accepting states, the path from the seed to the accepting state would appear twice in the corresponding counterexample found by our algorithm. Note that this subtlety does not effect completeness, a lasso with accepting seed exists, if and only if, a lasso with an accepting

Algorithm 6.2 External A* for Accepting Cycle Detection in LTL Model Checking

Input: $\mathcal{I}$: The set of initial states
Output: A lasso shaped path from $\mathcal{I}$ that visits a state $s \in \mathcal{T}$, if one exists

1: **for all** $u \in \mathcal{I}$ **do** //INSERT ALL THE INITIAL STATES IN THE OPEN LIST
2: $\mathit{Open}(0, h_{\mathit{pri}}(\langle, u, u, \texttt{false}\rangle)) \leftarrow \{\langle u, u, \texttt{false}\rangle\}$ //ALL STATES MUST BE INSERTED FOR PRIMARY SEARCH
3: **if** $u \in \mathcal{T}$ **then** //IF THE SEARCH IS STARTING FROM AN ACCEPTING STATE
4: $\mathit{Open}(0, h_{\mathit{sec}}(\langle u, \hat{u}, \texttt{true}\rangle)) \leftarrow \{\langle u, u, \texttt{true}\rangle\}$ //A STATE FOR THE SECONDARY SEARCH
5: **end for**
6: $f_{\min} \leftarrow \min\{h | \mathit{Open}(0, h) \neq \emptyset\}$
7: **while** $(f_{\min} \neq \infty)$ **do**
8: $g_{\min} \leftarrow \min\{i \mid \mathit{Open}(i, f_{\min} - i) \neq \emptyset\}$
9: **while** $(g_{\min} \leq f_{\min})$ **do** //LOOP FOR THE WHOLE $f_{\min}$ DIAGONAL
10: $h_{\max} \leftarrow f_{\min} - g_{\min}$
11: $\mathit{Open}(g_{\min}, h_{\max}) \leftarrow \mathit{sort\text{-}and\text{-}remove\text{-}duplicates}(\mathit{Open}(g_{\min}, h_{\max}))$
12: **for** $i \leftarrow 1$ **to** $\mathcal{L}(\mathcal{G})$ **do** //REMOVE DUPLICATES FROM $\mathcal{L}(\mathcal{G})$ LAYERS
13: **if** $i - g_{\min} < 0$ **then** break //SUBTRACT ONLY LEGAL LAYERS
14: $\mathit{Open}(g_{\min}, h_{\max}) \leftarrow \mathit{Open}(g_{\min}, h_{\max}) \setminus \mathit{Open}(g_{\min} - i, h_{\max})$
15: **end for**
16: **for all** states $\langle u, \hat{u}, \mathit{stage}\rangle \in \mathit{Open}(g_{\min}, h_{\max})$ **do**
17: **for all** States $v \in \mathit{Succ}(u)$ **do** //FOR ALL SUCCESSORS v OF u
18: **if** $\mathit{stage} = \texttt{true}$ **then** //– SECONDARY SEARCH –
19: **if** $v = \hat{u}$ **then** //LASSO CLOSED
20: **return** Construct path$(\langle v, \hat{u}, \mathit{stage}\rangle)$
21: **else**
22: $\hat{v} \leftarrow \hat{u}$ //SAVE THE ACCEPTING STATE FROM THE PREVIOUS STATE
23: $\mathit{Open}(l+1) \leftarrow \mathit{Open}(l+1) \cup \{\langle v, \hat{v}, \texttt{true}\rangle\}$
24: **else** //– PRIMARY SEARCH –
25: $\mathit{Open}(g_{\min}+1, h_{\mathit{LaS}}(\langle v, v, \texttt{false}\rangle)) \leftarrow \mathit{Open}(g_{\min}+1, h_{\mathit{LaS}}(\langle v, v, \texttt{false}\rangle)) \cup \{\langle v, v, \texttt{false}\rangle\}$
26: **if** $v \in \mathcal{T}$ **then** //AN ACCEPTING/FINAL STATE IS REACHED
27: $\mathit{Open}(g_{\min}+1, h_{\mathit{LaS}}(\langle v, v, \texttt{true}\rangle)) \leftarrow \mathit{Open}(g_{\min}+1, h_{\mathit{LaS}}(\langle v, v, \texttt{true}\rangle)) \cup \{\langle v, v, \texttt{true}\rangle\}$
28: **end for**
29: **end for**
30: $g_{\min} \leftarrow g_{\min} + 1$
31: **end while**
32: $f_{\min} \leftarrow \min\{i + j > f_{\min} \mid \mathit{Open}(i, j) \neq \emptyset\} \cup \{\infty\}$ //SELECT THE NEW f DIAGONAL TO EXPAND
33: **end while**
34: **return** $\emptyset$

cycle exists.

I/O Complexity: The basic results of correctness of External A* also extends to External A* for Accepting Cycle Detection algorithm. The worst case asymptotic I/O complexity is bounded by the following theorem.

Theorem 6.6 (I/O Complexity of External A* for Accepting Cycle Detection) *Given a consistent heuristic estimate,* External A* for Accepting Cycle Detection*, formulated as Algorithm 6.2 finds the shortest lasso-shaped counterexample with a worst-case I/O complexity of*

$$O(sort(|\mathcal{T}||\mathcal{R}|) + \mathcal{L}(\mathcal{G}) \cdot scan(|\mathcal{T}||\mathcal{S}|)) \text{ I/Os,}$$

where $\mathcal{L}(\mathcal{G})$ is the locality of the graph $\mathcal{G}$.

Proof The loops in Line 7 and Line 9 guarantee that the buckets will be expanded in increasing $f_{\min}$ order. The consistency of a heuristic, e.g., h_{LaS^2}, guarantees that the new states will never fall in a bucket that is on the left side of the $f_{\min}$ diagonal. Moreover, the termination condition guarantees that the counter-example will only be reported once its bucket is chosen for expansion. All these three conditions are sufficient to prove that the algorithm will terminate with the minimum lasso-shaped counter-example, with an accepting state as the seed.

Since we spawn a secondary search for every accepting state, the extended search space will consist of replicated copies of the original search graph for each of the accepting states. The total number of states in this extended graph would be equal to $|\mathcal{T}| \cdot |\mathcal{S}|$; similarly the transitions would also increase to $|\mathcal{T}| \cdot |\mathcal{R}|$. Consequently, $O(sort(|\mathcal{T}||\mathcal{R}|))$ I/Os are needed to externally sort the list of successors for removing duplicates, and $O(\mathcal{L}(\mathcal{G}) \cdot sort(|\mathcal{S}||\mathcal{R}|))$ I/Os are needed to subtract $\mathcal{L}(\mathcal{G})$ (locality) previous layers to remove duplicates seen earlier. ■

The worst case I/O complexities of both the External Breadth-First search and External A* for Accepting Cycle Detection are the same. However, as always with guided search, if good heuristic estimates are available, the set of expanded states (and, respectively, the transitions) can be much smaller than by the blind search.

The solution path is reconstructed by backward chaining starting with the final state. Since in external search predecessor pointers are not available, we may store with each state its predecessor on a shortest path – doubling the required disk space. Given that l is the length of the counterexample, the I/O complexity of path reconstruction is bounded by the scanning time of at most l buckets in consideration and clearly bounded by $O(scan(|\mathcal{T}||\mathcal{S}|))$ I/Os.

6.7 Duplicate Detection Scope in Extended State Space

We first define the notion of a duplicate state in the extended state space.

Definition 6.1 (Duplicate State in Extended State Space) *A state $\langle u, \hat{u}, stage \rangle$ in the extended state space is a duplicate of another state $\langle v, \hat{v}, stage' \rangle$, if and only if, $u = v$, $\hat{u} = \hat{v}$ and $stage = stage'$.*

This criterion is sufficient to completely distinguish primary search from secondary search.

While performing the bucket subtraction, the duplicate states in the primary search can be identified just by looking at previous $\mathcal{L}(\mathcal{G})$ layers. Analogously, the secondary search

can also be made duplicate free by looking at $\mathcal{L}(\mathcal{G})$ previous layers. If dynamic locality computation is used, two separate localities, one for each search, have to be maintained. The subtraction will then amount to removing the maximum of the two values.

6.8 Experiments

We chose a small internal buffer size for buffered reading and writing, consisting of only 1,997 states. We applied internal (hash table based) and external (delayed) duplicate detection. Duplicate elimination with respect to visited states in previous buckets was not done. This reduced the number of scans to linear-time complexity at the cost of re-expanding some states, but without compromising the optimality of the counterexample lengths. All three heuristics, h_{pri}, h_{sec}, and h_{LaS} have been implemented and used in the experiments.

When comparing our approach to Spin, it should be noted that Spin was invoked with partial order reduction. Actually, as indicated by (Lluch Lafuente 2003a), partial order reduction preserves completeness but not optimality. It may lead to non-optimal counterexamples.

In our first set of experiments, we used an elevator simulation protocol derived from a model by Armin Biere. Table 6.1 shows the exploration results. We report the number of expanded states, the number of states inserted in the hash table, the CPU time consumed and the length of the counterexample obtained. The sizes of the counterexamples are divided into prefix (stem of lasso) and cycle length.

We compare the results of the exploration of External BFS and External A* as implemented in IO-HSF-Spin with Nested-DFS as implemented in Spin v4.2.

For the sake of uniformity with the statistical information provided by Spin, the number of expanded and inserted states are provided instead of the number of stored states and explored transitions*.

Spin and IO-HSF-Spin return counterexamples that start at accepting states†. We observe that Spin's counterexamples are in general longer than the ones in IO-HSF-Spin‡.

Algorithm	Depth	Stored	Transitions	Time		Space in Gigabytes	
				CPU	Real	RAM	Hard disk
IO-HSF-Spin							
External A*	67+34	1,855,625	3,672,977	31s	1m 56s	0.273	0.305
External BFS	67+34	3,046,741	6,023,549	50s	4m 33s	0.0948	0.516
Spin v4.2.4							
Nested DFS	109+100	11,149	33,900	0.046s	0.094s	0.000664	–

Table 6.1: LTL Model Checking with External A*, External BFS and RAM-based Nested DFS for 2-Elevator protocol

*The counterexamples are produced with the options `-t -p`.

†Without the predefined bound on the search depth, Spin tends to find very long counterexamples, e.g. with around 10,000 steps. We therefore chose an iterative Depth-First Search strategy `-i` for Spin. As this option may be caught in a depth anomaly (Lluch Lafuente 2003a) we also checked option `-DREACH`, which should return optimal traces. However, the results we obtained with this setting were not better than with `-i`.

‡This is not necessarily due to their non-optimality, but probably a result of using different measurements for steps; Spin is likely to put some additional increment on synchronized never-claim transitions.

From the results of our first experiments, we do not see a large gain of External A* compared to External BFS in the number of expanded and inserted states. The established counterexample lengths match. In the running times, however, we see that External A* is considerably faster. There are two reasons for this discrepancy. First, as there are less buckets in External BFS (one for each layer) as compared to External A*, there are more I/Os needed for external sorting. The other reason is that the number of generated nodes that fall into the buckets that are not considered for expansion (with counterexamples longer than the optimal) is greater for External BFS.

Spin's exploration is remarkably good, as it requires only 5 milliseconds for generating an optimized trail. The number of stored nodes for Nested-DFS is much smaller as compared to blind BFS and A*, but the established counterexample is much longer.

Algorithm	Depth	Stored	Transitions	Time		Space in Gigabytes	
				CPU	Real	RAM	Hard disk
IO-HSF-Spin							
External A*	15+5	391	257	11s	24s	0.0898	0.0000977
External BFS	15+5	1,286	1,149	<1s	2s	0.0703	0.000541
Spin v4.2.4							
Nested DFS	18+5	8,500	155,963	44s	1m 08s	0.00772	–

Table 6.2: LTL Model Checking with External A*, External BFS and Internal Nested DFS for SGC protocol

In the second experiment, we selected a larger protocol, as used in (Zhang 1999). In Table 6.2, we see an opposite behavior as compared to the previous experiment. External search performed a much smaller number of expansions than internal iterated Nested DFS. The reason is that iterative improvement strategy takes a long time to reduce the counterexample length to a feasibly low number.

In this set of experiments, we see an opposite behavior when compared with the last set. Now the number of expansions in External A* is much smaller than that of External BFS – which we attribute to good guidance. On the other hand, the CPU time consumed by External BFS is less than the time taken by External A*. The reason is that the distribution of heuristic estimates was very fine-grained, which resulted in the allocation of many sparingly used internal buckets.

Algorithm	Depth	Stored	Transitions	Time		Space in Gigabytes	
				CPU	Real	RAM	Hard disk
IO-HSF-Spin							
External A*	196+2	259,410	235,935	25s	3m 0s	0.247	0.639
External BFS	196+2	46,160	129,310	21s	2m 9s	0.253	0.115
Spin v4.2.4							
Nested DFS	–	–	–	–	–	out-of-mem	–

Table 6.3: LTL Model Checking with External A*, External BFS and Internal Nested DFS for 64-Dining Philosopher

In the third set of experiments, we chose the scalable Dining Philosophers protocol with 64 philosophers. The LTL property we checked for was

```
[](philosopher[1]@eat -> <>philosopher[2]@eat),
```

realizing the *response* property that states that *it always holds that if the first philosopher eats, at some point in the future, the second philosopher will also eat*. Table 6.3 shows our results. The number of inserted nodes is, however, smaller for External BFS. This is due to a different order of states in the bucket expanded last. As a result, the goal is generated earlier in External BFS than in External A*.

Spin, unfortunately, ran out of memory. It found counterexample in very large depth, but was unable to shorten the trail. Even provided with a depth bound of 300 it was unable to terminate its iterated improvement strategy, due to the limits of main memory, which in our case was 2 gigabytes. Manually adapting the search depth to the optimum of 212 allowed Spin to complete its exploration finding a counterexample with a acceptance cycle seeded at depth 207.

A larger set of experiments describing the performance of External BFS for liveness checking (Algorithm 6.1) are reported in (Barnat, Brim, & Šimeček 2007). The results are discussed in the following section on related work.

6.9 Related work

The approach presented above was the first complete I/O efficient algorithm for on-the-fly liveness checking (Edelkamp & Jabbar 2006c). A recent extension by Barnat, Brim, & Šimeček (2007) proposes another I/O efficient algorithm for accepting cycle detection. Contrary to our algorithm, this algorithm is not on-the-fly and searches for the accepting cycles only when the whole Kripke structure is generated.

The algorithm has its roots in the OWCTY (One Way Catch Them Young) algorithm by Fisler *et al.* (2001). OWCTY is an accepting cycle detection algorithm based on topological sort and strongly connected components (SCC). It was presented in symbolic setting, where a set of states is represented by a characteristic boolean function called BDD (Binary Decision Diagrams). The algorithm itself is an off-line algorithm – it generates the whole state space and then iteratively prunes the parts of the state space that do not lead to any accepting cycle. The underlying exploration strategy is Breadth-First based. Since it works on sets of states, rather than on individual states, OWCTY is an excellent candidate for parallel and external settings. Černá & Pelánek (2003a) presented a parallel variant of this algorithm.

The I/O efficient variant of OWCTY works on iteratively refining an approximation set until only the reachable cycles are left in it. Starting with the whole reachable state space in this set, the algorithm alternates between two phases: Reachability and Elimination. During the *reachability* phase, all states that are not reachable from an accepting state are removed – as it is for certain that these states do not belong to an accepting cycle. This approximation set is then further refined in the *elimination* phase. In this phase, we remove all states that have no predecessors left in the approximation set due to the last reachability phase. The process continues until a fix-point is reached – no more deletions are possible. The resulting states in the approximation set correspond to all the accepting cycles in the graph.

The total I/O complexity of External OWCTY is

$$O\left(l_{SCC} \cdot \left((h_{BFS} + p_{\max}) \cdot scan(|\mathcal{S}|) + sort(|\mathcal{R}|)\right)\right)$$

Here l_{SCC} is the length of the longest path in the SCC graph, $p_{\max}$ denotes the longest path

in the graph going through trivial strongly connected components (without self-loops), and h_{BFS} is the height of the BFS tree.

The experimental results compare the performance of the External Breadth-First Search as suggested in Algorithm 6.1 with External OWCTY. It has been reported that External BFS for Accepting Cycle Detection performed best in the *falsification* scenario, i.e., while error targeting. The new off-line approach though performed well when the model was correct and there were no accepting cycles. The main reason is that it avoids the quadratic increase in the state space due to the liveness as safety transformation, but on the cost of being not on-the-fly.

Very recently, Barnat *et al.* (Barnat *et al.*) have presented an EM LTL liveness checking algorithm extended from *Maximally Accepting Predecessors* (MAP) (cf. Section 4.5.3). The authors first characterize different algorithms on the basis of a property that they referred to as *revisiting resistant* property. An algorithm is *revisiting resistant*, if the re-expansions of states does not harm the correctness of the algorithm. It is identified that the algorithms that are based on BFS are revisiting resistant, while algorithms such as OWCTY that are based on topological sorting, are not. Recall that in External BFS, for each newly generated layer, we need to subtract the previous layers. For very small layers, subtraction time can become a major overhead. The authors suggested a decision procedure for deciding whether a newly generated layer has to go through subtraction, or not. This property is exploited in designing the External MAP algorithm. To guarantee, completeness, MAP has to perform many scans of the whole state space, which puts its worst-case I/O complexity to

$$O\left(|\mathcal{T}| \cdot ((d + |\mathcal{T}|) \cdot scan(|\mathcal{S}|) + sort(|\mathcal{T}| \cdot |\mathcal{R}|))\right),$$

where d is the diameter of the graph. In the reported empirical evaluation, External MAP showed a significant performance improvement over Algorithm 6.1 and External OWCTY on invalid properties. This behavior was attributed to the few iterations required on the provided LTL properties. On models where the LTL properties were valid, the performance was comparable to External OWCTY.

6.10 Summary

In this chapter we have combined directed, external approaches to compute optimal counterexamples for LTL properties in explicit-state model checking. Previous attempts of integrating external search into model checking were mainly restricted to safety properties only. With External A* and External BFS for Accepting Cycle Detection (Edelkamp & Jabbar 2006c), we have contributed the *first I/O efficient algorithms* for liveness checking to the model checking community. Contrarily to *Nested DFS*, our approach provides an optimality guarantee on the length of the counterexample.

The search space is generated using state pairs of active and cycle seed state, which supports the design of monotone LTL heuristics for directed model checking. Primary and secondary search states are examined together in one common file. The underlying exploration algorithm extends External A* to allow accepting cycles to be found. As with External A*, the approach can effectively be parallelized. Up to synchronization mechanisms for work distribution, no communication between the individual processes is needed, which in large problems allows almost linear speed-ups in a distributed environment. The next chapter is

dedicated to the construction of a *Distributed External A** algorithm that allows safety and liveness checking to be conducted on a network of workstations.

With this research, we hope to have pushed the limits of practical model checking, where the internal memory does not limit the number of realistic models that can be verified.

Chapter 7

Distributed Model Checking

Scaling a model typically involves adding new processes. This in turn means that expanding a state now implies firing more transitions, or to put it differently, new edges are added to each state. A direct implication is an increase in the internal computation time per state. Profilings of our external directed model checker revealed some interesting bottlenecks in its running time. Surprisingly, in a disk-based model checker, internal processing time for successor generations and state comparisons substantially exceeded disk access times. With multi-core machines and clusters of workstations widely available, a natural solution to remove this bottleneck is to develop a distributed variant of External A*.

Parallel or *distributed search algorithms*[*] are designed to solve algorithmic problems by using a network of processors/computers and distributing the work load among the contributing processors. An efficient solution can only be obtained, if the organization between the different tasks can be optimized and distributed in such a way that the working power is effectively used. *Distributed model checking* (Stern & Dill 1997)[†] tackles with the state explosion problem by profiting from the amount of resources provided by parallel environments. A speedup is expected, if the load is distributed uniformly with a low inter-processes communication cost.

In this chapter, we present a multi-processor variant of External A* for external directed model checking that constitutes a two-fold improvement of the single-processor counterpart. Firstly, the internal workload is divided among different processors that can either be residing on the same machine or on different machines. Secondly, we suggest an improved distributed duplicate detection scheme based on multiple processors and multiple hard disks. We show that under some realistic assumptions, we achieve a number of I/Os that is linear to the size of the state space. The distributed version of External A* that we term as *Distributed External A** (Jabbar & Edelkamp 2006) is based on the observation that the internal work in each individual bucket can be parallelized among different processors.

[*] As it refers to related work, even for this text terminology is not consistent. In AI literature, the term *parallel search* is preferred, while in model checking research, the term *distributed search* is commonly chosen. In theory, parallel algorithms commonly refers to a synchronous scenario (mostly according to a fixed architecture), while *distributed algorithms* are preferably used in an asynchronous setting. In this sense, the chapter considers the less restricted distributed scenario.

[†] The first work on distributed model checking is reported by Aggarwal, Alonso, & Courcoubetis (1987), but the algorithm was never implemented. Hence we attribute the field of distributed model checking to the first paper (Stern & Dill 1997) that reports a practical advantage of using a network of computers for model checking.

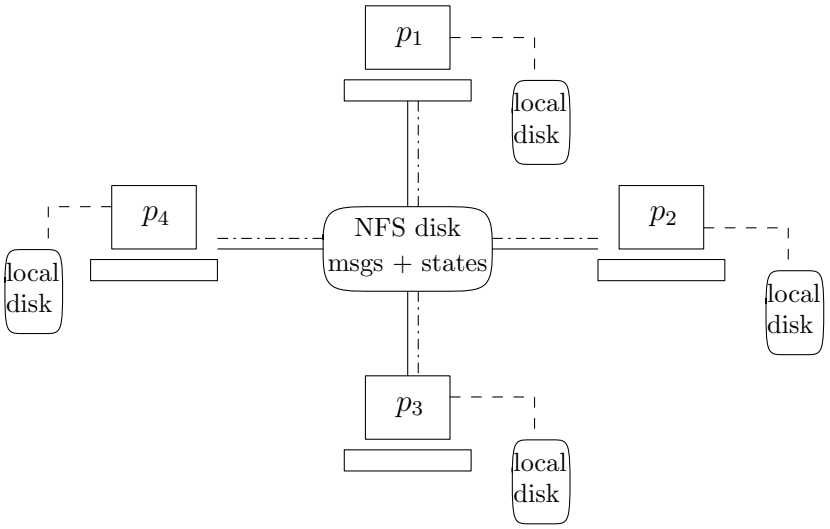

Figure 7.1: An example network architecture with 4 computing nodes. NFS shared disk space is shown in the middle. Optionally, processes can have their local hard disks too for temporary files.

Model checkers that support communication channels and queues need to deal with states of varying sizes. This issue is also addressed in our solution.

Structure of the chapter: We first discuss our assumptions on the network architecture and the connectivity of the computing nodes and the storage devices. Then we present our messaging data structures that are based on shared files. The algorithm is conceptually divided into two phases: distributed expansion and distributed sorting. Each of them is discussed in a separate section. A pseudo-code that formulates the Distributed External A* algorithm is presented next along with an I/O complexity analysis. A set of experiments is provided in the end along with an extensive overview of the related work in the area of distributed search.

7.1 Network Architecture

We assume a network scenario, where P workstations/processors $\{p_1, p_2, \ldots p_P\}$[‡] are interconnected through Ethernet, or remotely over TCP/IP. All workstations have access to a shared hard disk through a network file system (NFS). Optionally, a local hard disk can be present at each workstation to save the intermediate non-duplicate free files. The same setting also extends to multi-core or multiple processors systems. An example with 4 computing nodes is presented in Figure 7.1.

[‡]Abusing the notations, we advise the reader not to confuse p_is with the EFSMs from the previous discussions.

7.2 Messaging over Shared File System

The ability to communicate with other workstations or processors is an integral part of a distributed system. The most common solution is to use a C++ library called MPI[§] (Message Passing Interface) that provides the basic routines for message handling between different processors. To keep the implementation simple, we have avoided the MPI solution. Instead, a new scheme is devised, based on shared files, for message passing and coordination among different computing nodes. These message files are accessible (through shared hard disk space) to all the computing nodes. Such files basically correspond to *critical sections* and should be equipped with mutual exclusion primitives to avoid unnecessary concurrent reads and writes. In order to implement a *set-and-lock* procedure on these message files, we employed a rather simple but efficient scheme – the Linux rename (`mv`) command. The command is executed on a message file to rename it to a unique file name known only by the reading process. Once the reading/writing is done on the file, the file is renamed back to the original file name. For example, whenever the process p_i (i is a unique number automatically assigned to every process that enters the pool) has to write on a shared file, say `sfile`, it issues an operating system rename (`mv`) command to rename the file into $\langle p_i \rangle$`.sfile`. If the command fails, it implies that the file is currently being used by another process and the command should be repeated after a small random wait. Since the granularity of a kernel-level command is much finer than any other program implemented on top of it, the above technique performed remarkably well.

In the following, the three types of generic messages that will be used in the subsequent discussion are enumerated along with a brief description of their usage. All of them are realized by using the above mentioned scheme for mutual exclusion. Through out this chapter, we assume a *master* process and one or many *client* processes – though these are *dynamic roles* and change depending on the situation.

1. *Queues – master to clients*: These types of messages are initiated only by the master. They constitute the expansion and duplicate detection requests. A file is created on the shared hard disk with the jobs from the master. Each client process can read this file and *dequeue* a task. Dequeuing requires writing new information to the file, and hence, creates the problem of concurrent reads and writes.

2. *Acknowledgments – clients to master*: These messages are used by clients who have finished the assigned task, to send a message to the master process as an acknowledgment. Such file messages also require reading and writing by different processors to the same file – another scenario for concurrent reads and writes

3. *Broadcasts – client/master to all*: As the name implies, broadcasts messages are used by both the clients and the master. They are mainly used for broadcasting the termination signal.

7.3 Distributed Expansion

The key observation on which we base the development of *Distributed External A** is:

[§] `http://www-unix.mcs.anl.gov/mpi/`

Observation 1 *Since each two states s, s' in a bucket Open(i, j) have the same depths $g(s) = g(s') = i$ and same heuristic estimates $h(s) = h(s') = j$, they can both be expanded at different processors at the same time.*

This observation follows from the fact that External A* preserves the shortest paths and that the heuristic function is *total*.

7.3.1 Partitioning and Load Balancing

There are various methods of partitioning a state space for workload distribution. A typical practice in distributed state space generation algorithms (see e.g., (Stern & Dill 1997)) is to use a hash function $h_{part} : \mathcal{S} \rightarrow \{p_1, p_2, \ldots p_P\}$ to divide the state space $\mathcal{S}$ among P processors. It is usually assumed that h_{part} uniformly distributes the state space. For each newly generated state, its ownership is checked by the hash function and is then transferred (possibly buffered) to the responsible processor.

In our case, a direct adaptation of such a partitioning is not very fruitful. Since a bucket is saved as a file, selecting a single state, deciding its ownership, and sending it to another processor puts extra I/O and network load. Instead, we decided to partition a bucket into equi-sized parts. The master process queries the file size and the number of active processors in the pool. It then broadcasts the number of bytes to be expanded as:

$$\mathit{ExpansionBytes} = \left\lceil \frac{\mathit{bytes}(\mathit{Open}(g, h))}{P} \right\rceil,$$

where *bytes* is a function that when given a bucket, returns the total number of bytes consumed by the bucket on the hard disk. For improved I/O, this number is supposed to divide the system's block size B. As concurrent read operations are allowed on file systems, multiple processes reading different parts of the same file impose no concurrency conflicts.

The master initiates the expansion process for a particular bucket *Open*(g, h). The expansion queue that we term here as `expand-queue`, is an instance of the file-based queues discussed in the previous section. An expansion request is written to the queue as a tuple

$$\langle g, h, \mathit{minByte}, \mathit{Size} \rangle,$$

which asks to "*Expand a portion of the bucket 'Open(g, h)' starting from the byte 'minByte', and not to expand beyond 'Size'*". For example, an expansion request for the bucket *Open*$(2, 4)$ having a size of 10000 bytes in a two processors scenario would look something like this: $\langle 2, 4, 0, 10000 \rangle$ with *ExpansionBytes* broadcast with a value of 5000 bytes.

Dynamic Load Balancing: A realistic scenario in a networked environment is the disparity in the computational strength of different processors. This would lead to excessive waiting times for the faster processors while they wait for the slower to finish. Such a scenario can be avoided by equipping each processor p_i with an extra parameter $\epsilon(i) \in (0, P]$, such that $\sum_{i=1}^{P} \epsilon(i) = P$. Upon reading the *ExpansionBytes*, p_i divides it with $\epsilon(i)$ and utilizes the result for extracting the bytes that it should expand. The parameter can be changed on-the-fly by the current master by sampling the time taken by different processors The optimal load balancing exists when $\forall i, \epsilon(i) = 1$

7.3.2 Task Dequeuing

Once the file `expand-queue` is generated, several client processes try to get an exclusive right to it. The winner process p_i, reads the file and assigns itself to expanding *ExpansionBytes* number of bytes of the bucket *Open*(g, h). At the same time, it replaces the task with a new task of

$$\langle g, h, \mathit{minByte} + \mathit{ExpansionBytes} + 1, \mathit{Size} \rangle$$

in the queue file. For our example, $\langle 2, 4, 5001, 10000 \rangle$ will be enqueued in the queue.

Dynamic Master Role We assume *master* to be an ordinary process defined as the one that finalizes the work for a bucket. The dequeuing process also defines the role of a new *master*. The new master is the process that got the task to expand the 'last' part of the bucket. It is now master's responsibility to coordinate the flow of the algorithm. After dequeuing the last part, it replaces the queue entry with the task

$$\langle g, h, \mathit{Size}, \mathit{Size} + 1 \rangle$$

in the `expand-queue` to mark the end of expansion tasks. The master should abdicate from this role once it has announced a new expansion task. However, it can happen that the same processor again takes the role as master, if it is the one to dequeue the last task.

7.3.3 Successor Generation

For the expansion to work correctly, the state size must divide the dequeued task range. Unfortunately, in model checking the dynamic contents of message channels within a state can result in states of variable sizes. This problem is solved by using a 4-letter sentinel between the states to mark the boundaries. A task byte range is then aligned with the new boundary, while reading the states files. If, after reading the assigned range, it is found that a state is not completely read, the reading processor is forced to read the whole state. On the other hand, the second processor is forced to start expansion only from the next boundary and to discard the incomplete state.

The results of expansions are saved in separate files: *Open*$(g + 1, h - 1, p_i)$, *Open*$(g + 1, h, p_i)$, *Open*$(g + 1, h + 1, p_i)$. The third argument is added to distinguish the successors generated by the processor p_i from the others. We will write these 3-parameters buckets as *sub-buckets*. To avoid network traffic for non-duplicate-free states, these successors files can be saved on the local hard disk until they are *refined* (externally sorted and with previous layers subtracted).

7.3.4 Acknowledgment of Expansion

After finishing the job, each processor p_i writes an entry in an acknowledgment file as a tuple

$$\langle p_i, g, h, \mathit{minByte}, \mathit{minByte} + \mathit{ExpansionBytes} \rangle$$

to announce that it has finished expanding the range of bytes from *minByte* to *minByte*+ *ExpansionBytes* of the bucket *Open*(g, h). It is now the task of the master to query the original file size and decide if all processors have finished their job. In such a case, the algorithm can

proceed to the sorting and duplicates removal of the new bucket in the respective expansion order: *Open*$(g+1, h-1)$ for External A* and *Open*$(g+1, 0)$ for External Breadth-First Search.

7.4 Distributed Sorting

In contrast to the explicit graphs, where a vertex is just a 32-bit or a 64-bit integer, it is not uncommon in model checking to face a state of size 100 kilobytes or even more. The direct effect is the time expensive comparison operations required during the external sort. As a remedy to this problem, we propose two different methods suitable for two typical connectivity scenarios in the following. The coordination is done through the second queue that we term as `refine-queue`. It contains the g and h values of the bucket that is waiting to be expanded. Its organization is similar to the `expand-queue` and allows clients to dequeue work.

7.4.1 Distributed Sort Single Merge

Since the successors are independent of each other and are saved in separate files for each individual processor, duplicate removal can be done concurrently in each file. This is achieved by *master* inserting the tuple $\langle g, h \rangle$ in the `refine-queue`, once the previous bucket has been fully expanded by all the processors. Each processor p_i reads the queue and initiates a refinement process consisting of external sorting and subtracting the previous layers for the sub-bucket *Open*(g, h, p_i). The result is a sorted and duplicate free sequence for each of the sub-buckets. An acknowledgment is sent to the *master* through a shared file (mutexes have to be handled here, since many processors might be accessing the same acknowledgment file).

Once the *master* is done refining its own sub-bucket, it starts reading the acknowledgments sent by the clients. Only when all the clients are done with the refinement of their sub-buckets, the *master* starts merging the results into one common bucket file *Open*(g, h). This file is then sorted to remove the duplicates that were spread across different sub-buckets. Since the previous layers have already been scanned and subtracted from the sub-buckets, the subtraction for the full bucket can be completely skipped.

7.4.2 Distributed Sort Distributed Merge

The previous method requires a full single merge to produce a duplicate free bucket, during which the clients just sit idle, hence wasting a significant amount of computation time. The following scheme tries to solve this problem through a disjoint partitioning of the bucket. For each bucket that is under consideration, we establish four stages in the algorithm. These phases are visualized in Figure 7.2 (top to bottom). The zig-zag curves represent the sorting order of the states sequentially stored in the files. The sorting criterion is defined by the state's hash key, which dominates low-level state comparison based on the relatively long state descriptor.

Sorting on Partition Order

In the *first sorting stage*, each processor sorts its own file. In a distributed setting, we exploit the fact that the files can be sorted in parallel. To reduce the number of unnecessary I/Os

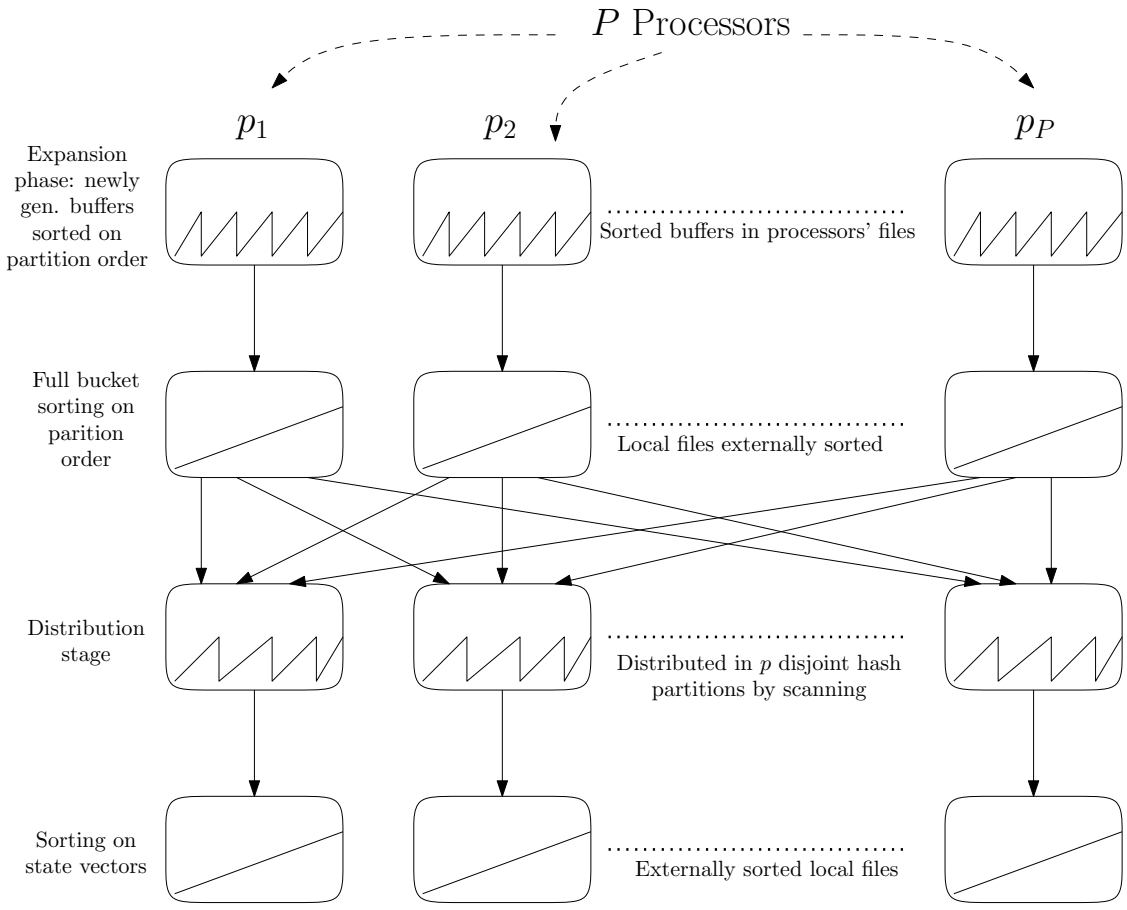

Figure 7.2: External Sorting with Distributed Sort Distributed Merge in *Distributed External A**. Each column corresponds to a process and rows represent different stages in the refinement process. Peaks in rectangles denote sorted state sets.

for reading and writing duplicate nodes we suggest some improvements on the internal sorting method. To remove some duplicates on-the-fly, i.e., as soon as they are generated, we propose to equip each process with its own small hash table in place of the internal memory buffer used for the bucket. This hash table can efficiently catch many of the duplicate states as soon as they are generated. The hash table is based on chaining, with chains sorted along the state comparison function. However, if a hash table exceeds a pre-defined size, the entire hash table is flushed to the disk. This can be done using a mere scan of the hash table. The result is a sequence of states sorted by the hash value and then by the state vector in case of a collision.

Distribution of Sorted States

In the *distribution stage*, each processor distributes the states in the sorted files into different files according to the hash value's range. This is a parallel scan with a number of file pointers that is equivalent to the number of files that have been generated. As all input files are presorted this is a mere scan. No all-including file is generated, keeping the individual file sizes

small.

Sorting on State Vectors

The previous stage results in a file consisting of a sequence of sorted buffers using the hash value's range. In the *second sorting stage*, processors merge these buffers using the actual state vector as the sorting key. The number of peaks in each individual file is limited by the number of input files (= number of processors), and the number of output files is determined by the selected partitioning of the hash index range. The outputs of this phase are sorted and partitioned buffers. Using the hash index as the sorting key we establish that the concatenation of files is in fact totally sorted.

7.5 Termination Detection

When any state $t \in \mathcal{T}$ is expanded by a processor p_i, a message is broadcast to all the processors to terminate the search. The broadcasting is again done through file-based messaging as proposed in Section 7.2. The message is dequeued by a processor *only* when it is ready to dequeue a refinement task. In the worst case, it can happen that the goal was found almost in the beginning of a bucket, but still the whole bucket is fully expanded by all the other processors till they notice the presence of the termination signal.

7.6 Distributed External A*

Algorithm 7.1 shows the pseudo-code of Distributed External A* for safety model checking. The pseudo-code for distributed liveness can be constructed analogously. The algorithm is divided into two main parts: the work for all processors(Lines 5–28) and the exclusive job for the master (Lines 29–48). The work for a process p_i starts by listening for a *GoalFound* message. If no such information is broadcast, it proceeds to dequeue an expansion request. If there is an expansion task, it reads the broadcast *ExpansionBytes* message to find out its share of bytes to expand. After finding out the range of bytes, p_i has to expand, it enqueues a new expansion task in the `expand-queue` (Line 3). Moreover, if p_i is extracting the last expansion task for a bucket, it also declares itself as a *master* and inserts a dummy task in the `expand-queue` (Line 14).

Given that there are no expansion tasks, p_i listens for the duplicates removal task. If a task is found in the queue `refine-queue`, an external sorting and scanning routine is started on the *Open*(g, h, i) sub-bucket, followed by a file subtraction for removing the previous layers. The signal for the end of a refinement process is sent through the acknowledgment channel *Acks-Refine*. If p_i was not selected as a *master*, the control is again transferred to the label `Dequeue-Task` to listen to the task channels.

For a master, the coordination work starts from Line 29. The variable $f_{\min}$ is used, as earlier, to keep the expansion order in increasing $g + h$ values of the buckets. A refinement task for the next bucket *Open*(g, h) is announced, where all processes p_j are requested to make their sub-buckets *Open*(g, h, j) duplicate free and send an acknowledgment. The master, meanwhile, refines its own sub-bucket. It then waits for all the acknowledgments to arrive. Note that in case of an abnormal termination of any of the client processes, the master process will wait infinitely in the *busy-wait* loop in Line ln:busywait. This condition can

Algorithm 7.1 Distributed External A* with Distributed Sort Single Merge.

Input: $\mathcal{I}$: The set of initial states; p_i: Processor Number; P: Total Processes
Output: A path from $\mathcal{I}$ to a state $t \in \mathcal{T}$, if one exists, $\emptyset$ otherwise.

1: **if** $p_i = 0$ **then**
2: $Open(0, h(\mathcal{I})) \leftarrow \mathcal{I}$
3: $enqueue$(expand-queue, $0, h(\mathcal{I})$)
4: $f_{\min} \leftarrow h(\mathcal{I})$; $g \leftarrow 0$
5: Dequeue-Task:
6: $master \leftarrow$ `false` //SELECT THE CURRENT PROCESS AS MASTER
7: **if** *read*(*msgGoalFound*) **then** //HAS SOMEONE FOUND A GOAL?
8: **return** `true`
9: **if** expand-queue $\neq \emptyset$ **then** //IS THERE AN EXPANSION TASK?
10: $\langle g, h, minByte, Size\rangle \leftarrow dequeue$(expand-queue) //DEQUEUE THE EXPANSION TASK
11: *ExpansionBytes* $\leftarrow$ *read*(*msgExpansionBytes*) //READ THE NO. OF BYTES TO EXPAND
12: **if** *minByte* + *ExapansionBytes* $\geq$ *Size* **then** //LAST PART PICKED FOR EXPANSION?
13: $master \leftarrow$ `true` //SELECT THE CURRENT PROCESS AS MASTER
14: $enqueue$(expand-queue, $\langle g, h,$ *Size* $+ 1,$ *Size*$\rangle$) //NOTHING LEFT FOR EXPANSION
15: **else** //ENQUEUE THE LEFTOVER BYTES FOR EXPANSION
16: $enqueue$(expand-queue, $\langle g, h,$ *minByte* + *ExpansionBytes* $+ 1,$ *Size*$\rangle$)
17: *Temp*$(g) \leftarrow$ Partition of the bucket from *minByte* to *minByte* + *ExpansionBytes*
18: **if** $\exists s \in$ *Temp*$(g),\ s.t.\ s \in Targets$ **then**
19: *broadcast*(*msgGoalFound* $\leftarrow$ `true`)
20: **return** *Construct-Solution*(s)
21: *Open*$(g+1, h-1, p_i)$, *Open*$(g+1, h, p_i)$, *Open*$(g+1, h+1, p_i) \leftarrow$ *Succ*(*Temp*(g)) //EXPAND
22: **else if** refine-queue $\neq \emptyset$ **then** //IS THERE A SORTING TASK?
23: $\langle g, h\rangle \leftarrow dequeue$(refine-queue) //DEQUEUE A REFINE TASK
24: *sort-and-remove-duplicates*(*Open*(g, h, p_i))
25: *subtract*(*Open*(g, h, p_i))
26: *Acks-Refines* $\leftarrow$ *Acks-Refines* $\cup \langle g, h, p_i\rangle$ //SEND AN ACKNOWLEDGMENT FOR REFINEMENT
27: **if not** $master$ **then**
28: **goto** Dequeue-Task
29: $f_{\min} \leftarrow g + h$
30: **while** $f_{\min} \leq f_{\max}$ **do**
31: **while** $g < f_{\min}$ **do**
32: $g \leftarrow g + 1$
33: $h \leftarrow f_{\min} - g$
34: $enqueue$(refine-queue, $\langle g, h\rangle$) //ENQUEUE A NEW DUPLICATES REMOVAL TASK
35: *sort-and-remove-duplicates*(*Open*(g, h, p_i)) //EXTERNALLY SORT THE BUCKET
36: *subtract*(*Open*(g, h, p_i)) //SUBTRACT LOCALITY MANY PREVIOUS LAYERS
37: **while** $\exists k, s.t.\ \langle g, h, p_k\rangle \notin$ *Acks-Refine* **do** //A PROCESS p_k IS STILL SORTING?
38: **continue** //WAIT FOR p_k TO FINISH. CAN BE EXTENDED WITH A TIME-OUT.
39: **end while**
40: *Open*$(g, h) \leftarrow$ *Open*$(g, h, 1) \cup \ldots \cup$ *Open*(g, h, P) //MERGE SUB-BUCKETS INTO ONE.
41: *sort-and-remove-duplicates*(*Open*(g, h)) //NO SUBTRACTION IS REQUIRED!
42: *broadcast*$(msgExpansionBytes \leftarrow \lceil bytes(Open(g, h))/P\rceil)$
43: $enqueue$(expand-queue, $g, h, 0,$ *Size*) //ENQUEUE A NEW TASK
44: **goto** Dequeue-Task
45: **end while**
46: $f_{\min} \leftarrow f_{\min} + 1$
47: $g \leftarrow 0$
48: **end while**
49: **return** $\emptyset$

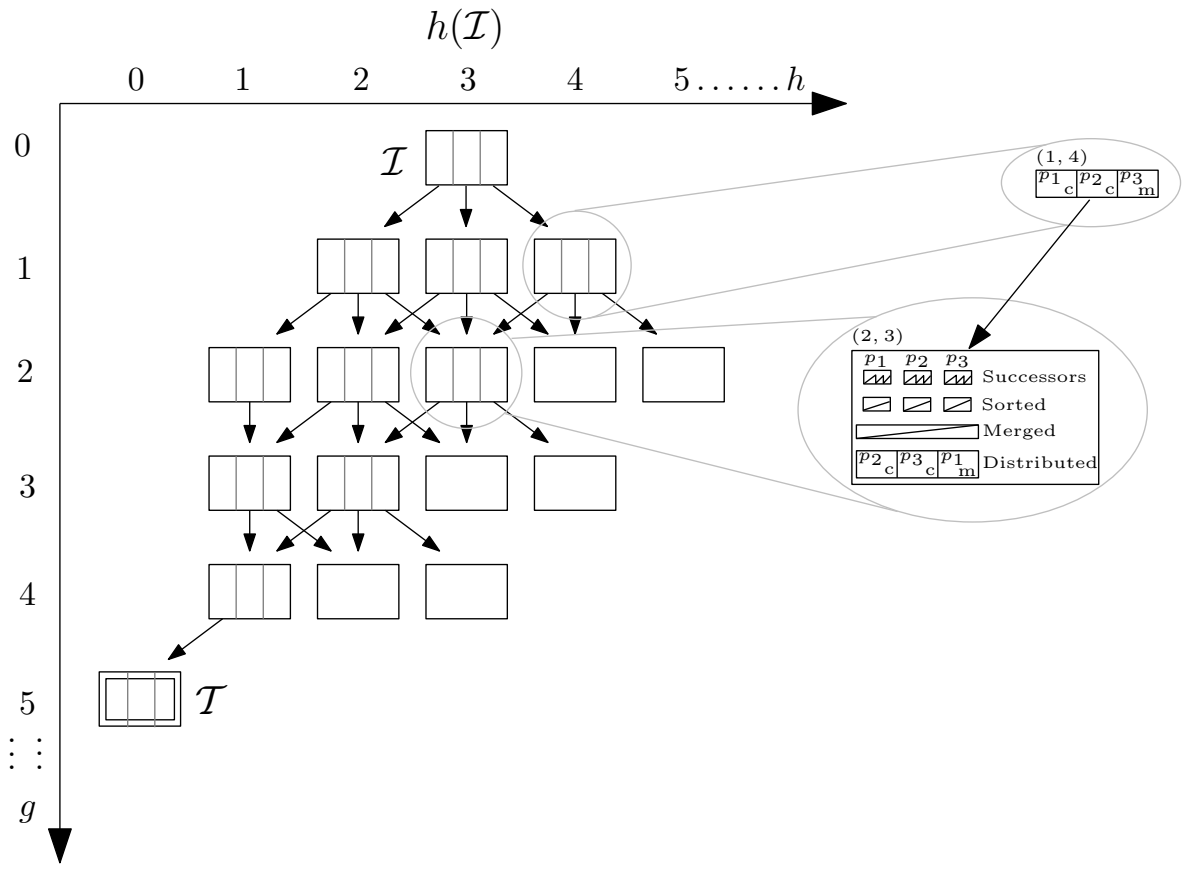

Figure 7.3: Distribution of buckets in External A* among several processors. Two buckets are zoomed in. The bucket $Open(1, 4)$ is shared among three processors with p_3 being the master.

easily be avoided by the use of a time-out and reassigning the unfinished job to the master or to any other client processes. All the refined sub-buckets are then merged into one bucket $Open(g, h)$. The bucket is then transferred to an external sorting routine to remove the duplicates that were spread across multiple sub-buckets. Since each sub-bucket has already been scanned for duplicates in the previous layers, a subtraction phase for the unified bucket is not required. If the bucket is non-empty, an expansion task in then queued in the `expand-queue` (Line 43) and the master gives away its role as a master. The algorithm continues until $f_{\min}$ is less than the given $f_{\max}$ or until the goal has been found.

Figure 7.3 illustrates the workings of distributed exploration. The internal work for exploration of a bucket is uniformly distributed among a set of available processors, that expand and sort individual files as described above.

7.7 Correctness and I/O Complexity of Distributed External A*

We will first prove that parallelism preserves the optimal efficiency of External A* algorithm in the presence of consistent heuristics.

Theorem 7.1 (Correctness of Distributed External A*) *Distributed External A*, in the presence of consistent heuristic estimates, terminates with the shortest path from an initial state $s \in \mathcal{I}$ to a target state $t \in \mathcal{T}$ (if such a path exits), while preserving optimal efficiency.*

Proof A search algorithm is *optimally efficient*, if it does not expand any state that has a higher depth value than a state $t \in \mathcal{T}$, i.e., it does not expand any state s such that $g(s) > \min_{t \in \mathcal{T}} g(t)$. The construction of Distributed External A* guarantees that:

a. The expansion of a bucket starts only when the previous bucket is fully expanded,

b. The termination signal is broadcast as soon as an end state is extracted for expansion.

Both conditions imply that the algorithm terminates only when a target state with the minimum f-value is read from a bucket for expansion. Since at that time all the processors are also working on the same bucket, in the worst case, many end states with the same f-value would be extracted for expansion by different processors – which does not harm the accuracy. Moreover, since Distributed External A* distributes only the expansion and sorting efforts and does not effect the expansion order of A*, the algorithm terminates with the optimal path between s and t. ■

Lemma 7.2 *Let B be the I/O block size, P the number of computing nodes, M the size of the internal memory, and $P \cdot B \leq M$. Moreover, let the states of every bucket be uniformly distributed among sub-buckets. Then, the worst case I/O complexity of Distributed Sort Single Merge is*

$$O\left(\frac{sort(|\mathcal{R}|)}{P} + scan(|\mathcal{R}|\right) \text{ I/Os.}$$

Proof Since the P sub-buckets are of equal sizes, each sub-bucket can be externally sorted individually by the corresponding computing node. As these sub-buckets contain the result of expanding duplicate-free states, the total number of states in the sub-buckets will be in the order of the number of edges in the state space. Hence, the worst case I/O complexity of this phase, when summed over all the buckets is

$$O\left(\frac{sort(|\mathcal{R}|)}{P}\right) \text{ I/Os.}$$

Since there are just P sorted sub-buckets left, and $P \cdot B \leq M$, i.e., P buffers can be read into the main memory, the merging phase by the master will take extra $scan(|\mathcal{R}|)$ I/Os to produce a duplicate free and sorted bucket file in the worst case. The I/O complexity of Distributed Sort Single Merge accumulates to

$$O\left(\frac{sort(|\mathcal{R}|)}{P} + scan(|\mathcal{R}|\right) \text{ I/Os,}$$

which proves the statement of the theorem. ■

Lemma 7.3 *Let B be the I/O block size, P the number of computing nodes, M the size of the internal memory, and $P \cdot B \leq M$. Let the states of every bucket be uniformly distributed among sub-buckets,*

and there exists a hash function (possibly non-injective) $\mathcal{H} : \mathcal{S} \leftarrow \mathbb{Z}$ that uniformly assigns hash values to each state. Then, the worst case I/O complexity of Distributed Sort Distributed Merge is

$$O\left(\frac{sort(|\mathcal{R}|)}{P}\right) \text{ I/Os.}$$

Proof The proof is similar to Lemma 7.2. There are three main phases of this algorithm. In the first phase, the sub-buckets are sorted according to the hash values. During sorting, the duplicate states with a sub-bucket are also removed. The I/O complexity of this phase, when summed over all states generated through every edge, accumulates to

$$O\left(\frac{sort(|\mathcal{R}|)}{P}\right) \text{ I/Os.}$$

In the second phase, the sorted sub-bucket is divided, according to the hash ranges, among the P processors. This step can be done in parallel with a worst case I/O complexity of $O(scan(|\mathcal{R}|)/P)$ I/Os. As a result of the second phase, the states are now disjointly partitioned among the P processors. Each of the sub-buckets has P sorted buffers in it, which can be combined by a mere external scan into one sorted sequence, requiring $O(scan(|\mathcal{S}|)/P)$ I/Os. The total complexity of the whole sorting phase then accumulates to

$$O\left(\frac{sort(|\mathcal{R}|)}{P}\right) \text{ I/Os.}$$

■

Theorem 7.4 (Distributed External A* with Distributed Sort Single Merge) *Distributed External A* algorithm with Distributed Sort and Single Merge takes*

$$O\left(\frac{sort(|\mathcal{R}|)}{P} + \mathcal{L}(\mathcal{G}) \cdot scan(|\mathcal{S}|)\right) \text{ I/Os}$$

in the worst case to find the shortest counterexample of a safety property, where P is the number of processors and $\mathcal{L}(\mathcal{G})$ is the locality of the state space.

Proof The first term follows from Lemma 7.3. The parallelism does not effect the subtraction process, since the scanning of previous $\mathcal{L}(\mathcal{G})$ buckets has to be done for all the sub-buckets. The subtraction takes $O(\mathcal{L}(\mathcal{G}) \cdot scan(|\mathcal{S}|)$ I/Os in the worst case. ■

The external memory model assumes no restriction on the operating system or the hardware. Though, in practice, we have certain restrictions. The most notable one in our context is the limited number of open file pointers per process. We may, however, assume that the number of processes is smaller than the number of file pointers. Furthermore, given enough main memory, we can also assume that the number of flushed buffers (the number of peaks wrt. the sorted order of the file) is also smaller than the number of file pointers. Using this assumption, we can achieve a linear number of I/O for delayed duplicate elimination. The proof is rather simple. The number of peaks k in each individual file is bounded either by the number of flushed buffers or by the number of processes, so that a simple scan with k-file pointers suffices to finalize the sorting.

Theorem 7.5 (Distributed External A* with Distributed Sort Distributed Merge) *Let f be the number of file pointers that can be opened simultaneously on a working thread. If $M \geq f \cdot B$, Distributed External A* algorithm with Distributed Sort and Distributed Merge takes*

$$O\left(\frac{scan(|\mathcal{R}|)}{P} + \mathcal{L}(\mathcal{G}) \cdot scan(|\mathcal{S}|)\right)$$

I/Os in the worst case to find the shortest counterexample to a safety property, where P is the number of processors and $\mathcal{L}(\mathcal{G})$ is the locality of the state space.

Proof The proof is analogous to the proof of Theorem 7.4. Additionally, the complexity of external sorting (Lemma 7.3) is reduced to scanning due to the assumption $M \geq f \cdot B$. The complexity of subtraction remains by $O(\mathcal{L}(\mathcal{G}) \cdot scan(|\mathcal{S}|))$. ■

An important observation is that the more processors we invest, the finer the partitioning of the state space, and the smaller the individual file sizes in a partitioned representation. Therefore, a side effect of having more processors at hand is an improvement in I/O performance based on existing hardware resource bounds.

7.8 Experiments

The Distributed External A* has been implemented for both safety and liveness checking on top of IO-HSF-Spin. The experiments are performed on different kinds of network architectures; the particular details are presented individually. In all the experiments, only the *Distributed Sort Single Merge* scheme for refinement is employed, where each process generates and refines its own set of successors. A master process then combines all the files into one duplicate free file.

The timing information has been collected through the system *time* command. We first report results on deadlock detection using *the number of active processes* as the heuristic estimate. We report both *real* time (the total elapsed time) and *CPU* time (total number of CPU-seconds that the process spent in user mode). The speedup in the columns is calculated by dividing the time taken by a serial execution by the time taken by the parallel execution.

Two characteristics protocols are chosen for safety checking: the CORBA-GIOP protocol as introduced by (Kamel & Leue 2000), and the Optical Telegraph protocol that comes with Spin distribution. The CORBA-GIOP can be scaled according to two different parameters, the number of servers and the number of clients. We selected three settings: *a*) 3 clients and 2 servers, *b*) 4 clients and 1 server, and *c*) 4 clients and 2 servers. For Optical Telegraph, we chose one instance with 9 and another with 14 stations.

CORBA-GIOP protocol has a longer state vector that puts much load on the I/O. On the other hand, Optical Telegraph has a smaller state vector, but takes a longer time to be computed which puts more load on the internal processing. Moreover, the number of duplicates generated in Optical Telegraph is much higher than for CORBA-GIOP.

In the first set of experiments, the multi-processor machine is used. Table 7.1 and 7.2 depict the times[¶] for three different runs consisting of a single process, 2 processes, and 3 processes. The space requirements for a run of our algorithm are as described in the earlier tables.

[¶]The smallest given CPU time always corresponds to the process that first established the error in the protocol.

Problem:	GIOP 3-2			
Time	Proc. 1	Proc. 2	Proc. 3	Speed-up
real	25m 59s			1.0
CPU	18m 20s			1.0
real	17m 30s	17m 29s		1.48
CPU	9m 49s	9m 44s		1.89
real	15m 55s	16m 6s	15m 58s	1.64
CPU	7m 32s	7m 28s	7m 22s	2.44
Problem:	GIOP 4-1			
Time	Proc. 1	Proc. 2	Proc. 3	Speed-up
real	73m 10s			1.0
CPU	52m 50s			1.0
real	41m 42s	41m 38s		1.75
CPU	25m 56s	25m 49s		2.04
real	37m 24s	34m 27s	37m 20s	2.12
CPU	18m 8s	18m 11s	18m20s	2.91
Problem:	GIOP 4-2			
Time	Proc. 1	Proc. 2	Proc. 3	Speed-up
real	269m 9s			1.0
CPU	186m 12s			1.0
real	165m 25s	165m 25s		1.62
CPU	91m 10s	90m 32s		2.04
real	151m 6s	151m 3s	151m 5s	1.78
CPU	63m 12s	63m 35s	63m 59s	2.93

Table 7.1: Times for Distributed External A* in GIOP on a multiprocessor machine.

For GIOP 4-1, we see a gain by a factor of 1.75 for two processors and 2.12 for three processors in the total elapsed time. For Optical Telegraph, this gain went up to 2.41, which was expected, due to its complex internal processing. In actual CPU-time (*user*), we see an almost linear speedup that depicts the uniform distribution of internal workload and, hence, highlights the potential of the presented approach.

Problem:	Optical-9			
Time	Proc. 1	Proc. 2	Proc. 3	Speed-up
real	55m 53s			1.0
CPU	43m 26s			1.0
real	31m 43s	31m 36s		1.76
CPU	22m 46s	22m 58s		1.89
real	23m 32s	23m 17s	23m 10s	2.41
CPU	15m 20s	14m 24s	14m 25s	3.01

Table 7.2: Times for Distributed External A* in Optical Telegraph on a multiprocessor machine.

The largest problem we have been able to solve so far is the deadlock detection in Optical Telegraph protocol with 14 stations (Table 7.3). This problem is solved on a network cluster with 6 Terabytes GPFS (General Parallel File System) shared hard disk and Opteron 2.4 GHz computers. The process size remained fixed at 1.6GB for both single processor and four pro-

cessors' run. With only one processor, it took around 20 days to find the optimal deadlock. The days dropped to 8 when 4 processors were employed. For both runs, the total hard disk space consumption was 3 Terabytes. Unfortunately, due to heavy load on the cluster, we were not able to repeat the experiment for more processors.

Problem:	Optical-14				
Time	Proc. 1	Proc. 2	Proc. 3	Proc. 4	Speed-up
real	479h 46m				1.0
CPU	173h 08m				1.0
real	195h 40m	195h 39m	195h 39m	195h 39m	2.44
CPU	76h 58m	80h 52m	53h 11m	51h 21m	2.27

Table 7.3: Times for Distributed External A* on a larger instance of Optical Telegraph. Total space consumption: 3 Terabytes.

In Table 7.4, we report distributed liveness checking results. We again choose the Dining Philosopher example (see Table 7.4), now scaled-up to 128-philosophers. First, we note that disk space consumption is considerably large. The single processor version could not finish its exploration. One file for the set generated states became larger than 2 gigabytes and was "killed" by the operating system. The reason that the multi-processor versions could finalize their implementation, is due to the partitioning of bucket into sub-buckets. It delayed the creation of an all-including file until each sub-bucket was refined completely. The lengths of the produced counterexamples match and the observed speed-up is noticeable.

Problem:	128-Philosophers			
Time	Proc. 1	Proc. 2	Proc. 3	Speed-up
real	–			
CPU	–			
real	17m 51s	17m 19s		1.0
CPU	8m 20s	8m 56s		1.0
real	15m 25s	16m 1s	15m 38s	1.12
CPU	6m 19s	6m 22s	6m 59s	1.31

Table 7.4: Time for LTL Model Checking with Distributed External A* for 128-Dining Philosophers.

Tables 7.5 and 7.6 show our results for the scenario of two machines connected together via NFS. In GIOP 3-2, we observe a small speed-up by a factor of 1.08. In GIOP 4-1, this gain increased to about a factor of 1.30. When tracing this limited gain, we found that the CPUs were not used at full speed. The bottleneck turned out to be the underlying NFS layer that was limiting the disk accesses to only about 5 Megabytes/sec.

For Optical Telegraph, we see a bigger reduction of about a factor of 1.41. Even though this protocol has a relatively small state vector, the successor generation is complex, which puts much load on internal computation. As in the former setting, the total CPU-seconds consumed by a process in single processor mode is reduced to almost half when two processors are used.

Problem:	GIOP 3-2		
Time	Proc. 1	Proc. 2	Speed-up
real	35m 39s		1.0
CPU	11m 38s		1.0
real	32m 52s	33m 0s	1.08
CPU	6m 35s	6m 34s	1.76
Problem:	GIOP 4-1		
Time	Proc. 1	Proc. 2	Speed-up
real	100m 27s		1.0
CPU	31m 6s		1.0
real	76m 38s	76m 39s	1.3
CPU	15m 52s	15m 31s	1.96

Table 7.5: Times for Distributed External A* in GIOP on two computers and NFS.

Problem:	Optical 9		
Time	Proc. 1	Proc. 2	Speed-up
real	76m 33s		1.0
CPU	26m 37s		1.0
real	54m 20s	54m 6s	1.41
CPU	14m 11s	14m 12s	1.87

Table 7.6: Times for Distributed External A* in Optical Telegraph on two computers and NFS.

7.9 Related work

Conventional model checking techniques have high memory requirements and are very computationally intensive; they are thus unsuitable for handling real-world systems that exhibit complex behaviors which cannot be captured by simple models having a small or regular state space. Various authors have proposed ways of solving this problem by distributing the memory and computational requirements over a cluster of workstations.

One of the first efforts in this direction is reported in (Aggarwal, Alonso, & Courcoubetis 1987). Stern & Dill (1997) employed a hash-based partitioning scheme for dividing the whole state space into multiple computing nodes. The proposed approach was implemented on top of Murϕ verifier. Lerda & Sisto (1999) experimented with a different partition function based on the states of only one process automata. The rational behind such a function is that a transition usually performs only a few local changes in a system. Therefore, with a high probability the successor state might also belong to the current node. Haverkort, Bell, & Bohnenkamp (1999) introduced the distributed search for stochastic Petri nets. Distributed verification in μ calculus is reported by (Bollig, Leucker, & Weber 2001) and for CTL* by (Inggs & Barringer 2006). There have been attempts to also consider symbolic techniques, real-time (Behrmann, Hune, & Vaandrager 2000) and SAT-solving (Garavel, Mateescu, & Smarandache 2001) in a distributed fashion.

Recently the trend is directed towards verification on multi-core machines. Such machines offer the advantage of having negligible overhead for state transfers due to shared memory. Holzmann & Bosnacki (2007) presented a method for multi-core extension of Spin,

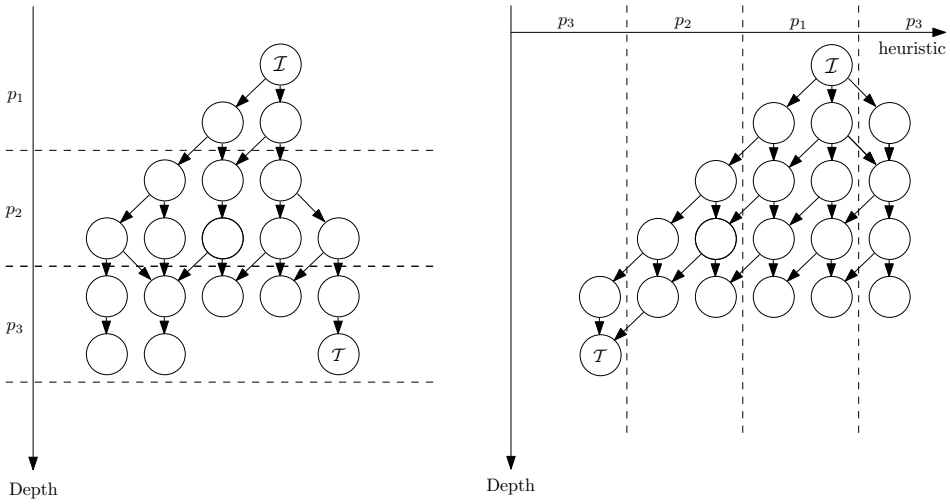

Figure 7.4: State space partitioning with depth-slicing (left) and on heuristic values (right). pi's denote the computing nodes. $\mathcal{I}$ is the initial state and the error state is shown with $\mathcal{T}$.

where the safety analysis is applicable to N-core systems. It horizontally slices the depth-first search tree and assigns each slice to a different node. The situation is depicted to the left of Figure 7.4. The superiority of this method on a hash-based partitioning is due to the fact that with high probability the successors of a state also stay at the same computing node – hence saving the network overhead.

Walking on the similar footsteps, Edelkamp, Jabbar, & Sulewski (2007) recently proposed a *vertical* partitioning of the search tree based on the heuristic values. The core advantage is that it is not only suited to depth-first, but to any general state expanding strategy including greedy best-first search, A* and breadth-first search. A visualization is given to the right of Figure 7.4. The new scheme has been implemented in the context of verification of multi-threaded C++ programs.

Distributed Liveness Checking Recall that LTL model checking mainly entails finding accepting cycles in a state space, which is performed with the nested depth-first search algorithm. The correctness of this algorithm depends on the depth-first traversal of the state space. Since depth-first search is inherently sequential (Reif 1985), additional data structures and synchronization mechanisms have to be added to the algorithm. These requirements can waste the resources offered by the distributed environment. Moreover, formally proving the correctness of the resulting algorithms is not easy. It is possible, however, to avoid these problems by using partition functions that localize cycles within equivalence classes. The above described methods for defining partitions can used for this purpose, leading to a distributed algorithm that performs the second search in parallel. The main limiting factor is that scalability and load distribution depend on the structure of the model and the specification.

Barnat, Brim, & Stříbrná (2001) reported the first effort towards checking liveness properties in a distributed setting. Here the Spin model checker has been extended to perform nested depth-first search in a distributed manner. They proposed to maintain a dependency structure for all the accepting states visited. The nested parts for these accepting states are

then started as separate procedures based on the order dictated by the dependency structure.

A wide body of important results on distributed verification for both safety and liveness is contributed by the Paradise lab mostly implemented in Divine environment (Barnat *et al.* 2006). A distributed cycle detection algorithm for LTL model checking based on parallel Breadth-First search is reported in (Barnat, Brim, & Chaloupka 2003). Another algorithm by the same group is an extension of 'OWCTY' algorithm for distributed setting (Černá & Pelánek 2003a).

In Multi-core scenario, the method by Holzmann & Bosnacki (2007) scales up to only dual-core processors. Barnat, Brim, & Rockai (2007) extended different parallel LTL model checking algorithms to N-core systems and report an extensive experimental evaluations of different liveness checking algorithms.

Distributed External Breadth-First Search in Undirected Graphs Another approach that very closely resembles to ours is by Korf & Schultze (2005). The authors propose to partition the entire state space into files using a hash function. The hash address is used to distribute and to locate states in those files. As the considered state spaces like the Fifteen-Puzzle are regular permutation games, each state can be perfectly hashed to a unique index. Since all state spaces are undirected, in order to avoid regenerating explored states, *frontier search* (see Section 2.7) stores, with each node, its *used* operators in the form of a bit-vector in the size of the operator labels available. The *used* operators allow to distinguish neighboring states that have already been explored from those that have not. Consequently, the generation of predecessors is avoided.

Korf & Schultze (2005) also proposed to avoid sorting-based delayed duplicate detection by using *hash-based delayed duplicate detection*. It uses two orthogonal hash functions: one for partitioning the state space into files, and one to address states in a file. When a state is explored, its children are written to a particular file based on the first hash value. In cases like the sliding-tiles puzzle, the filename corresponds to parts of the state vector. For space efficiency, it is favorable to perfectly hash the rest of the state vector in order to obtain a compressed representation. The representation as a permutation index can be computed in time linear to the length of the state vector. Figure 7.9 depicts the layered exploration on the external partition of the state space. Even on a single processor, multi-threading is important to maximize the performance of disk-based algorithms. The reason is that a single-threaded implementation will run until it has to read from or write to disk. At that point, it will block itself until the I/O operation has been completed. Moreover, hash-based delayed duplicate detection is well-suited for distribution. Within an iteration, most file expansions and merges can be performed independently.

To realize parallel processing a work queue is maintained, which contains parent files waiting to be expanded, and child files waiting to be merged. At the start of each iteration, the queue is initialized to contain all parent files. Once all the neighbors of a child file are finalized, it is inserted in the queue for merging. Each thread works as follows. It first locks the work queue. Two parent files conflict, if they can generate states that hash to the same child file. The algorithm checks whether the first parent file conflicts with any other file currently being expanded. If so, it scans the queue for a parent file with no conflicts. It swaps the position of that file with the one at the head of the queue, grabs the non-conflicting file, unlocks the queue, and expands the file. For each child file it generates, it checks to see if all of its parents have been expanded. If so, it puts the child file to the queue for merging,

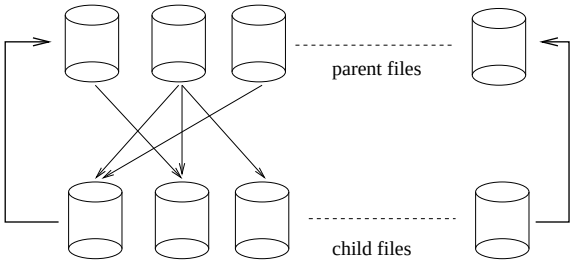

Figure 7.5: Externally stored state space with parent and child files.

and then returns to the queue for more work. If there is no more work in the queue, any idle thread will wait for the current iteration to finish. At the end of each iteration, the work queue is initialized to contain all parent files for the next iteration.

Zhou & Hansen (2007) proposed a parallel variant of *structured duplicate detection* (SDD). SDD is based on disjointly partitioning the state space, such that each partition and its neighbouring partitions, i.e., where the potential successors might fall into, fit into the RAM. The parallel version of SDD assigns partitions that do not share a common neighboring partition to different processors. Heuristic guidance is integrated into the algorithm by using Breadth-First Heuristic Search as the exploration order. The algorithm has been implemented and evaluated on STRIPS planning problem. A speedup is reported for a maximum of 4 instances running on a two processors dual-core Xeon machine with local hard disk.

A recent approach to parallel and external model checking is presented by Holub & Tuma (2007). The approach distinguishes itself from the others in that the state space is not divided but is circulated among all the computing nodes. It has been evaluated on a dinning philosophers instance involving 13 philosophers, a produce-consume problem and an industrial example. A linear speedup is obtained in all three problems. The exact amount of hard disk space consumed is not reported. Moreover, no I/O complexity results are provided.

7.10 Summary

This chapter has shown a successful approach for extending the External A* exploration to a distributed environment, such as multi-processor machines or workstation clusters. Expansion and delayed duplicate detection are parallelized while avoiding concurrent write accesses, which are often not available.

Distributed External A* (Jabbar & Edelkamp 2006) is the *first algorithm to have combined distributed, external, and guided search* both in model checking and in AI. In model checking, it still remains as the only external and distributed algorithm that supports *liveness checking*.

Under reasonable assumptions on the number of file pointers per process, the number of I/Os required for externally sorting a file is reduced to external scanning – which is linear in the size of the model. The parallel algorithm presented is not specific to model checking only, but can be applied to other areas where searching in a large state space is required.

Distributed External A* differs from other distributed model checking approaches in several ways. Most significant of them is that it can impose an order on the exploration that allows us to include heuristic guidance to reach a desired state, faster. Such a facility does not exist in other external algorithms. Moreover, it does not require any regular structure to be

present in the state space, which makes it superior to parallel structured duplicate detection algorithm.

Error trails provided by depth-first search exploration engines are often exceedingly lengthy. Employed with a lower-bound heuristic, the proposed algorithm yields counter-examples of optimal length, and is, therefore, an important step to ease error comprehension for the programmer/software designer.

The approach has been implemented for both *safety* and *liveness* checking in the LTL model checker IO-HSF-Spin. Experimental results show an almost linear speedup in the CPU-time and a significant gain in the total elapsed time. By fine tuning the low-level routines, we expect to close this gap between the CPU and the total time. For our largest execution (three Terabytes), Distributed External A* has succeeded in decreasing the running time required by the single processor version from 20 days to just 8 days by employing four processors.

Chapter 8

Real-Time Model Checking

Many safety critical systems involve time, a typical example being a railway crossing through a busy street. The road barriers should close before the arrival of the train and should open again once the train has passed. A time-controller is required here to guarantee that the stop signals are turned on and that the barriers are closed at a safe time for the intersecting traffic to stop completely. Model checking of systems that involve time constraints poses a great challenge to the model checking community because of the possible number of reachable states. The formalism best-suited to model timed systems is *timed automata* (Alur & Dill 1994).

Real-time model checking with timed automata is an important decidable subfield of the analysis of hybrid automata (Henzinger *et al.* 1998) with a number of industrial applications. Similar to their untimed counterpart, model checking of timed systems also suffers from the so-called state space explosion problem. For large timed systems, limited RAM is the main cause of failure of real-time model checking. UPPAAL (Larsen *et al.* 1997) is one very successful verification tool based on timed automata. It provides the facility for modeling, simulation and validation of real-time systems. It deals with non-deterministic processes with finite control structure, channel or shared variable communication, and real-valued clocks. UPPAAL-CORA (Larsen *et al.* 2001) is an extension of UPPAAL designed for efficient cost-optimal reachability analysis in priced timed automata. UPPAAL-CORA is also competitive in resource-optimal scheduling (Rasmussen, Larsen, & Subramani 2004).

This chapter explores the possibilities of equipping the algorithms for real-time model checking and real-time cost-optimal scheduling with external memory (Edelkamp & Jabbar 2007). The notion of a state in these systems is different than what we have seen in the previous chapters, which shifts the challenge to I/O efficient duplicates removal while searching for a cost-optimal path. Two new algorithms are proposed: *External Breadth-First Search* for a blind but complete real-time verification and *Iterative Broadening External Breadth-First Branch-and-Bound* for cost-optimal reachability in timed systems. The proposed algorithms provide a controlled and guided exploration of the state space.

Structure of the chapter: First, a review of timed automata for real-time model checking is provided along with an external memory exploration algorithm for timed automata. Priced timed automata are discussed next. Then, a RAM-based Branch-and-Bound algorithm for cost-optimal reachability analysis are discussed. An EM Branch-and-Bound algorithm is

presented which mitigates the memory bottleneck problem in the RAM-based algorithm. Later, we propose an iterative broadening beam search algorithm that tries to find a good upper bound by searching in only a fragment of the state space. The approach has been successfully implemented in UPPAAL-CORA. Finally, experimental results on various aircraft landing scheduling problems are presented.

We restrict ourselves to the cost optimization variant of reachability analysis. For extending these explorations to real-time model checking with respect to temporal properties we refer the reader to (Cassez *et al.* 2005). Moreover, in this text, we consider real-time model checking with *timed automata*, for which the reachability problem is decidable but PSPACE-hard (Alur & Dill 1994). It should be noted here that, to remain consistent with the standard notations in the field of timed automata, we will reuse some of the symbols defined in the previous chapters.

8.1 Real-Time Model Checking with Timed Automata

Timed automata can be viewed as an extension of classical finite automata equiped with clocks and clock constraints. These constraints, when corresponding to states are called *invariants*, and restrict the time allowed to stay at a state. When corresponding to transitions, these constraints are called *guards*, and restrict the use of the a transition. The clocks C are real-valued variables and are used to measure durations. The values of all the clocks in the system are denoted as a vector, also called clock valuation function $v : C \rightarrow \mathbb{R}^+$. The constraints are defined over clocks and can be generated by the following grammar: for $x, y \in C$, a constraint α is defined as,

$$\alpha ::= x \prec d \mid x - y \prec d \mid \neg\alpha \mid (\alpha \wedge \alpha),$$

where $d \in \mathbb{Z}$ and $\prec \in \{<, \leq\}$. These constraints yield two different kinds of transitions: delay and edge transitions. *Delay* transitions refer to waiting for some duration in the current state s – provided the *invariant*(s) holds. These transition let the clock variables increase during the time delay. *Edge* transitions are simple state to state transitions, additionally, when executed, they can reset some of the clock variables. Edge transitions are allowed to be executed only if *guard*(t) evaluates to true. An edge transition is taken without an increase in the clock variables, i.e., in time 0. *Trajectories* are alternating sequences of states and transitions and define a path within the automata. The reachability task is to determine, if the goal in the form of a partial assignment to the ordinary and clock variables can be reached or not. The optimal reachability problem is to find a trajectory that minimizes the overall path length.

For reachability analysis on timed automata, one is faced with the problem of an infinite-state space. This infiniteness is due to the fact that the clocks are real-valued and hence an exhaustive state space exploration can yield infinite branches. Alur & Dill (1994) suggested a procedure to partition the hyper-space induced by clock values in finite equivalence classes called *regions*. Decidability result for timed automata are due to these regions. This partitioning is mainly based on the observation that for clock constraints only the natural numbers are used to bound clock values, which means that the fractional part of a clock has a limited effect. Moreover, if the value of a clock goes beyond the maximum number with which it is compared to in the system, then any value taken by that clock becomes uninteresting.

Formally, Alur & Dill (1994) suggested the following three rules that are guaranteed to give finitely many equivalent classes of clock valuations. Let *frac*$(v(x))$ represent the fractional part of the valuation of clock x. We say that for two clock valuations v, v', $v \approx v'$, if and only if

1. $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ or $v(x) > k(x)$ and $v'(x) > k(x), \forall x \in C$, and
2. *frac*$(v(x)) \leq$ *frac*$(v(y))$ iff *frac*$(v'(x)) \leq$ *frac*$(v'(y))$, $\forall x, y \in C$ with $v(x) \leq k(x)$ and $v(y) > k(y)$
3. *frac*$(v(x)) = 0$ iff *frac*$(v'(x)) = 0$ $\forall x \in C$ with $v(x) \leq k(x)$.

Here $k(x)$ denotes the maximum value with which clock x is compared in the constraints. We call these equivalence classes as *clock regions*.

Although this partitioning scheme provides us with finitely many equivalence classes, which makes real-time model checking *decidable*, from a practical point of view, this partition is still very *fine*. Alur & Dill (1994) extended this idea to a *coarser* representation called *Zone*, which is defined on a convex union of regions. Formally, a *zone* Z over a set of clocks C is a finite conjunction of simple difference constraints of the form $x - y \leq d$ or $x - y < d$, with $x, y \in C$ and integer d^*.

For a clock vector u and a zone Z we write $u \in Z$ if u satisfies the constraints in Z. The two main operations on (clock) zones are clock *reset* $\{x\}Z = \{u[0/x] \mid u \in Z\}$ that resets all the clocks x, and *delay* or *future* (d time units) $Z^{\uparrow} = \{u + d \mid u \in Z\}$. The reachability problem in timed automata can then be reduced to the reachability analysis in *zone automata*. In a zone automaton, each state is a *symbolic state* corresponding to one or many states in the original timed automaton. The new state is represented as a tuple (l, Z), with l being the discrete part containing the local state of the automata, and Z is the convex $|C|$-dimensional hypersurface in Euclidean space. Semantically, (l, Z) now represents the set of all states (l, u) with $u \in Z$. Let $\mathcal{B}(C)$ denote the set of constraints defined on clocks C and $\mathcal{P}(C)$ the power set of C. Formally, a timed automaton can be defined as follows:

Definition 8.1 (Timed Automaton) *A timed automaton is a tuple* TA $= (\mathcal{S}, l_0, \mathcal{R}, \text{Inv}, \mathcal{T})$*, where*

- $\mathcal{S}$ *is the set of states,*
- (l_0, Z_0) *is the initial state with an empty zone,*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{B}(C) \times \mathcal{P}(C) \times \mathcal{S}$ *is the transition relation making states to their successors, given that the constraints on the edge are satisfied,*
- Inv : $\mathcal{S} \rightarrow \mathcal{B}(\mathcal{C})$ *assigns invariants to the states, and*
- $\mathcal{T}$ *is the set of accepting/target states.*

Zones can be visualized as an intersection of the space defined by each clock constraint in a $|C|$-dimensional Euclidean space.

Example Let Z be a zone defined over two clock variables x and y as

$$Z := x \leq 4 | 0 \leq x | y \leq 3 | 0 \leq y | x - y \leq 3 | y - x \leq 1.$$

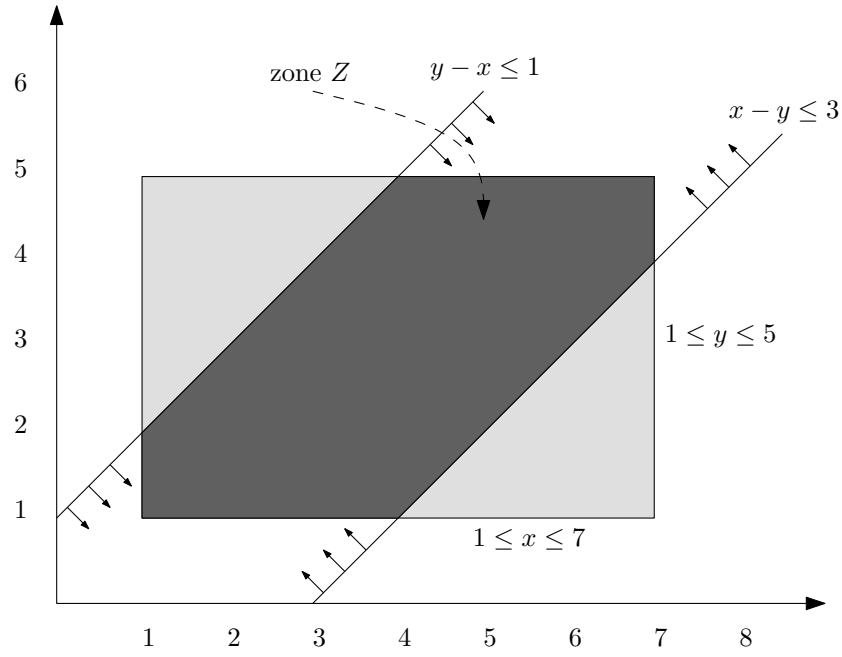

Figure 8.1: Geometrical representation of a zone defined on 2 clock variables in 2D Euclidean space

The visualization of the zone as an intersection of the constraints is depicted in Figure 8.1.

Theoretically, the number of zones can be exponential in the number of *regions*, however, in practice, number of reachable zones are very much restricted in an on-the-fly generated zone-based automaton. For a more thorough discussion on the decidability of timed automata, readers are directed to (Bengtsson & Yi 2004).

8.2 External Search in Real-Time Systems

One of the involved differences between real-time reachability and ordinary reachability analysis is the *inclusion check*. While in (delayed) duplicate elimination, we omit all identical states from further consideration, in real-time model checking we have to check inclusions of the form $Z \subseteq Z'$ to detect duplicate states. We say that Z is *closed under entailment*, if no constraint of Z can be strengthened without reducing the solution set. The time-complexity for inclusion checking is linear to the number of constraints in Z.

Subsequently, while porting real-time model checking algorithms to an external setting, we have to provide an option for the elimination of zones. Since we cannot define a *total order* on zones, trivial external sorting schemes are useless in our case. In our proposal of External Breadth-First Search we exploit the fact that two states (l, Z) and (l', Z') are comparable only when $l = l'$. This motivates the definition of *zone union* $\mathcal{U}$ where all zones correspond to the states sharing a common discrete part l, and for all $Z, Z' \in \mathcal{U}$, we have $Z \not\subseteq Z'$.

*Unary constraints $x \leq d$ or $x < d$ are rewritten as $x - x_0 \leq d$ and $x - x_0 < d$ for some start time clock variable x_0, $x - y \geq d$ as $y - x \leq -d$ and $x = y$ as $x - y \leq 0$ and $y - x \leq 0$.

Duplicate states can now be removed by first sorting with respect to the discrete part l, which will bring all states sharing the same l close together, and then doing a one-to-one comparison among all such states. The result of this phase is a file where states are sorted according to the discrete parts l forming duplicate free zone unions.

However, the one-to-one comparison of all the zones for a particular l can only be performed I/O-efficiently when all the states sharing the same l can be read into the main memory. Throughout this presentation, we assume that this requirement holds. The same assumption also holds during set refinement with respect to previous BFS layers. We load both the zone union from the predecessor file and the one in the unrefined file and check for the entailment condition.

State spaces that appear in real-time model checking are usually directed and hence just removing duplicates with respect to the previous two layers is not sufficient. The crucial complexity parameter is the locality or duplicate elimination scope (c.f. Chapter 5) which defines the number of previous levels to be considered. In this text, the notion of locality for an automaton *TA* is referred to as $\mathcal{L}(TA)$. Let Z_0 denote the empty zone. Let (l_0, Z_0) be the start state of the timed automata. The locality of a directed search graph generated from the unfolding of a timed automata is defined as

$$\max_{\forall (l,Z),(l',Z') \in \mathcal{S},\ s.t.,\ (l',Z') \in Succ(l,Z)} \{\delta((l_0,Z_0),(l,Z)) - \delta((l_0,Z_0),(l',Z'))\} + 1.$$

Algorithm 8.1 External Breadth-First Search for Reachability Analysis in Timed Automata

Input: A timed automata $TA = (\mathcal{S}, l_0, \mathcal{R}, Inv, \mathcal{T})$; a symbolic initial state (l_0, Z_0).
Output: A path from (l_0, Z_0) to a state (l, Z), s.t. $l \in \mathcal{T}$, if one exists, $\emptyset$ otherwise.

1: $Open(0) \leftarrow \{(l_0, Z_0)\}$ //START WITH THE INITIAL STATE
2: $j \leftarrow 1$
3: **while** $(Open(j-1) \neq \emptyset)$ **do**
4: $A(j) \leftarrow Succ(Open(j-1))$
5: **for all** $(l, Z) \in A(j)$ **do** //ITERATE ON ALL SUCCESSORS
6: **if** $(l \in \mathcal{T})$ **then** //GOAL FOUND
7: **return** *Construct-Solution*(l, Z) //RETURN SOLUTION
8: **end for**
9: $A'(j) \leftarrow$ *sort-and-remove-redundant-zones* $(A(j))$ //DUPLICATES WITHIN THE LAYER
10: **for** $loc \leftarrow 1$ **to** $\mathcal{L}(TA)$ **do** //DUPLICATES SEEN IN PREVIOUS LAYERS
11: $A''(j) \leftarrow A'(j) \setminus \{(l, Z') \in Open(j - loc) \mid (l, Z) \in A'(j),\ Z \subseteq Z'\}$ //REMOVE REDUNDANT ZONES UNDER ENTAILMENT
12: **end for**
13: $Open(j) \leftarrow A''(j)$
14: $j \leftarrow j + 1$
15: **end while**
16: **return** $\emptyset$

In Algorithm 8.1, we depict the pseudo-code of External Breadth-First Search for real-time systems. Starting with the initial state in layer $Open(0)$, the algorithm expands all the nodes of the set $Open(j-1)$ generating the successors in layer j. Duplicates are removed in two steps: first all the redundant zones from within a layer are removed through external

sorting and compaction, and then duplicate zones wrt. $\mathcal{L}(TA)$ previous layers are discarded using external subtraction. The sets A, A', and A'' act as temporary sets. Each set is mapped to a file and a corresponding internal memory buffer. New states are first inserted into the buffer and then flushed to the file once the buffer becomes full.

For a timed automaton *TA* with $\mathcal{S}$ as the set of states and $\mathcal{R}$ the set of transitions in a real-time system, we obtain the following worst-case I/O complexity of External Breadth-First Search.

Theorem 8.1 (I/O Complexity of Real-Time External BFS) *For the problem of symbolic reachability in timed automata, if all zone unions individually fit into the main memory* External Breadth-First Search *can be executed in* $O(\text{sort}(|\mathcal{R}|) + \mathcal{L}(\text{TA}) \cdot \text{scan}(|\mathcal{S}|))$ *I/Os.*

Proof The proof extends the I/O complexity of External Breadth-First Search in undirected graphs. For directed graphs, the duplicate elimination scope is equal to $\mathcal{L}(TA)$, which, in turn, effects the number of layers that we have to scan in order to remove all the duplicates. ■

The memory assumption is almost always fulfilled in practice, as current amounts of main memory can maintain several millions of zones. If some zone unions still fail to fit into main memory, we have to rescan the zone unions in one file repeatedly to compare against the zones of the other file. If the size of the largest zone union is $\mathcal{U}_{\max}$, this will accumulate to

$$O(\mathcal{L}(TA) \cdot \frac{|\mathcal{R}|}{\mathcal{U}_{\max}} \cdot scan(\mathcal{U}_{\max})^2)$$

I/Os in the worst case for checking the duplicates in the previous layer and for compacting a sorted file.

8.3 Linearly Priced Timed Automata

Linearly priced timed automata (LPTA) are timed automata with (linear) cost variables. For the sake of brevity, we restrict their introduction to one cost variable c. Cost increases at states with respect to a predefined rate and in transitions with respect to an update operation. The cost-optimal reachability problem is to find a trajectory that minimizes the overall path costs. Figure 8.2 shows a timed automata with 3 states s_1 (*init*), s_2 (*intermediate*), s_3 (*goal*) with two clock variables x and y and the clock constraints defined on the transitions. The rate of cost variable c is 4 at s_1 and 2 at state s_2. The optimal trajectory is $(d(0), t_1, d(4), t_2)$ that gives a cost of 13 for reaching location s_3. The optimal trajectory can be interpreted as: wait 0 steps in s_1, take t_1 to s_2, where four time steps should be spent before taking the transition to the goal s_3.

Similar to the timed automata, for PTAs we use the notion of priced zone to represent the symbolic states. Let Δ_Z be the unique clock valuation of Z such that for all $u \in Z$ and $\forall x \in C$, we have, $\Delta_Z \leq u(x)$, i.e., Δ_Z is the lowest corner of the $|C|$-dimensional hypersurface representing a zone. In the following, Δ_Z is referred to as the zone offset.

For the internal state representation, we exploit the fact that prices are linear cost hyperplanes of zones. A *priced zone* $\mathcal{Z}$ is a triple (Z, c, r), where Z is a zone, integer c describes the cost of Δ_Z and $r : C \to \mathbb{Z}$ gives the rate for a given clock. In other words, prices of zones are defined by the respective slopes that the cost function hyperplanes have in the direction of the clock variable axes. Furthermore, with $f : \mathcal{Z} \to \mathbb{Z}$, we denote the cost evaluation

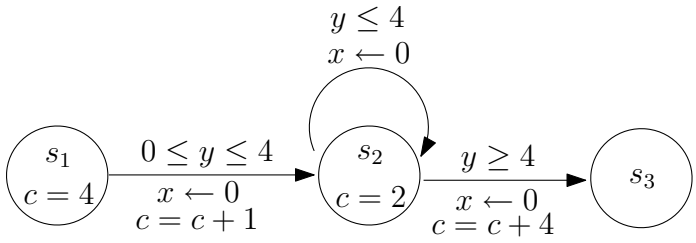

Figure 8.2: A priced timed automata.

function based on priced zones $\mathcal{Z}$. The cost value f for a given clock $x \in C$ in the priced zone $\mathcal{Z} = (Z, c, r)$ can then be computed as $c + \sum_{x \in C} r(x)(v(x) - \Delta_z(x))$. Formally, a priced timed automaton can be described as follows (Larsen *et al.* 2001):

Definition 8.2 (Priced Timed Automaton) *A priced timed automaton* PTA *over clocks C is a tuple $(\mathcal{S}, l_0, \mathcal{R}, \text{Inv}, P, \mathcal{T})$, where*

- *$\mathcal{S}$ is a finite set of automata locations,*
- *$(l_0, \mathcal{Z}_0)$ is the initial state with empty priced zone $\mathcal{Z}_0$,*
- *$\mathcal{R} \subseteq \mathcal{S} \times \mathcal{B}(C) \times \mathcal{P}(C) \times \mathcal{S}$ is the set of transitions, each consisting of a parent state, the guard on the transition, the clocks to reset and the successor state,*
- Inv *assigns invariants to locations,*
- *$P : (\mathcal{S} \cup \mathcal{R}) \to \mathbb{N}$ assigns prices to the states and transitions, and*
- *$\mathcal{T}$ is the set of accepting/target states.*

8.3.1 Cost-Optimal Reachability Analysis in Priced Timed Automata

Algorithm 8.2 shows the cost minimization Branch-and-Bound algorithm that explores a model based on two lists, called the *passed* and the *waiting* list. In agreement with the standard notation in this dissertation, in the following we denote the lists as *Closed* and *Open*, respectively.

In the algorithm $(l_0, \mathcal{Z}_0)$ denotes the start state and l_g the goal condition. Available transition are abbreviated with *Succ*, and $f(\mathcal{Z})$ denotes the cost evaluation based on the priced zone $\mathcal{Z}$. Probably the most expensive operation is the inclusion check $\mathcal{Z} \not\subseteq \mathcal{Z}'$ which tests whether a larger zone with respect to the current one has not yet been considered. For priced zones, we additionally require that the cost values of the stored zones are smaller.

UPPAAL uses difference bounded matrices to store all the constraints of the form $x - y \leq d$ (Alur & Dill 1994). The internal data structure (Behrmann *et al.* 2002) unifies the passed and waiting lists. Its abstract data type has been described as a pair $(P, W) \in 2^S \times 2^S$ with symbolic states S and two access operations *put* and *get*. The implementation of the combined passed-waiting list uses a hash table defined on the discrete part of the states (automata state and the ordinary state variables) to store the states. In each hash table slot, a list of zones, called *zone unions*, is maintained. These zone unions represent the set of states that share the same discrete part but have different zones.

Algorithm 8.2 Branch-and-Bound in UPPAAL-CORA for Cost-Optimal Reachability Analysis

Input: $(l_0, \mathcal{Z}_0)$: The initial state
Output: A path from $(l_0, \mathcal{Z}_0)$ to a state $(l_g, \mathcal{Z}_g)$ s.t. $l_g \in \mathcal{T}$, if one exists.

1: $Cost \leftarrow \infty$
2: $Open \leftarrow \{(l_0, \mathcal{Z}_0)\}$
3: $Closed \leftarrow \emptyset$
4: **while** $(Open \neq \emptyset)$ **do**
5: $\quad (l, \mathcal{Z}) \leftarrow Select(Open)$
6: $\quad$ **if** $(l \in \mathcal{T} \wedge f(\mathcal{Z}) < Cost)$ **then**
7: $\quad\quad Cost \leftarrow f(\mathcal{Z})$
8: $\quad\quad (l_g, \mathcal{Z}_g) \leftarrow (l, \mathcal{Z})$ //SAVE THE BEST GOAL FOUND SO FAR
9: $\quad$ **if** $\forall (l, \mathcal{Z}') \in Closed$: $\mathcal{Z} \not\subseteq \mathcal{Z}'$ **then** //ZONE ENTAILMENT CHECK IS DELAYED TILL EXPANSION
10: $\quad\quad Closed \leftarrow Closed \cup \{(l, \mathcal{Z})\}$
11: $\quad\quad$ **for all** $(l', \mathcal{Z}') \in Succ(l, \mathcal{Z})$ **do** //FOR ALL SUCCESSORS
12: $\quad\quad\quad Open \leftarrow Open \cup \{(l', \mathcal{Z}')\}$ //INSERT SUCCESSORS IN THE *Open* LIST
13: $\quad\quad$ **end for**
14: **end while**
15: **if** $Cost < \infty$ **then**
16: $\quad$ **return** $Construct\text{-}Solution(l_g, \mathcal{Z}_g)$
17: **return** $\emptyset$

Our main objective is to enhance exploration by exploiting memory resources. Bit-string compression of the matrices has been tested, which gives a space performance of 25% to 70%, but the inclusion check becomes expensive (Behrmann *et al.* 2002). For larger gains, either new representations have to be found or further resources have to be uncovered.

8.3.2 External Search in Priced Timed Automata

Until now, we have been mainly discussing external search in directed and unweighted state spaces. But, as we move towards priced real-time systems where timed automata are extended with a cost variable, we find ourselves dealing with a weighted state space. Moreover, we are no longer interested in just any path to a particular goal state, but in an optimal path with respect to our new cost variable.

In priced real-time systems, cost f is a monotonically increasing function implying that for all $(u, v) \in \mathcal{R}$, we have $f(u) \leq f(v)$. If f^* is the optimal solution cost, the following definition captures the notion of cost-optimality for a set of goals $\mathcal{T}$.

Definition 8.3 (Cost-Optimality) *An algorithm is* Cost-Optimal*, if and only if, it terminates with a state $t \in \mathcal{T}$ and $f(t) = f^*$.*

In such directed and weighted graphs, BFS does not guarantee an optimal solution. A natural extension of BFS is to continue the search when a goal is found and keep on searching until a *better* goal is found or the state space is exhausted. A Branch-and-Bound (BnB) search algorithm is an extension to an uninformed search algorithm that does not stop when it finds the first goal, but instead *prunes* all the states that do not improve on the last solution cost. Given that the cost function is monotone, which is the case with f, BnB always terminates with an optimal solution.

The main traversal policy of a Branch-and-Bound algorithm can be borrowed from either Breadth-First, Depth-First, or Best-First Search. A Best-First BnB algorithm, though very well suited for small-sized problems can create a bottleneck for larger problems. Best-First Search picks a state u such that for all $v \in \mathit{Open}$, we have $f(u) \leq f(v)$, for the next expansion. This selection criterion calls for a much larger horizon to be saved in the memory as compared to a Breadth-First or a Depth-First Search. Moreover, both Depth-First and Best-First traversal policies show no locality in the way they expand states – unlike Breadth-First Search, where every state in a layer j is expanded before any state of the layer $j + 1$. This property makes Breadth-First Search a good candidate for Branch-and-Bound.

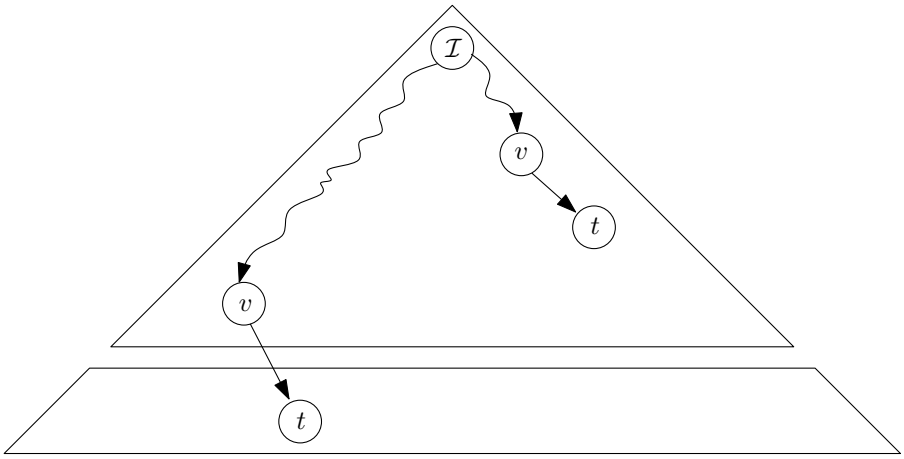

Figure 8.3: Anomaly in the Breadth-First Branch-and-Bound; t is a goal state.

Because of being in a weighted state space, we have to pay an overhead by re-opening already seen states. Consider the following example as illustrated in Figure 8.3 . A Breadth-First Search visits state v for the first time (top right copy) and stores it. Goal state t is also visited and its cost is saved. When the search reaches state v for the second time along a longer path (bottom left copy), but this time with a better cost, v will be pruned away while subtracting previous layers and t will never be reached. If the new path to t has a better cost, we lose our claim for optimality. This anomaly is reported by Lluch Lafuente (2003a). Due to this anomaly, the duplicate detection policy has to be adapted in order to make it compatible with weighted state spaces. Now we are not allowed to remove a duplicate state if its cost is better than what we have seen earlier.

Definition 8.4 (Duplicate State in Priced Timed Automata) *A state $(l, \mathcal{Z})$ is a duplicate state of $(l', \mathcal{Z}')$, if and only if, $l = l'$, $\mathcal{Z} \subseteq \mathcal{Z}'$ and $f(\mathcal{Z}) \geq f(\mathcal{Z}')$.*

In Algorithm 8.3, we formulate our discussion on External Breadth-First Branch-and-Bound in pseudo-code. The set *Open* represents the BFS layer and the sets A, A' and A'' are temporary variables. Initially the goal cost *Cost* is initialized with ∞ and a goal state with a better value is searched for in the successor set $A(j)$. States with a higher value than the best goal cost are pruned and saved in $A'(j)$. In the next step, we remove redundant states based on our definition of duplicate states. The workings of the algorithm is depicted in Figure 8.4. On the x-axis we denote the layers of Breadth-First exploration. Each layer is sorted with increasing cost value. Upon arriving at the first goal t_1, the next layer is pruned

Algorithm 8.3 External Breadth-First Branch-and-Bound for Cost-Optimal Reachability Analysis in Priced Timed Automata

Input: A linearly priced timed automaton $PTA = (\mathcal{S}, l_0, \mathcal{R}, Inv, P, \mathcal{T})$; A symbolic initial state $(l_0, \mathcal{Z}_0)$; $\mathcal{L}(PTA)$ is the locality of PTA.

Output: A path from $(l_0, \mathcal{Z}_0)$ to a state $(l_g, \mathcal{Z}_g)$ s.t. $l_g \in \mathcal{T}$, if one exists, $\emptyset$ otherwise

1: $Cost \leftarrow \infty$; $j \leftarrow 1$ // BEST GOAL COST IS ∞
2: $(l_g, \mathcal{Z}_g) \leftarrow$ **null** //BEST GOAL STATE IS NULL
3: $Open(-1) \leftarrow \emptyset$; $Open(0) \leftarrow \{(l_0, \mathcal{Z}_0)\}$ //START WITH THE INITIAL STATE
4: **while** $(Open(j-1) \neq \emptyset)$ **do**
5: $Temp(j) \leftarrow Succ(Open(j-1))$
6: **for all** $(l, \mathcal{Z}) \in Temp(j)$ **do** //ITERATE ON ALL SUCCESSORS
7: **if** $(l \in \mathcal{T} \wedge f(\mathcal{Z}) < Cost)$ **then** //ANOTHER GOAL FOUND
8: $Cost \leftarrow f(\mathcal{Z})$ //A BETTER GOAL HAS BEEN FOUND. SAVE THE COST.
9: $(l_g, \mathcal{Z}_g) \leftarrow (l, \mathcal{Z})$ //SAVE THE BEST GOAL STATE
10: **end for**
11: $Temp'(j) \leftarrow Temp(j) \setminus \{(l, \mathcal{Z}) \in Temp(j) \mid f(\mathcal{Z}) \geq Cost\}$ // PRUNE THE EXPENSIVE STATES
12: $Temp''(j) \leftarrow$ *sort-and-remove-redundant-zones* $(Temp'(j))$ //DUPLICATES WITHIN THE LAYER
13: **for** $loc \leftarrow 1$ **to** $\mathcal{L}(PTA)$ **do**
14: $Temp''(j) \leftarrow Temp''(j) \setminus \{(l, \mathcal{Z}') \in Open(j - loc) \mid (l, \mathcal{Z}) \in Temp''(j), \mathcal{Z} \subseteq \mathcal{Z}' \wedge f(\mathcal{Z}) \geq f(\mathcal{Z}')\}$ //DUPLICATES SEEN IN PREVIOUS LAYERS
15: **end for**
16: $Open(j) \leftarrow Temp''(j)$
17: $j \leftarrow j + 1$
18: **end while**
19: **if** $Cost \neq \infty$ **then**
20: *Construct-Solution*$(l_g, \mathcal{Z}_g)$ //CONSTRUCT SOLUTION TO THE BEST GOAL FOUND
21: **return** $\emptyset$

to only consider the nodes that have a better cost value. The exploration terminates when the last goal t_4 with the minimal cost value is expanded and no successor of t_4 improves the cost.

The I/O complexity of the External Breadth-First Branch-and-Bound depends on the number of times a state is re-expanded. The worst-case scenario appears when the whole state space fits into one layer and the next layer has the same states but with better cost values. The following theorem states the cost-optimality and I/O complexity of the algorithm.

Theorem 8.2 (I/O Complexity of External BF Branch-and-Bound) *For the problem of cost-optimal symbolic reachability in priced timed automata with monotonic costs, if all zone unions individually fit into the main memory, External Breadth-First Branch-and-Bound is* Cost-Optimal *and can be executed in* $O(D \cdot (sort(|\mathcal{R}|) + \mathcal{L}(\text{PTA}) \cdot scan(|\mathcal{S}|)))$ *I/Os, where* D *is the maximal depth explored.*

Proof Since External Breadth-First Branch-and-Bound expands at least all states $(l, \mathcal{Z})$ with $f(l, \mathcal{Z}) < f^*$, the algorithm terminates with the optimal solution. The *I/O* complexity of the

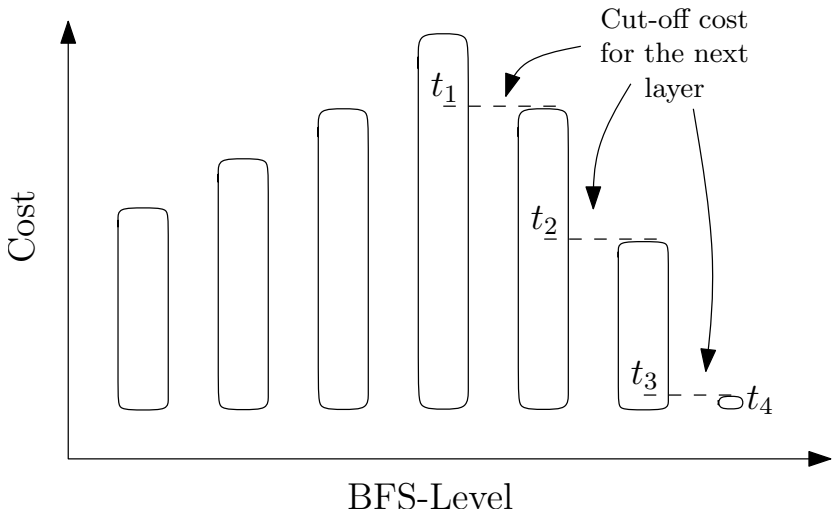

Figure 8.4: A sample run of External Breadth-First Branch-and-Bound; the t_i's represent different goals.

algorithm is inherited from the External Breadth-First Search search (cf. Theorem 8.1). The factor D is introduced due to re-openings. ■

Furthermore, we can say that if there are several goal states in the state space with different solution costs, then an External Breadth-First Branch-and-Bound run will explore at most as many states as a complete External Breadth-First Search run.

Lemma 8.3 *If m is the number of states expanded by External Breadth-First Branch-and-Bound and n is the number of states expanded by a complete exploration of External Breadth-First Search, then $m \leq n$.*

Proof External Breadth-First Branch-and-Bound does not change the order in which states are looked at during a complete External Breadth-First Search exploration. There can be two cases:

1. $|\mathcal{T}| = 1$: There exists just one goal state t which is also the last state in a Breadth-First search tree. For this case clearly $n = m$.

2. $|\mathcal{T}| > 1$: There exists more than one goal state in the search tree. Let $t_1, t_2 \in \mathcal{T}$ be the two goal states with $f(t_1) > f(t_2) = f^*$ and *depth*$(t_1) <$ *depth*(t_2). Since t_1 will be expanded first, $f(t_1)$ will be used as the pruning value during the next iterations. In case there does not exist any state u in the search tree between t_1 and t_2 with $f(u) > f(t_2)$, $n = m$, else $m < n$.

■

The behavior of External Breadth-First Branch-and-Bound largely depends on how fast it reaches to some solution so that it can use that solution cost to further prune away the search space. There exists a very trivial solution to this problem where the user provides some upper bound U on the solution cost that can be used for pruning. In case the upper bound U is actually equal to the optimal solution cost f^*, the algorithm is trivially *Cost-Optimal*.

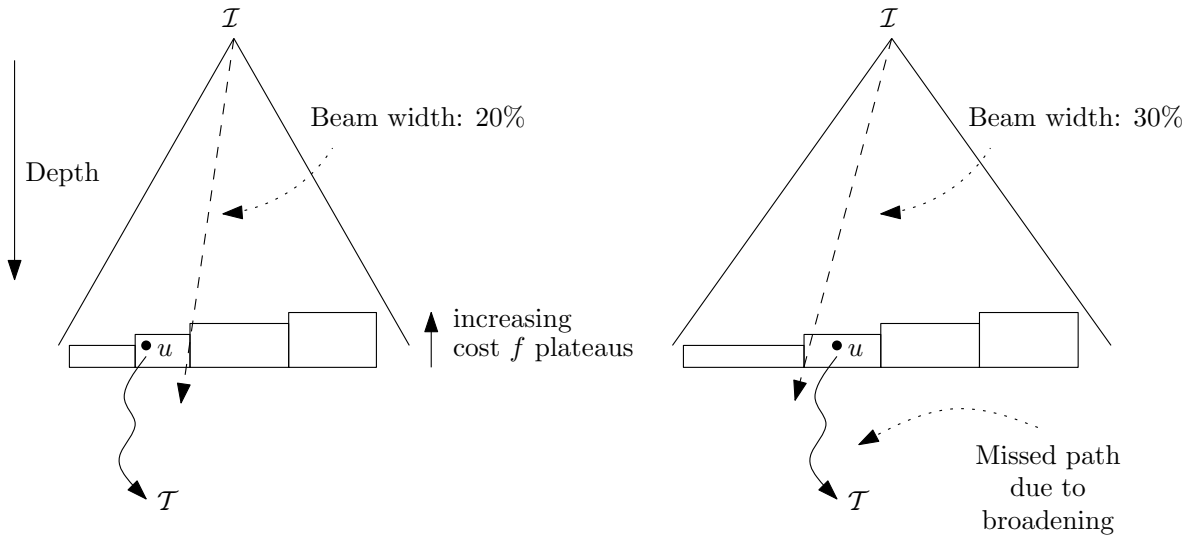

Figure 8.5: Problem with blind iterative broadening. Left beam search arrives at the target but the right one does not due to broadening. Rectangles represent sets of states with same cost values.

Lemma 8.4 *External Breadth-First Branch-and-Bound with $U = f^*$ is* Cost-Optimal.

Proof Since the cost function f in our real-time domain is monotonically increasing, i.e., for all $(u, v) \in \mathcal{R}$, we have $f(u) \leq f(v)$, we will never prune any node that can ultimately take us to the goal node. ■

8.4 Iterative Broadening External Breadth-First Branch-and-Bound

We observe that the efficiency of External Breadth-First Branch-and-Bound is inversely proportional to the factor $U - f^*$. The more realistic the upper bound is, the bigger the pruning and, hence, the lesser the number of expansions. This observation guides us to an iterative strategy to find a good upper bound. We suggest to use only the first $k\%$ of the states when sorted with respect to the increasing cost value and discard the rest of the states in the layer. Hopefully, the algorithm will terminate with a solution, giving us a good upper bound on the optimal solution cost. Using the found solution cost as the upper bound for an increased value of k, we hope to converge to optimal solution cost when k approaches to 100. We will denote the parameter k as the *beam width*.

Unfortunately, there is an apparent problem with this approach. It is possible that for one particular iteration we arrive at a goal state, but at the next iteration we do not. This problem is more frequent in real-time domains, where there can be many different states with the same f-value, residing in a set that has no total order. The algorithm is not guaranteed to converge with increasing k (an exception is when $k = 100\%$ and the whole state space is considered).

Figure 8.5 depicts the mentioned problem. The state u is on the path to the target state $\mathcal{T}$ and is selected in the 20% beam width iteration. But for the next iteration, broadening of

the beam width creates large cost plateaux that shift the state to the right of the next beam selection. A direct consequence is that the state u is not selected and the target is not reached.

Let k_i be the value of k in the ith iteration. For the algorithm to converge, the coverage area of the $(i+1)$th iteration must be at least as large as the coverage area of the ith iteration. Formally, for any layer j,

$$\mathit{Open}_i(j) \subseteq \mathit{Open}_{i+1}(j) \tag{8.1}$$

Such a guarantee can only be given if the maximum cost value that was chosen in the $(i+1)$th iteration for layer j is greater than or equal to the maximum cost value chosen in the i-th iteration. For Condition 8.1 to hold throughout the exploration, we propose the following selection criterion.

Selection Criterion the best $k\%$ states of a layer *plus* all the states that have the same f-value as that of the last state of the selected list *plus* all the states that have a smaller f-value than that of the selected maximum f-value of the last iteration.

With this selection criterion, for a particular cost f', we either choose all the states with a f value equal to f', or choose none.

Figure 8.4 shows the pseudo-code for the actual exploration involving upper bound pruning and the above mentioned selection criteria. The parameters of the algorithms are the beam width k (in percent), the upper bound U and the vector $F_{\max}$ of maximal f-values from the last iteration. With successive iterations, the value of k is increased and the solution cost value of the previous iteration is used as an upper bound. The set *Open* denotes the search frontier, sliced into layers as before. The sets A, A' and A'' are temporary sets, to construct the search frontier for the next iteration. Both the new *Cost* and the new vector of maximal f-values are returned. We use π_n to denote the n-th element in the sorted permutation of a set.

Correctness

Let U_i' be the cost of the solution found by Iterative Broadening External Breadth-First Branch-and-Bound in the ith iteration with $k = k_i$ and $U = U_i$ as the arguments. In the following, we show that the algorithm converges for increasing value of k.

Lemma 8.5 *The selection criterion for Iterative Broadening External Breadth-First Branch-and-Bound guarantees the coverage condition for every iteration i.*

Proof We prove it by induction on the layer j. For $j = 0$, $\mathit{Open}_i(0) \subseteq \mathit{Open}_{i+1}(0)$. Assume that it holds for layer $j-1$ i.e, $\mathit{Open}_i(j-1) \subseteq \mathit{Open}_{i+1}(j-1)$. Generating the successor sets for both sides of the relation yields $\mathit{Succ}(\mathit{Open}_i(j-1)) \subseteq \mathit{Succ}(\mathit{Open}_{i+1}(j-1))$. Removing duplicates from the successor sets on both sides does not change the subset condition. Now we turn to pruning. The selection criterion guarantees that the values $F^j_{\max}$ increase monotonically for increasing value of i, i.e., $F^j_{i,\max} \leq F^j_{i+1,\max}$. Moreover cost plateaux are completely searched. Therefore, pruning does not change the subset condition, so that $\mathit{Open}_i(j) \subseteq \mathit{Open}_{i+1}(j)$. ■

Algorithm 8.4 Iterative Broadening External Breadth-First Branch-and-Bound for Cost-Optimal Reachability Analysis in Priced Timed Automata

Input: A linearly priced timed automaton $PTA = (\mathcal{S}, l_0, \mathcal{R}, Inv, P, \mathcal{T})$; A symbolic initial state $(l_0, \mathcal{Z}_0)$; k: beam width in percent; U: Upper bound; $F_{\max}$: max cost values from the previous iteration; $\mathcal{L}(PTA)$ is the locality of *PTA*.

Output: A path from $(l_0, \mathcal{Z}_0)$ to a state $(l_g, \mathcal{Z}_g) \in PTA$, if one exists.

1: $(l_g, \mathcal{Z}_g) \leftarrow$ **null** //BEST GOAL STATE IS NULL
2: $Cost \leftarrow U$; $j \leftarrow 1$ // BEST GOAL COST IS U
3: $Open(-1) \leftarrow \emptyset$; $Open(0) \leftarrow \{(l_0, \mathcal{Z}_0)\}$ //ALWAYS SART WITH THE INITIAL STATE
4: **while** $(Open(j-1) \neq \emptyset)$ **do**
5: $Temp(j) \leftarrow Succ(Open(j-1))$
6: **for all** $(l, \mathcal{Z}) \in Temp(j)$ **do** //ITERATE ON ALL SUCCESSORS
7: **if** $(l \cap \mathcal{T} \neq \emptyset \wedge f(\mathcal{Z}) < Cost)$ **then** //ANOTHER GOAL FOUND
8: $Cost \leftarrow f(\mathcal{Z})$ //COST OF THE NEW GOAL
9: $(l_g, \mathcal{Z}_g) \leftarrow (l, \mathcal{Z})$ //SAVE THE BEST GOAL STATE
10: **end for**
11: $Temp'(j) \leftarrow Temp(j) \setminus \{(l, \mathcal{Z}) \in Temp(j) \mid f(\mathcal{Z}) \geq Cost\}$ // PRUNE THE EXPENSIVE STATES
12: $Temp''(j) \leftarrow$ *remove redundant zones within* $Temp'(j)$ //DUPLICATES WITHIN THE LAYER
13: **for** $loc \leftarrow 1$ **to** $\mathcal{L}(PTA)$ **do** //DUPLICATES SEEN IN PREVIOUS LAYERS
14: $Temp''(j) \leftarrow Temp''(j) \setminus \{(l, \mathcal{Z}') \in Open(j - loc) \mid (l, \mathcal{Z}) \in Temp''(j), \mathcal{Z} \subseteq \mathcal{Z}' \wedge f(\mathcal{Z}) \geq f(\mathcal{Z}')\}$
15: **end for**
16: $Temp''(j) \leftarrow$ *External-sort* $Temp''(j)$ *w.r.t the cost function* f
17: $n \leftarrow \lfloor (k \cdot |Temp''(j)|)/100 \rfloor$ //THERE ARE n STATES IN THE BEST $k\%$
18: $(l_n, \mathcal{Z}_n) \leftarrow \pi_n(Temp''(j))$ //PICK THE n-TH STATE
19: $F^j_{\max} \leftarrow \max\{F^j_{\max}, f(\mathcal{Z}_n)\}$ //COMPUTE THE NEW MAX F VALUE FOR THE LAYER
20: $Open(j) \leftarrow \{(l, \mathcal{Z}) \in Temp''(j) \mid f(\mathcal{Z}) \leq F^j_{\max}\}$ //KEEP ONLY THE *best* STATES
21: $j \leftarrow j + 1$
22: **end while**
23: **if** $(Cost < U)$ **then** //IF THE BOUND HAS IMPROVED CONSTRUCT THE SOLUTION
24: **return** *Construct-Solution*$(l_g, \mathcal{Z}_g)$
25: **return** $(Cost, F_{\max})$ //RETURN NEW UPPER BOUND

Lemma 8.6 *For all iterations i in Iterative Broadening External Breadth-First Branch-and-Bound, we have $U'_{i+1} \leq U'_i$.*

Proof Since the coverage area of iteration $i+1$ is larger than the coverage area of iteration i, in the worst case it does not improve on the solution quality i.e., $U'_{i+1} = U'_i \leq U_i$, else we have $U'_{i+1} \leq U'_i \leq U_i$. In both cases, $U'_{i+1} \leq U'_i$. ■

Theorem 8.7 (Optimality of Iterative Broadening) *Iterative Broadening External Breadth-First Branch-and-Bound converges to the optimal solution.*

Proof Lemma 8.5 provides the necessary ground for the coverage of whole state space, which implies the completeness of the algorithm and Lemma 8.6 provides the convergence to the optimal solution cost that proves its optimality. ■

8.5 Aircraft Landing Scheduling Problem

The aircraft landing scheduling problem, first presented by (Beasley *et al.* 2000), can be viewed as an instance of the job-shop scheduling paradigm. Given a set of aircrafts and a set of runways, the task is to allot a landing schedule to each of the aircraft. Each aircraft i has an earliest landing time E_i and a latest ending time L_i along with a target (preferred) landing time T_i. Their are penalties attached to the time elapsed between E_i and T_i, and between T_i and L_i.

Moreover, each plane creates some turbulence when it lands and this dictates a gap between successive landings of two airplanes. These gaps are different for different kinds of airplanes and hence provided for every pair of airplanes.

The task is to allocate individual landing times t_i to each plane $i \in N$ such that: *a*) landing penalties are minimized by making every aircraft land at some time within a predetermined time window, *b*) separation constraints are respected between the landings of successive aircrafts.

Formally, let g_i (h_i) denotes the penalty cost for plane i landing ahead of (after) the target time and S_{ij} denotes the separation time between aircraft i and j if i lands before j. Furthermore, let b_{ij} be a binary variable indicating that aircraft i lands before j and M be a very large number to control that each plane is landed at most once. Since the penalties are linear to the time, the aircraft landing problem can thus be formulated as a linear programming problem:

$$\textbf{\textit{minimize}} \sum_{i \in N} g_i \; max\{0, T_i - t_i\} + h_i \; max\{0, t_i - T_i\}$$

$$\begin{aligned}
\textit{Subject to:} \quad t_i S_{ij} &\leq t_j + M b_{ji} && \forall i \neq j \in N, && (8.2)\\
b_{ji} + b_{ij} &= 1 && \forall i \neq j \in N, && (8.3)\\
E_i \leq t_i &\leq L_i && \forall i \in N, && (8.4)\\
b_{ij} &\in \{0, 1\} && && (8.5)
\end{aligned}$$

Ernst, Krishnamoorthy, & Storer (1999) discussed a number of possible solution algorithms, some of which are suitable for an approximate solution, while some guarantee an exact solution. The exact solution is based on a variant of Simplex algorithm.

Behrmann, Larsen, & Rasmussen (2005) presented a modeling of the aircraft scheduling problem in the language specification used by UPPAAL-CORA. It involves considering a timed automaton for each of the airplane and runways. The problem can then be seen in the light of model checking as a reachability analysis problem involving the minimization of cost.

8.6 Experiments

We have implemented the algorithms External Breadth-First Branch-and-Bound, and Iterative Broadening External Breadth-First Branch-and-Bound on top of UPPAAL-CORA. Our implementation also extends UPPAAL making it capable of performing External Breadth-

First Search in timed automata. The main memory requirements are kept constant[†]. Hash tables are replaced by files on hard disk with a small internal buffer for I/O efficiency. As the maximum file size on most file systems is 2GB, we also provide large file support, that splits files if they become too large. Trails for found solutions are reconstructed by saving the predecessor together with every state, by using backtracking along the stored files, and by looking for matching predecessors. This results in a I/O complexity that is at most linear to the number of stored states.

A limited functionality (which nonetheless does not compromise the correctness of the approach) of the current implementation is in duplication detection. We remove duplicates from the internal buffer before flushing it but the duplicates within different flushed buffers are not merged and are not subtracted from the previous layers. All experiments are run on a Pentium-4 with 150 GB of hard disk space and 2GB RAM running Linux. We choose different instances of aircraft landing scheduling (ALS) for validation of our approach.

We start with a smaller instance involving just 1 runway and 10 planes. Table 8.1 (left) provides the results of running Iterative Broadening External Breadth-First Branch-and-Bound. Here k denotes the coverage, U the initial bound and U' the optimal solution obtained. The effects of pruning on the number of expanded states are quite evident. We also see a converging behavior of the algorithm. In the last row we report the results for External Breadth-First Branch-and-Bound to show the effect of pruning on the search space. Our result matches with the one found by internal memory algorithm in UPPAAL-CORA. In Table 8.1 (right) we illustrates the results for a more difficult instance. For this instance, we created two independent sets of automata encoding runways and planes. We then instantiated 1 runway and 10 planes from the first set and 1 runway and 10 planes from the other set. UPPAAL-CORA with internal BnB could not solve the instance because of memory requirements. Being a doubled variant of the first instance, the solution had to be 1400, which validates our implementation. With Iterative Broadening, we were able to find an optimal solution. On the other hand, External Breadth-First Branch-and-Bound could not finalize its execution in two hours consuming about 3 GB with 280 bytes per state, while expanding depth 19 - optimal solution lies at depth 40. The process was manually killed.

k	U	U'	$Expanded$
1	∞	970	91
20	970	970	91
40	970	810	125
60	810	710	281
80	710	700	439
100	700	700	577
100	∞	700	31,458

k	U	U'	$Expanded$
0.1	∞	1940	1,060
20	1940	1940	1,285
40	1940	1420	18142
60	1420	1410	69,341
80	1410	1410	147,128
100	1410	1400	195,145
100	∞	—	—

Table 8.1: Aircraft Landing Scheduling Problem with 1 runway and 10 planes (left), and with 2 runways and 20 planes (right).

For the third instance, we chose another instance of aircraft scheduling problem that was obtained by a translation from PDDL planning models (Dierks 2005). The internal version of UPPAAL-CORA failed to reach any solution for 3 planes and after quickly consuming about

[†]Up to a leak of at most 100 MB per hour.

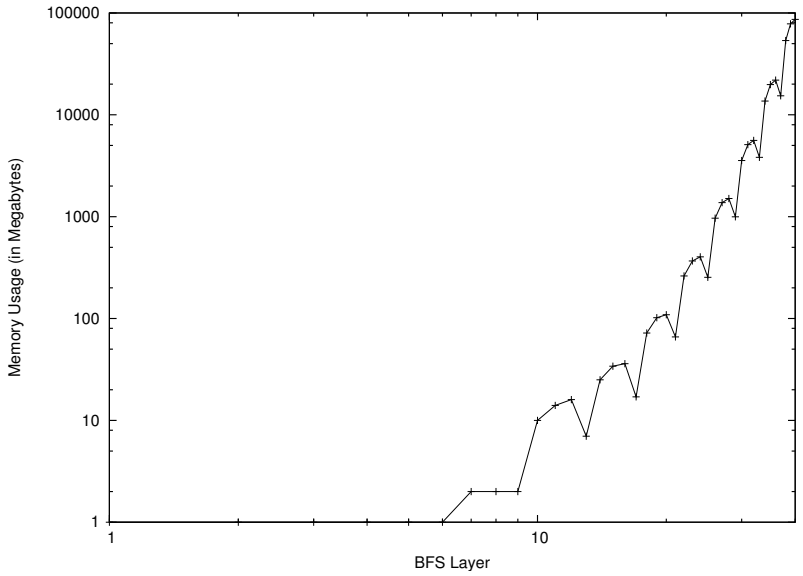

Figure 8.6: Space consumption for each BFS Layer.

1.6 GB of main memory started to swap on hard disk. For this instance just for 3 planes a total of 13 clocks were used. Our iterative broadening strategy, for $k < 100$ didn't produce any solution. For $k = 100$, the algorithm ran for about 12 hours consuming a total of 311 GB and ran out of hard disk space using a mere 2KB per state. On a hard disk with just 150 GB available, this was achieved by removing the previous layers manually. Up till the 40th layer there was no solution. In Fig. 8.6, we depict the graph where space consumption for each layer is shown. The internal size of the program remained under 1.8 GB.

8.7 Related work

Iterative Broadening has been introduced by (Ginsberg & Harvey 1992). The Breadth-First BnB approach is related to Breadth-First Heuristic Search (BFHS) (Zhou & Hansen 2004a), a frontier search method that was designed to save internal memory (cf. Chapter 3). It is based on the observation that a Breadth-First Search frontier is often much smaller than a Best-First Search frontier. A recent extension of BFHS is its integration with beam search known as Beam-Stack Search (Zhou & Hansen 2005). As it iterates on different beams, this algorithm is a natural competitor for Iterative Broadening External Breadth-First Branch-and-Bound. This algorithm is also guaranteed to continuously converge. There are several differences to our approach. The beam width in Beam-Stack Search is driven by the limits of main memory (previous layers can be flushed to the hard disk). Such a limit is not needed in our case, as we exploit the secondary storage. Moreover, a backtracking strategy is employed to pick more elements from the previous layer in case the upper bound is not improved. Several variants of beam search have also been widely used in model checking. Recently, Wijs (2007) proposed a wide range of beam search algorithms for model checking μ-CRL systems.

8.8 Summary

This chapter discussed external search for model checking of real-time systems represented by timed automata. The results are then extended to external exploration in priced timed automata domains. We contribute Iterative Broadening External Breadth-First Branch-and-Bound that stores each Breadth-First level on the hard-disk. The algorithm achieves its completeness by trying to find an upper bound on the optimal solution in an incomplete search tree. Iteratively, the upper bound is made tighter and the coverage of the search space is increased. The correctness and completeness proofs for the suggested algorithms are presented along with experimental results on different instances of aircraft landing scheduling to validate the practicality of our approach. Having performed an exploration of more than a quarter of a Terabyte, we believe to have pushed the limits of practical scheduling and model checking in real-time domains.

As external exploration realizes a controlled streamed access to states, there is also potential for a parallel implementation. A parallel and distributed reachability checking algorithm of UPPAAL based on the *Message Passing Interface* (MPI) partitions the list of explored states using a simple hash function (Behrman, Hune, & Vaandrager 2000). It restricts itself to blind exploration. Using techniques developed in Chapter 7, the external algorithms for timed automata exploration can be extended to work in a distributed environment.

We have not talked about heuristic search, although the UPPAAL-CORA models incorporate hand-coded search heuristics to accelerate the exploration. A recent proposal to generate heuristics for UPPAAL automatically has recently been provided by (Kupferschmid *et al.* 2006). Given admissible heuristics, the pruning in Branch-and-Bound can be accelerated, while consistent heuristics will allow us to use External A* for real-time systems.

Part II

Action Planning

Chapter 9

Deterministic Planning

In Artificial Intelligence, *planning* refers to the process of finding a sequence of actions that when applied to a given initial state transforms it into a desired goal state. In recent years, AI Planning has seen significant growth in both theory and practice. Most of these approaches revolve around *search*. The underlying transition graph is generated on-the-fly while searching for the state(s) where desired goal criteria are fulfilled. *PDDL* (Planning Domain Definition Language) (McDermott & others 1998) provides a common framework for defining planning domains and problems. Starting from a purely propositional framework, it has now grown into accommodating more complex planning problems. Planning also suffers from the same *state space explosion* problem. For large planning problems, limited RAM can be the major hurdle while searching for a solution. In this chapter, we suggest some External Memory algorithms for planning in deterministic setting, i.e., where the outcome of an action is completely defined. The non-deterministic and probabilistic settings are covered in the next chapter.

Structure of the chapter: First and foremost, we formulate the planning problem and explain the notion of a state in planning. Starting from a purely propositional formalism, we extend it to metric planning problems with linear expressions. We also give a brief overview of the new extensions to PDDL planning and discuss a recently proposed solution for dealing with them. Introducing heuristic guidance in the form of relaxed plan computation, we discuss an EM variant of Enforced Hill Climbing for *suboptimal* planning. This is followed by an algorithmic solution for the *optimality* problem. To bound the *duplicate detection scope* in planning graphs, we provide the theoretical basis for an automated algorithm. Finally, we present some large exploration results to validate our approach.

9.1 Planning Problem

A deterministic planning problem can be regarded as a state space exploration problem in implicit graphs. A vertex or a state in this graph consists of a set of *facts* and *fluents* that are assigned to boolean or numerical values in that state. Successors are generated by applying operators/actions.

A planning task is mostly defined using a well-established formalism called PDDL (Planning Domain Definition Language) (Long & Fox 2003). PDDL has its roots in STRIPS (Stanford Research Institute Problem Solver) and ADL (Action Description Language). STRIPS (Fikes & Nilsson 1971) is a propositional formalism that defines the *facts* that hold in a finite environment. A state in STRIPS is a conjunction of a subset of these *facts*. An *action* or an *operator* is enabled, if its pre-conditions (also defined as facts) hold in a state. Upon execution, some of the facts are made `false`, i.e., removed from the state, while new facts are added to it. For example, an environment containing a grid of rooms can be described by the connections of rooms with each other (doors). Similarly, an object, say a robot, can be placed inside a room by a fact defining the position of the robot. A planning operator for '*moving*' the robot to an adjacent room, is executed by removing the *fact* of the current position and inserting the new position as a *fact* inside the state.

ADL (Action Description Language) (Pednault 1991) was an effort to integrate the theory of Situation Calculus in the STRIPS formalism. Facts in ADL can be defined not only in propositional logic, but also in first-order predicate logic. It also introduced negations, quantifications, disjunctions, and conditional effects in STRIPS. Conditional effects gave a new meaning to the operators. They allowed to have various effects of the same operator depending on the state on which they are applied. As described earlier, PDDL is a combination of both STRIPS and ADL. It explicitly distinguishes between a planning *domain* and a planning *problem*. A domain here is the description of the predicates, the types of variables, and the set of operators. A problem, on the other hand, consists of the initial state and the required goal. A PDDL problem can be translated back into STRIPS by instantiating all the predicates – a process commonly known as *grounding* in the planning community.

Definition 9.1 (Propositional Planning) *A fully instantiated (grounded) propositional planning task $\mathcal{P}$ is a four-tuple, $\mathcal{P} = \langle AP, \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$, where*

- *AP is the set of atomic propositions* (facts)*,*
- *$\mathcal{I} \subseteq AP$ is the initial state,*
- *$\mathcal{T} \subseteq AP$ the set of goal states, and*
- *$\mathcal{O}$ the set of planning actions.*

Definition 9.2 (Propositional Action) *A propositional action a is a three-tuple ($pre(a)$, $\mathit{eff}(a)^+$, $\mathit{eff}(a)^-$), where*

- *$pre(a) \subseteq AP$ is a set of pre-conditions for applying a,*
- *$\mathit{eff}(a)^+ \subseteq AP$, is a set of new facts to be added to the new state, and*
- *$\mathit{eff}(a)^- \subseteq AP$ is a set of facts to be deleted from the new state.*

An action a is applicable on a state u, if $pre(a) \subseteq u$. The successor state u' is generated as

$$u' = \text{Succ}(u) = (u \setminus \mathit{eff}(a)^-) \cup \mathit{eff}(a)^+$$

Definition 9.3 (Solution to a Planning Problem) *A solution to a planning task is a sequence of actions $a_1, a_2, \ldots, a_n$ that when applied to the initial state $\mathcal{I}$ transforms it into a goal state $t \in \mathcal{T}$.*

```
(define (domain petri)
(:requirements :typing :strips)
(:types transition place - object)
(:predicates (incoming ?p - place ?t - transition)
             (outgoing ?t - transition ?p - place)
             (blocked  ?t - transition)
             (marked   ?p - place)
)

(:derived (blocked ?t - transition)
   (exists (?p - place)
        (and (incoming ?p ?t)
             (not (marked ?p)))))

(:derived deadlock (forall (?t - transition)
                           (blocked ?t)))

(:action fire-transition
:parameters (?t - transition)
:precondition
  (forall (?p - place)
    (or (not (incoming ?p ?t))
        (marked ?p)))
:effect
  (and
    (forall (?p - place)
      (when (and (incoming ?p ?t)
                 (marked ?p))
            (not (marked ?p))))

    (forall (?p - place)
      (when (and (outgoing ?t ?p)
                 (not (marked ?p)))
            (marked ?p)))
  )
))
```

```
(define (problem philosopher)
(:domain petri)
(:objects
   p11 p12 p21 p22 p31 p32 - place
   t11 t12 t21 t22 t31 t32 -
                       transition)
(:init
  (incoming p11 t11)
  (incoming p12 t12)

  (incoming p21 t21)
  (incoming p22 t22)

  (incoming p31 t31)
  (incoming p32 t32)

  (incoming p11 t22)
  (incoming p12 t21)

  (outgoing t11 p21)
  (outgoing t12 p22)

  (outgoing t21 p31)
  (outgoing t22 p32)

  (outgoing t31 p11)
  (outgoing t32 p12)

  (outgoing t31 p12)
  (outgoing t32 p11)

  (marked p11)
  (marked p12)
)
(:goal
  (deadlock)))
```

Figure 9.1: Deadlock detection problem in 1-safe Petri nets as a PDDL domain (left) and its corresponding problem (right)

The objective of optimal propositional planning is to find the plan with the minimal number of actions.

Propositional planning is decidable but (PSPACE) hard (Bylander 1994).

Example In Figure 9.1, we see an example of a planning domain and its associated problem defined in PDDL. The domain models 1-safe Petri nets taken from a model checking problem (Edelkamp & Jabbar 2006b). Being 1-safe, each place can only have either just one token (marked) or none. The structure of the Petri net is modeled by using the incoming and outgoing arcs between the places and the transitions. The example Petri net as a PDDL *problem* is depicted in the right column of the figure. It models a deadlock detection problem of two dining philosophers.

9.2 Metric Planning

Metric planning (Fox & Long 2003) involves reasoning about continuous state variables and arithmetic expressions. It is a numerical extension to the STRIPS planning formalism, where actions modify the value of numeric state variables. The task is then to

- find a path from an initial state to a state where all goal criteria are fulfilled, and
- *additionally* optimize an objective function.

Definition 9.4 (Metric Planning) *A metric planning task $\mathcal{P}_\mathcal{C}$ is a 6-tuple, $\mathcal{P}_\mathcal{C} = \langle \mathcal{V}, AP, \mathcal{O}, \mathcal{I}, \mathcal{T}, \mathcal{C} \rangle$, where:*

- *$\mathcal{V} = \{v^1, \ldots, v^k\}$ is the set of numerical state variables (*fluents*) from domains $D_1, D_2, \ldots D_k \subseteq \mathbb{R}$*
- *$\mathcal{O}$ is the set of extended planning actions,*
- *$\mathcal{I}$ is the extended initial state,*
- *$\mathcal{T}$ is the extended planning goal, and*
- *$\mathcal{C}$ is the domain cost metric to be optimized.*

Definition 9.5 (State in Metric Planning) *An extended planning state s_n is a pair $(p(s_n), v(s_n))$, where $p(s_n) \subseteq AP$ and $v(s_n) = (v^1(s), \ldots, v^k(s)) \in D_1 \times \ldots \times D_k$.*

An action *action* is annotated by a set of numerical preconditions (*condition*(*action*) and a set of numerical effects *effect*(*action*), both defined over arithmetic expressions. Extended goals also contain conditions.

Definition 9.6 (Action Condition) *A condition is a 3-tuple (exp, comp, exp'), where exp and exp' are arithmetic expressions over the set of operators $\{+, *, -, /\}$ and $\text{comp} \in \{<, \leq, =, \geq, >\}$. For the evaluation of an expression exp in state s, its evaluation is recursively defined as*

$$exp(s) \quad = \quad \begin{cases} \text{const} & \text{if } exp = \text{const} \\ v^i(s) & \text{if } exp = v^i \\ exp'(s) + exp''(s) & \text{if } exp = exp' + exp'' \\ exp'(s) exp''(s) & \text{if } exp = exp' exp'' \\ exp'(s) - exp''(s) & \text{if } exp = exp' - exp'' \\ exp'(s) / exp''(s) & \text{if } exp = exp' / exp'' \end{cases}$$

Definition 9.7 (Assignment) *An assignment is a pair (v, exp), where exp is an arithmetic expression, and v is a state variable. A successor s' of a state $s = (v^1, \ldots, v^k)$ in Metric planning is defined with respect to the pair (v^i, exp) as $s' = (v^1, \ldots v^{i-1}, exp(s), v^{i+1}, \ldots, v^k)$.*

The domain cost metric $\mathcal{C}$ is an expression that is evaluated in an end state of a plan that satisfies the goal condition.

Metric planning (over infinite variables) is a considerable extension in language expressiveness. As one can encode arbitrary Turing machine computations in real numbers, even the decision problem becomes undecidable (Helmert 2002). Nonetheless, this does not mean

```
(:action fire-transition
 :parameters (?t - transition)
 :preconditions
   (forall (?p - place)
     (or (not (incoming ?p ?t))
         (> (number-of-tokens ?p) 0)))
 :effects
   (forall (?p - place)
     (when (incoming ?p ?t)
           (decrease (number-of-tokens ?p))))
   (forall (?p - place)
     (when (outgoing ?t ?p)
           (increase (number-of-tokens ?p)))))
```

Figure 9.2: Numerical planning operator of a Petri net transition.

that metric planning is impossible, as solution finding can succeed in semi-decidable problem classes. Moreover, the class of actual benchmark problems is simpler. For many cases, even finite domain encodings for numerical variables can be found. For this text, we restrict preconditions and effects to be linear expressions. This is not a severe limitation in planning practice, as all planning benchmarks released so far can be compiled to a linear representation (Hoffmann 2003a).

In explicit-state planning, the domains D_i are arbitrary reals (floating point numbers), while in symbolic search planning with BDDs, the D_i's have to be restricted to finite domains that can be encoded in binary. The latter assumption implies that the problem can, in principle, be specified as a propositional planning problem.

Example In Figure 9.2, the `fire-transition` operator is modeled again to deal with general Petri nets. The predicate `number-of-tokens` replaces the `marked` predicate. The firing of each transition decreases the number of tokens from the incoming arcs and increases the tokens in the places connected to the outgoing arcs.

9.3 Planning with Preferences

State trajectory and preference constraints are two language features introduced in PDDL3 (Gerevini & Long 2005) for describing benchmarks of the 5^{th} international planning competition (IPC-5). *State trajectory constraints* provide an important step of the agreed fragment of PDDL towards the description of *temporal control knowledge* (Bacchus & Kabanza 2000; Kabanza & Thiebaux 2005) and *temporally extended goals* (DeGiacomo & Vardi 1999; Lago, Pistore, & Traverso 2002; Pistore & Traverso 2001). They assert conditions that must be satisfied during the execution of a plan. *Preferences*, on the other hand, assert conditions that are *desirable* to be satisfied by a plan, but can be skipped by paying a penalty in the plan quality.

Example If we prefer that at the end of the plan, place `p22` should have a token, we write `(preference p (marked p22))` with a validity check `(is-violated p)` in the plan objective.

Using the approach of Edelkamp (2006), such preferences are compiled into variables over natural numbers and are included into the plan's *cost function*. This (linear) function allows planners to search for cost-optimal plans. For planning with preferences, the cost function first scales and then accumulates the numerical interpretation of the propositions referring to the violation of the preference constraints. The planning goal now corresponds to finding a plan with minimum preferences' violations. This reduction allows us to use the theory developed for metric planning (PDDL2) directly on the planning problems written in PDDL3. For detailed information on the compilation, we refer the reader to (Edelkamp 2006) and (Edelkamp, Jabbar, & Nazih 2006).

9.4 External Search in Planning

9.4.1 External Enforced Hill Climbing

Enforced Hill Climbing (EHC) is a *conservative* variant of Hill Climbing search. Starting from a start state, a Breadth-First Search is performed for a successor with a better heuristic value. As soon as such a successor is found, the hash tables are cleared and a fresh BFS is started. The process continues until the goal is reached. Since EHC performs a complete BFS on every state with a strictly better heuristic value, for directed graphs without dead-ends, Enforced Hill Climbing is complete and guaranteed to find a solution (Hoffmann & Nebel 2001).

Algorithm 9.1 External Enforced Hill Climbing

Input: A planning problem $\mathcal{P} = \langle AP, \mathcal{O}, \mathcal{I}, \mathcal{T}, \rangle$. $\mathcal{G}(\mathcal{P})$ is the underlying transition graph generated on-the-fly.
Output: A plan to the problem $\mathcal{P}$, if one exists.

1: $u \leftarrow \mathcal{I}$, $h = \mathit{Heuristic}(\mathcal{I})$
2: **while** $(h \neq 0)$ **do** //A GOAL HAS $h = 0$
3: $(u', h') \leftarrow \mathit{External\text{-}EHC\text{-}BFS}(u, h)$ //SEARCH FOR A BETTER STATE
4: **if** $(h' = \infty)$ **then**
5: **return** $\emptyset$
6: $u \leftarrow u'$
7: $h \leftarrow h'$
8: **end while**
9: **return** $\mathit{ConstructSolution}(u)$

Having External BFS in hand, an external algorithm for Enforced Hill Climbing can be constructed by utilizing the heuristic estimates and limiting the subtraction of previous layers to the locality of the graph as computed in the previous section. Algorithm 9.1 depicts the External Enforced Hill Climbing algorithm (Edelkamp & Jabbar 2006a). The externalization is embedded in the sub-procedure (Algorithm 9.2) that performs External BFS for a state with better heuristic estimate. Figure 9.3 shows an example execution of External EHC.

As heuristic guidance, one can chose relaxed plan heuristics for metric domains (Hoffmann 2003a). The metric version of the propositional relaxed plan heuristic analyzes an extended layered plan graph, where each fact-layer includes the encountered propositional atoms and numeric fluents. The forward construction of the plan graph iteratively applies operators until all goals are satisfied. The length of this relaxed plan is then used as a heuris-

Algorithm 9.2 External EHC-BFS

Input: A pair (u, h); u is the new start state with the heuristic estimate h
Output: A pair (u', h') such that $h' < h$, if one exists, $(\emptyset, \infty)$ otherwise

1: $Open(-1, h) \leftarrow \emptyset, Open(0, h) \leftarrow u, i \leftarrow 1$
2: **while** $(Open(i-1, h) \neq \emptyset)$ **do**
3: $Temp(i) \leftarrow Succ(Open(i-1, h))$
4: **for all** $v \in Temp(i)$ **do**
5: $h' = Heuristic(v)$
6: **if** $h' < h$ **then** //HAS A STATE WITH A BETTER HEURISTIC VALUE BEEN FOUND?
7: **return** (v, h')
8: **end for**
9: $Temp'(i) \leftarrow$ *sort-and-remove-duplicates* $(Temp(i))$
10: **for** $loc \leftarrow 1$ **to** $\mathcal{L}(\mathcal{G}(\mathcal{P}))$ **do** //SUBTRACT $\mathcal{L}(\mathcal{G}(\mathcal{P}))$ PREVIOUS LAYERS
11: **if** $i - loc < 0$ **then** break //SUBTRACT ONLY LEGAL LAYERS
12: $Temp'(i) \leftarrow Temp'(i) \setminus Open(i - loc)$
13: **end for**
14: $Open(i) \leftarrow Temp'(i)$
15: $i \leftarrow i + 1$
16: **end while**
17: **return** $(\emptyset, \infty)$

tic estimate to guide the search. The heuristic is neither admissible nor consistent, but very effective in practice.

Theorem 9.1 (I/O Complexity of External Enforced Hill Climbing) *Given a planning problem $\mathcal{P} = \langle AP, \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$, if $\mathcal{P}$ is deadlock-free, External Enforced Hill Climbing as depicted in Algorithm 9.1 and 9.2 will find a solution with at most*

$$O\left(h(\mathcal{I}) \cdot (sort(|\mathcal{R}|) + \mathcal{L}(\mathcal{G}(\mathcal{P})) \cdot scan(|\mathcal{S}|))\right) \text{ I/Os,}$$

where $h(\mathcal{I})$ is the heuristic value of the initial state, $\mathcal{L}(\mathcal{G}(\mathcal{P}))$ is the locality of the underlying transition graph of $\mathcal{P}$, with $\mathcal{S}$ and $\mathcal{R}$ denoting the states and transitions in the graph $\mathcal{G}(\mathcal{P})$.

Proof Algorithm 9.1 does not change the order of traversal when compared with the internal memory variant. Given that the planning graph is deadlock-free, the result in (Hoffmann & Nebel 2001) on the completeness of Enforced Hill Climbing can be adapted directly to the External EHC. Hence, Algorithm 9.1 will eventually terminate with a solution to the planning problem $\mathcal{P}$.

The algorithm performs a complete External BFS for every new lower value of heuristic encountered. Since the graph is directed, the I/O complexity of Breadth-First Search is bounded by

$$O(sort(|\mathcal{R}|) + \mathcal{L}(\mathcal{G}(\mathcal{P})) \cdot scan(|\mathcal{S}|)) \text{ I/Os.}$$

The term $sort(|\mathcal{R}|)$ is due to external sorting the list of successors, while the $\mathcal{L}(\mathcal{G}(\mathcal{P})) \cdot scan(|\mathcal{S}|)$ I/Os are required for subtracting the elements already expanded in the previous layers.

Starting from a heuristic value of $h(\mathcal{I})$, the algorithm searches for a new state with a lower heuristic value, say a *seed* state. In the worst case, each seed state differs by exactly

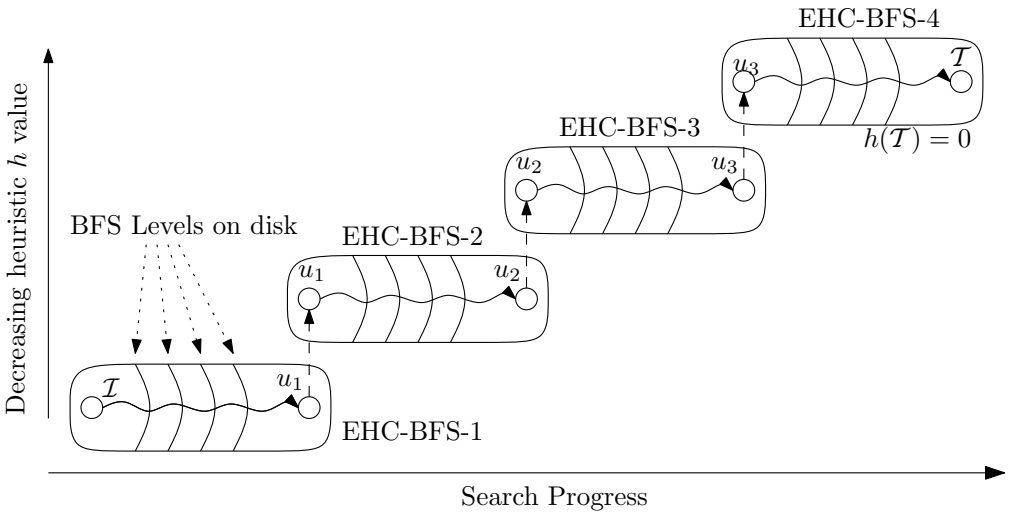

Figure 9.3: External Enforced Hill Climbing. Each rectangle correspond to one call of External-EHC-BFS algorithm. Dashed arrows are just for the sake of clarity to identify the the start of a new BFS from the newly found state u_i with $h(u_{i-1}) > h_{(u_i)}$.

one from the previous seed state in heuristic value. Consequently the external BFS has to be repeated at most $h(\mathcal{I})$ times. ■

9.4.2 Cost-Optimal External BFS

In metric planning, one often encounters a monotonic increasing cost function, in which the cost is accumulated along the path. Hence, when searching for the optimal plan, it is sufficient to use a branch-and-bound like procedure to prune the whole sub-tree where the plan cost is going to increase.

While planning in the presence of *preference constraints*, particularly for *goal preferences*, this monotonicity no longer holds. We reported in an earlier section that goal preferences are compiled into variables. Whenever a goal preference p is violated in a goal state, we have to pay a penalty in terms of a cost assigned to the variable corresponding to p. Let $\mathcal{C}(t_1)$ be the cost of that goal $t_1 \in \mathcal{T}$.

Now, this does not mean that while extending the plan and reaching a new goal state, the preference would still be violated. It can happen that by extending the plan, we encounter a new goal state $t_2 \in \mathcal{T}$ such that $\mathcal{C}(t_2) < \mathcal{C}(t_1)$. The result is a monotonically decreasing cost function that makes a branch-and-bound like procedure inapplicable. Essentially, we are forced to look at all states for finding an optimal plan.

Algorithm 9.3 displays the pseudo-code for external BFS exploration incrementally improving an upper bound U on the plan cost. Only the improved goals are reported. The algorithm is tricky when it comes to analyzing a layer in which more than one goal is contained. In this case, one goal $t \in \mathcal{T}$ with the minimum value $\mathcal{C}(t)$ has to be selected for solution reconstruction. First we compute the minimum and maximum value ($\min_{\mathcal{C}}$ and $\max_{\mathcal{C}}$) that $\mathcal{C}$ can take. If $\mathcal{C}$ is a linear function $\sum_{i=1}^{k} a_i v_i$ with $a_i \geq 0$, $i \in \{1, \ldots, k\}$ then $\min_{\mathcal{C}} = \sum_{i=1}^{k} a_i \min_{v_i}$ and $\max_{\mathcal{C}} = \sum_{i=1}^{k} a_i \max_{v_i}$. In case of unbounded variables $\max_{\mathcal{C}}$ is set to $+\infty$ and $\min_{\mathcal{C}}$ is set to $-\infty$.

The state sets that are used are represented in form of files. The search frontier denoting the current BFS layer is tested for an intersection with the goal, and this intersection is further reduced according to the already established bound.

Algorithm 9.3 Procedure Cost-Optimal-External-BFS

Input: A metric planning problem $\mathcal{P}_\mathcal{C} = \langle \mathcal{V}, AP, \mathcal{O}, \mathcal{I}, \mathcal{T}, \mathcal{C} \rangle$. U represents the best solution cost found so far. $\mathcal{G}(\mathcal{P}_\mathcal{C})$ is the underlying transition graph generated on-the-fly.

Output: An optimal plan to the metric problem $\mathcal{P}_\mathcal{C}$, if one exists.

1: $U \leftarrow \max_\mathcal{C}$; $i \leftarrow 1$
2: $Open(-1) \leftarrow \emptyset$; $Open(0) \leftarrow \{\mathcal{I}\}$
3: **while** $(Open(i-1) \neq \emptyset)$ **do**
4: $Temp(i) \leftarrow Succ(Open(i-1))$
5: **if** $(Temp(i) \cap \mathcal{T} \cap \{s \mid \mathcal{C}(s) < U\} \neq \emptyset)$ **then** //HAS A GOAL BEEN FOUND?
6: $U \leftarrow \min\{j \in [\min_\mathcal{C} \ldots U] \mid Temp(i) \cap \mathcal{T} \cap \{s \mid \mathcal{C}(s) < j\} \neq \emptyset\}$
7: $ConstructSolution(Temp(i) \cap \mathcal{T} \cap \{s \mid \mathcal{C}(s) < U\})$
8: $Temp'(i) \leftarrow$ *sort-and-remove-duplicates*$(Temp(i))$
9: **for** $loc \leftarrow 1$ **to** $\mathcal{L}(\mathcal{G}(\mathcal{P}_\mathcal{C}))$ **do** //SUBTRACT $\mathcal{L}(\mathcal{G}(\mathcal{P}_\mathcal{C}))$ PREVIOUS LAYERS
10: **if** $i - loc < 0$ **then** break //SUBTRACT ONLY LEGAL LAYERS
11: $Temp'(i) \leftarrow Temp'(i) \setminus Open(i - loc)$
12: **end for**
13: $Open(i) \leftarrow Temp'(i)$
14: $i \leftarrow i + 1$
15: **end while**

Theorem 9.2 (I/O Complexity of Cost-Optimal External BFS) *Given a metric planning problem The I/O complexity of Cost-Optimal External BFS is*

$$O(sort(|\mathcal{R}|) + \mathcal{L}(\mathcal{G}(\mathcal{P}_\mathcal{C})) \cdot scan(|\mathcal{S}|)) \text{ I/Os.}$$

Proof The term $sort(|\mathcal{R}|)$ is due to external sorting the list of successors. For subtraction of previous layers, we need to scan $\mathcal{L}(\mathcal{G}(\mathcal{P}_\mathcal{C}))$ previous layers from the each newly generated layer accumulating to $O(\mathcal{L}(\mathcal{G}(\mathcal{P}_\mathcal{C})) \cdot scan(|\mathcal{S}|))$ I/Os, in the worst case. ■

9.5 Duplicate Detection in Action Planning

A crucial issue in external memory algorithms is the removal of duplicates. Since there is no hash table involved, alternative methods are required to remove the duplicates. As we have seen earlier, that the number of layers sufficient for full duplicate detection depends on a property of the search graph called *locality*.

The question then arises is: How can we compute the locality in an implicitly given graph as they appear in action planning? In the following, a partial answer to this question is provided, which is based only on the *operators*. For the ease of presentation, we restrict to actions with cost of one. A duplicate node in an implicit graph appears when a sequence of operators, applied to a state generate the same state again, i.e., they cancel the effects of each other. Hence, the following definition:

Definition 9.8 (no-op *Sequence) A sequence of operators* $a_1, a_2, \ldots, a_k$ *is a* no-op sequence*, if its application on a state produces no effects, i.e,* $a_k \circ \ldots \circ a_2 \circ a_1 =$ no-op*,*

This definition allows to bound the locality in the following proposition. It generalizes the observation that for undirected search spaces, in which for each operator a_1 we find an inverse action a_2 such that $a_2 \circ a_1 = \textit{no-op}$.

Proposition 2 (no-op *Sequence determines Locality) Let* $\mathcal{O}$ *be the set of operators in the search space and* $l = |\mathcal{O}|$*. If for all operators* a_1 *we can provide a sequence* $a_2, \ldots, a_k$ *with* $a_k \circ \ldots \circ a_2 \circ a_1 =$ no-op*, then the locality* $\mathcal{L}$ *of the implicitly generated graph* $\mathcal{G}(\mathcal{P})$ *is at most* k*.*

Proof If $a_k \circ \ldots \circ a_2 \circ a_1 = \textit{no-op}$, each state v reached by applying a_1 on u can eventually result in generating u again in at most $k-1$ steps. If such a no-op sequence is available for every action, for all $u, v \in \mathcal{S}$,

$$\max_{u,v \in \mathit{Succ}(u)} \{\delta(v,u)\} = k - 1. \tag{9.1}$$

Theorem 5.10 states that for all $u, v \in \mathcal{S}$,

$$\max_{u,v \in \mathit{Succ}(u)} \{\delta(v,u)\} + 1 \geq \max_{u,v \in \mathit{Succ}(u)} \{\delta(\mathcal{I},u) - \delta(\mathcal{I},v)\} + 1$$

or,

$$\max_{u,v \in \mathit{Succ}(u)} \{\delta(v,u)\} + 1 \geq \mathcal{L}(\mathcal{G}(\mathcal{P}))$$

Using Equation 9.1 in the above inequality, we get

$$\mathcal{L}(\mathcal{G}(\mathcal{P})) \leq k$$

which gives us the locality of the planning graph. ■

The condition $a_k \circ \ldots \circ a_2 \circ a_1 = \textit{no-op}$ can be tested in $O(l^k)$ time. It suffices to check that the cumulative add effects of the sequence is equal to the cumulative delete effects. Using the denotation by (Haslum & Jonsson 2000), the cumulative add C_A and delete C_D effects of a sequence can be defined inductively as, $C_A(a_k) = A_k$, $C_D(a_k) = D_k$, and

$$\begin{aligned} C_A(a_1, \ldots, a_k) &= (C_A(a_1, \ldots, a_{k-1}) - D_k) \cup A_k \\ C_D(a_1, \ldots, a_k) &= (C_D(a_1, \ldots, a_{k-1}) - A_k) \cup D_k \end{aligned}$$

This result gives us the missing link to the successful application of External BFS in planning. Subtracting k previous layer including the generating layer from the successor list in a BFS guarantees its termination on finite planning graphs.

9.6 Experiments

In this section, we present two sets of experiments. All experiments are run on a Pentium-4 with 600 GB of hard-disk space and 2GB RAM running Linux. In the first set, we present non-optimal planning results with External EHC. This set of experiments are performed on one of the most challenging domains called Settlers, used in the third and the fourth international planning competitions. The distinguishing feature of the domain is that most

of the domain semantics is encoded with numeric variables. The whole problem set for this domain has been solved by only one planner, SGPlan, which is based on goal-ordering and Lagrange optimization and finds plans very fast but with large plan lengths.

We have extended the state-of-the-art planning system Metric-FF for external exploration. The new planning system is called MIPS-XXL, where MIPS stands for *Model Checking Integrated Planning System*. Metric-FF uses helpful-action pruning to use the actions that are used in the relaxed-plan construction. This pruning destroys the completeness of the method but can be very effective in practice. Running External EHC without helpful-action pruning resulted in shorter plan lengths while consuming lesser internal memory. In external search, predecessor pointers are not available. To output a plan, we, hence, save the full predecessor information with every state. Plan is then constructed by back-tracking from the last layer to the first layer. Our plans are validated by VAL tool (Howey, Long, & Fox 2005). In Table 9.1, we compare the plan lengths as found by External EHC to the ones found by SGPlan. The first 5 problems can be solved by internal Enforced Hill Climbing too when using helpful-action pruning. All the experiments in this table are performed without using helpful-action pruning except for problem 3. We also report the internal memory and external space consumption by our algorithm. Note that the internal memory requirement can be scaled down to an arbitrary amount by using small internal buffers. The locality we encountered during this domain is 3 as observed by checking the duplicates in all previous layers for every BFS invoked.

P. #	SGPlan	MIPS-XXL	Memory(MB)	Disk Space(MB)
1	106	53	115	11
2	82	27	114	1
3	133	52	117	75
4	199	64	264	1,384
6	193	79	281	186
7	292	>132	526	>43,670

Table 9.1: Exploration Results on Settlers Domain.

The exploration for problem 7 was canceled in the middle because of time constraints. The total time taken was 3 days and 14 hours. It consumed 48.89 Gigabytes of external storage while internal process remained constant at 526 Megabytes. Starting from a heuristic estimate of 87, the algorithm climbed down to 8 with a total depth of 132. The bottleneck in this exploration is the computationally expensive calculations for the heuristic estimates. Figure 9.4 shows the histogram of states' distribution across different BFS layers for this problem. Here we see different groups of layers that actually correspond to starting a new external BFS when a state with a better heuristic value is found.

For the second set, we present optimal planning results on one of the hardest problems in the Traveling Purchase Problem (TPP) domain introduced for the fifth international planning competition. In order to avoid a conflict with the on-going competition (IPC-2006), we only considered preferences p0A, p1A, p2A and p3A, in the goal condition. Table 9.2 shows the results of our exploration. We report the number of nodes in each layer obtained after refinement with respect to the previous layers. The locality as observed by the refinement with respect to all previous layers was 2. An entry in the *Goal Cost* column corresponds to

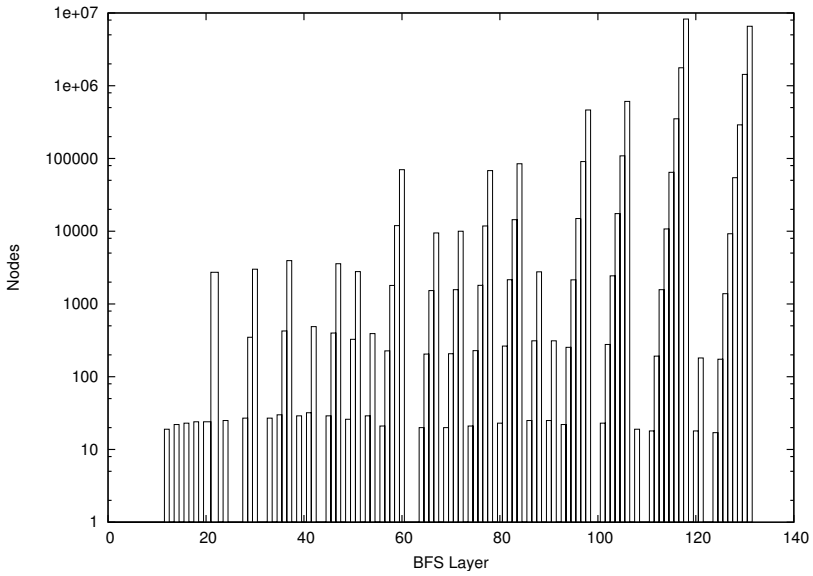

Figure 9.4: Histogram (logarithmic scale) on Number of Nodes in BFS Layers for External Enforced Hill Climbing on Problem-7 of Settlers.

the best goal cost found in that layer. The exploration had to be paused because of some technical problems in the machine making it unusable for two weeks.

Total time consumed in exploration and duplicates removal is around 30 days. The active layer being expanded was 26 with the best cost of 92 also found at this layer. The (**) in the 27-th row indicates that the reported node count and the space consumption for this layer does not reflect the final count. About 2×10^9 states were generated. Space consumption lies by 567 Gigabytes taken by $2,086,166,351$ nodes residing on the hard disk. For this particular exploration, the plan re-construction was switched off to safe hard disk space taken by predecessor states. Time is largely consumed by early duplicate detections. It turned out to be more space efficient to start an external sort and subtraction routine whenever 30 Million states were generated – instead of waiting till a whole layer is generated.

As is apparent from Table 9.2 that the branching factor has started to go down by a factor of 0.05 for each layer and a completion of this exploration is foreseen. The internal process size remained constant at 992 Megabytes. The exploration has been restarted several times using the pause-and-resume support implemented in the software.

9.7 Summary

Large graphs are often met in planning domains. Though inadmissible heuristics or some other technique to guide the search can result in faster search times, the plan lengths are often very large. We contribute External Enforced Hill Climbing Search that can utilize inadmissible heuristics to guide the search while finding a plan. For PDDL3 planning with preferences, we contributed cost-optimal External Breadth-First Search that can cope up with the monotonic decreasing cost functions that are out-of-reach of traditional branch-and-bound algorithms.

Level	Nodes	Size in Gigabytes	Best Goal Cost
0	1	0.000000272	105
1	2	0.00000054	-
2	10	0.00000272	-
3	61	0.0000166	-
4	252	0.000069	-
5	326	0.000089	104
6	3,153	0.00086	-
7	9,509	0.00259	-
8	26,209	0.0071	103
9	66,705	0.0181	-
10	158,311	0.0431	-
11	353,182	0.096	101
12	745,960	0.203	-
13	1,500,173	0.408	-
14	2,886,261	0.785	97
15	5,331,550	1.450	-
16	9,481,864	2.579	-
17	16,266,810	4.424	96
18	26,958,236	7.331	-
19	43,199,526	11.748	-
20	66,984,109	18.216	95
21	100,553,730	27.345	-
22	146,201,921	39.759	-
23	205,973,535	56.014	93
24	281,284,553	76.494	-
25	372,500,963	101.300	-
26	478,551,397	130.140	92
27**	327,128,042	88.961	-
SUM	2,086,166,351	567.325	92

Table 9.2: Exploration Results on Problem-5 of TPP Domain.

A crucial problem in external memory algorithms is the duplicate detection with respect to previous layers. Using the locality of the graph calculated directly from the operators themselves, we provide a bound on the number of previous layers that have to be looked at to avoid re-expansions. We report the largest exploration with BFS in planning domains – traversing a state space as large as about 500 Gigabytes in about 30 days. To the best of our knowledge, this is the longest running and the largest exploration reported in planning literature.

Zhou & Hansen (2006b) have also contributed an external planning approach that extends Structured Duplicate Detection (cf. Section 3.7) to domain-independent planning. As the application of SDD is very much independent on the availability of a graph partitioning function, the authors proposed to identify the group of *facts* that form an XOR relation. The idea of using XOR groups goes back to Edelkamp & Helmert (1999) where it was used in the context of symbolic planning. Experiments were performed only on pure propositional STRIPS domains with Breadth-First Heuristic Search along with SDD.

With the implementation of EM search in MIPS-XXL, we contribute the first planner that can deal with PDDL3 *and* supports the external search. The planner has participated in the International Planning Competition (IPC-5) and has won the *Distinguished Performance Award*. It was tuned to start the external search once the internal search fails to reach a

solution within the first 5 minutes of the 30 minutes limit. To reach a solution faster, the external search was also equipped with a beam search that picks only few best nodes to expand in the next level. The external search was able to find some plans within the specified times by keeping the process size constant at 900 MB (1 GB limit) while exploring state spaces as large as 8 GB on the hard disk.

Moreover, since states are kept on disk, external algorithms have a large potential for parallelization. We noticed that most of the execution time is consumed while calculating heuristic estimates. Using the similar techniques as developed in Chapter 7, the external planning algorithms presented here can be made compatible with the new multi-core architectures.

Chapter 10

Non-deterministic and Probabilistic Planning

The state spaces we have considered in the earlier chapters share *determinism* as a common property – the output of each action was fully determined. In this chapter, we consider a more general class of state spaces, namely *non-deterministic* and *probabilistic* state spaces. In these state spaces, the outcome of an action is not fully determined and is stochastic. An action, when applied to a given state, may succeed with a probability p, while it might not have any effect with a probability $1 - p$. Such state space models are called *Markov Decision Processes (MDP)* (Bellman 1957).

Applications of MDPs exist both in model checking and planning. In model checking, they provide the formalism to model stochastic systems that can be checked for properties described in probabilistic CTL or LTL (Hinton *et al.* 2006; Baier, Ciesinski, & Grer 2005). For example, consider a lossy channel between a sender and a receiver. Each message sent may or may not reach the receiver. For such a system one might want to know '*the probability that 5 out of 10 messages will be received*'.

MDPs are widely used in *planning under uncertainty*, or what is commonly referred to as *decision theoretic planning* (Boutilier, Dean, & Hanks 1999). A solution to a decision theoretic planning problem is a sequence of actions that achieves a certain goal with a *high probability* rather than just achieving the goal. A typical example of planning under uncertainty is the interaction of a robot with an environment. The robot interacts with real objects and, due to the non-accurate nature of its physical parts, might not always get the desired result.

MDPs are solved optimally through dynamic programming using either Value Iteration (VI) or Policy Iteration (many variations including heuristic search algorithms also exist). Both these algorithms involve iterative scans of the whole state space. For large state spaces that cannot fit into the available RAM, these methods are not usable. This chapter extends the Value Iteration algorithm, defined over a unified search model, to use external storage devices. The new algorithm is termed as *External Value Iteration* (Edelkamp, Jabbar, & Bonet 2007), and is applicable not only on MDPs, but also on deterministic problems, AND/OR graphs, and Game trees.

Structure of the chapter: First, the unified search model is presented. The Value Iteration algorithm to solve the problems defined in the new search model is presented next. Af-

terwards, we address the external implementation of Value Iteration suited for the unified search model. The algorithm is explained with a working example. Finally, large scale experimental results of the new algorithm on some known benchmark problems are provided.

10.1 The Unified Search Model

The general model for state space problems considered by Bonet & Geffner (2006) is able to accommodate diverse problems in AI including deterministic, AND/OR graphs, Game trees and MDPs. The model consists of:

U1. a discrete and finite state space $\mathcal{S}$,

U2. a non-empty subset of terminal states $\mathcal{T} \subseteq \mathcal{S}$,

U3. an initial state $\mathcal{I} \in \mathcal{S}$,

U4. subsets of applicable actions $A(u) \subseteq A$ for $u \in \mathcal{S} \setminus \mathcal{T}$,

U5. a transition function $\mathit{Succ}(a, u)$ for $u \in \mathcal{S} \setminus \mathcal{T}$, $a \in A(u)$,

U6. terminal costs $c_{\mathcal{T}} : \mathcal{T} \to \mathbb{R}$, and

U7. non-terminal costs $c : A \times \mathcal{S} \setminus \mathcal{T} \to \mathbb{R}$.

For deterministic models, the transition function *Succ* maps actions and non-terminal states into states. For AND/OR models, *Succ* maps actions and non-terminal states into *subsets of states*. Game trees are obtained from AND/OR graphs by using non-zero terminal costs and zero non-terminal costs. MDPs are non-deterministic models with probabilities $P_a(v|u)$ such that $P_a(v|u) > 0$ if $v \in \mathit{Succ}(a, u)$, and $\sum_{v \in \mathcal{S}} P_a(v|u) = 1$.

The solutions to these models can be expressed in terms of Bellman equations. For the deterministic case, we have

$$h(u) = \begin{cases} c_{\mathcal{T}}(u) & \text{if } u \in \mathcal{T}, \\ \min_{a \in A(u)} c(a, u) + h(\mathit{Succ}(a, u)) & \text{otherwise.} \end{cases}$$

For the non-deterministic, Additive and Max, cases

$$h_{add}(u) = \begin{cases} c_{\mathcal{T}}(u) & \text{if } u \in \mathcal{T}, \\ \min\limits_{a \in A(u)} c(a, u) + \sum\limits_{v \in \mathit{Succ}(a,u)} h(v) & \text{otherwise.} \end{cases}$$

$$h_{max}(u) = \begin{cases} c_{\mathcal{T}}(u) & \text{if } u \in \mathcal{T}, \\ \min\limits_{a \in A(u)} c(a, u) + \max\limits_{v \in \mathit{Succ}(a,u)} h(v) & \text{otherwise.} \end{cases}$$

And, for the MDP case, we have

$$h(u) = \begin{cases} c_{\mathcal{T}}(u) & \text{if } u \in \mathcal{T}, \\ \min\limits_{a \in A(u)} c(a, u) + \sum\limits_{v \in \mathcal{S}} P_a(v|u) h(v) & \text{otherwise.} \end{cases}$$

Solutions h^* to the Bellman equations are value functions of form $h : \mathcal{S} \to \mathbb{R}$. The value $h^*(u)$ expresses the minimum expected cost to reach a terminal state from state u. Policies,

on the other hand, are functions $\pi : \mathcal{S} \rightarrow A$ that map states into actions and generalize the notion of a plan for non-deterministic and probabilistic settings. In deterministic settings, a plan for state u consists of a sequence of actions to be applied at u that is guaranteed to reach a terminal state; it is optimal if its cost is minimum. Policies are greedy if they are best with respect to a given value functions; policies π^* greedy with respect to h^* are optimal.

In the following, the symbol $\mathcal{P}$ is used to denote the problem of finding a solution to one of the Bellman equations. The exact equation depends on the state space model being considered. The underlying graph generated during a state space exploration is denoted by $\mathcal{G}(\mathcal{P})$.

10.2 RAM based algorithms for Non-Deterministic and Probabilistic State Spaces

Because of the memory intensive nature of the Value Iteration algorithm (see next section), in the last decades, much effort has been directed towards using heuristic search in solving AND/OR graphs and MDPs. AO*, for example, extends A* over acyclic AND/OR graphs (Nilsson 1980). LAO* (Hansen & Zilberstein 2001) further extends AO* over AND/OR graphs with cycles and is well suited for Markov Decision Processes (MDPs). Real-Time Dynamic Programming (RTDP) extends the LRTA* search algorithm (Korf 1990) over non-deterministic and probabilistic search spaces (Barto, Bradtke, & Singh 1995). LAO* and RTDP aim at the same class of problems, the difference however is that RTDP relies on trial-based exploration of the search space – a concept adopted from reinforcement learning – to discover the relevant states of the problem and determine the order in which to perform value updates. LAO*, on the other hand, finds a solution by systematically expanding a search graph in a manner akin to A* and AO*. The IDAO* algorithm, developed in the context of optimal temporal planning, performs depth-first iterative-deepening to AND/OR graphs (Haslum 2006). All these algorithms have the interleaving of dynamic updates of cost estimates and the extension of the search frontier in common.

Learning DFS was introduced in (Bonet & Geffner 2005; Bonet & Geffner 2006) for a variety of models including deterministic models, Additive and Max AND/OR graphs, and MDPs. In the reported experiments, LDFS turned out to be superior to blind dynamic programming approaches like Value Iteration and heuristic search strategies like RTDP over MDPs. LDFS is designed on the unified search model described earlier in this chapter. It is able to deal with deterministic problems, AND/OR graphs under additive and max cost criteria, Game trees and MDPs. Interestingly, LDFS instantiates to state-of-the-art algorithms for some of these models, and to novel algorithms on others. It instantiates to IDA* over deterministic problems, while bounded-LDFS, LDFS with an explicit bound parameter, to the MTD($-\infty$) over Game trees (Plaat *et al.* 1996). On AND/OR models, it gives rise to novel algorithms (Bonet & Geffner 2005).

An orthogonal approach for large MDPs is to use approximation techniques to solve not the original problem, but an abstraction of it with the hope that the solution to the latter will be a good approximation to the input problem. There are different methods to do this. Perhaps the most known one is to use polynomial approximations or linear combinations of basis functions to represent the value function (Bellman, Kalaba, & Kotin 1963; Tsitsiklis & Roy 1996), or more complex ones such as those employing neural networks (Tesauro 1995; Bertsekas & Tsitsiklis 1996). Recently, there have been methods based on LP and constraint

Algorithm 10.1 Value Iteration

Input: State space model $\mathcal{P}$; initial heuristic (estimates) h; tolerance $\epsilon > 0$; maximum iterations $t_{\max}$.

Output: ϵ-Optimal value function if $t_{\max} = \infty$.

```
1: S ← Generate-State-Space(P)
2: for all u ∈ S do h_0(u) ← h(u)
3: t ← 0; Res ← +∞
4: while t < t_max ∧ Res > ε do
5:    Res ← 0
6:    for all u ∈ S do
7:       Apply update rule for h_{t+1}(u) based on the model
8:       Res ← max{|h_{t+1}(u) − h_t(u)|, Res}
9:    end for
10:   t ← t + 1
11: end while
12: return h_{t−1}
```

sampling in order to solve the resulting approximations (Farias & Roy 2004; Guestrin, Koller, & Parr 2001).

Unfortunately, for large search spaces that cannot fit into the RAM even in compressed form, all of the algorithms are doomed to failure.

10.3 Value Iteration

Value Iteration (Algorithm 10.1) is an unguided algorithm for solving the Bellman equations and hence obtaining optimal solutions for state space models defined by the conditions U1–U7.

The algorithm proceeds in two phases. In the first phase, the whole state space is generated from the initial state $\mathcal{I}$. In this process, an entry in a hash table (or vector) is allocated in order to store the h-value for each state u; this value is initialized to $c_{\mathcal{T}}(u)$ if $u \in \mathcal{T}$, or to a given heuristic estimate (or zero if no estimate is available) if u is non-terminal.

In the second phase, iterative scans of the state space are performed updating the values of non-terminal states u as:

$$h_{\text{new}}(u) \;:=\; \min_{a \in A(u)} Q(a,u) \tag{10.1}$$

where $Q(a,u)$, which depends on the model, is defined as

$$Q(a,u) \;=\; c(a,u) + h(\mathit{Succ}(a,u))$$

for deterministic models,

$$Q(a,u) \;=\; c(a,u) + \sum_{v \in \mathit{Succ}(a,u)} h(v)\,,$$

$$Q(a,u) \;=\; c(a,u) + \max_{v \in \mathit{Succ}(a,u)} h(v)$$

for non-deterministic Additive and Max models, and

$$Q(a,u) \;=\; c(a,u) + \sum_{v \in \mathcal{S}} P_a(v|u)h(v)$$

for MDPs.

Value Iteration converges to the solution h^* provided that $h^*(u) < \infty$ for all u. In the case of MDPs, which may have cyclic solutions, the number of iterations is not bounded and Value Iteration typically only converges in the limit (Bertsekas 1995). For this reason, for MDPs, Value Iteration is often terminated after a predefined bound of $t_{\max}$ iterations have been performed, or when the residual falls below a given $\epsilon > 0$. The residual is defined as

$$\max_{u \in \mathcal{S}} |h_{\text{new}}(u) - h(u)|.$$

Value Iteration is formulated as a pseduo-code in Alg. 10.1.

10.4 External Value Iteration

We now discuss our approach for extending the Value Iteration procedure to work on large state spaces that cannot fit into the RAM. We call the new algorithm *External Value Iteration*. The first and foremost difficulty one faces in designing an external memory variant of Value Iteration is the need to connect states with their predecessors. Note that in all previous algorithms in this dissertation, the search only had to proceed forward. But in Value Iteration, we need to back-propagate the information from successor states. Hence, instead of working on states, we chose to work on edges. The rationale shall become clearer soon. In our case, an edge is a 4-tuple

$$(u, v, a, h(v))$$

where u is called the predecessor state, v the stored state, a the operator that transforms u into v, and $h(v)$ is the current value for v. Clearly, v must belong to $\mathit{Succ}(a,u)$. In deterministic models, v is determined by u and a and so it can be completely dropped, but for the non-deterministic models, it is a necessity.

Similarly to the internal version of Value Iteration, the external version works in two phases. A forward phase, where the state space is generated, and a backward phase, where the heuristic values are repeatedly updated until an ϵ-optimal policy is computed, or $t_{\max}$ iterations are performed.

We will explain the algorithm using the graph in Fig. 10.1. The states are numbered from 1 to 10, the initial state is 1, while 8 and 10 are the terminal states. The numbers next to the states are the initial heuristic values.

10.4.1 Forward Phase: State Space Generation

Typically, a state space is generated by a depth-first or a breadth-first exploration that uses a hash table to avoid re-expansion of states. We choose an external breadth-first exploration to handle large state spaces. Since in an external setting a hash table is not affordable, we rely on *delayed duplication detection* (DDD). It consists of two phases, first removing duplicates within the newly generated layer, and then removing duplicates with respect to previously

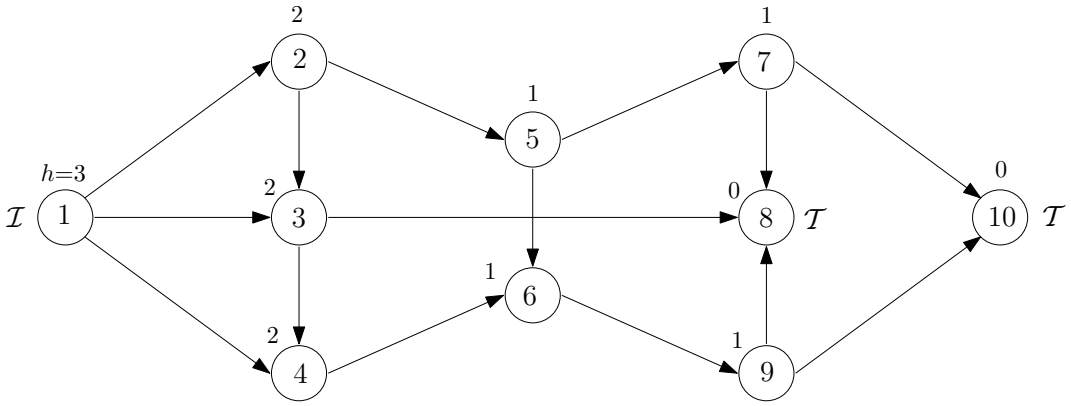

Figure 10.1: An example graph with initial h-values.

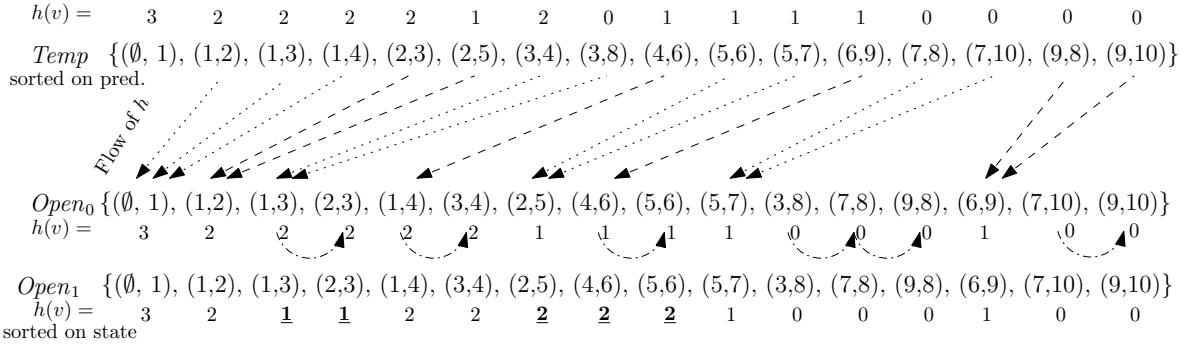

Figure 10.2: Backward phase. the files $open_0$ and *temp* are stored on disk. A parallel scan of both files is done from left to right. The file $open_1$ is the result of the first update. Values that changed in the first update are shown with bold underline typeface.

Algorithm 10.2 External Value Iteration

Input: State space model $\mathcal{P}$; initial value function h; tolerance $\epsilon > 0$; maximum iterations $t_{\max}$.
Output: ϵ-Optimal value function (stored on disk) if $t_{\max} = \infty$.

1: $Open(0) \leftarrow \{(\emptyset, \mathcal{I}, —, h(\mathcal{I}))$
2: $d \leftarrow 0$ //BFS LAYERS ARE INDEXED BY d
3: **while** $(Open(d) \neq \emptyset)$ **do** //ITERATE ON ALL NON-EMPTY BFS LAYERS
4: $d \leftarrow d + 1$
5: $Open(d) \leftarrow \{(u, v, a, h(v)) : u \in Open(d-1), a \in A(u), v \in Succ(a, u)\}$ //COLLECT SUCCESSORS
6: **Externally sort** $Open(d)$ with respect to edges (u, v)
7: Remove duplicate edges in $Open(d)$
8: **for** $loc \in \{1, \ldots, \mathcal{L}(\mathcal{G})\}$ **do** //SUBTRACT $\mathcal{L}(\mathcal{G}(\mathcal{P}))$ PREVIOUS LAYERS FROM $Open(d)$
9: $Open(d) \leftarrow Open(d) \setminus Open(d - loc)$
10: **end for**
11: **end while**
12: $Open_0 \leftarrow Open(0) \cup Open(1) \cup \ldots \cup Open(d-1)$ //MERGE ALL BFS LAYERS INTO ONE
13: **Sort** $Open_0$ with respect to states v
14: $t \leftarrow 0$ //t MANAGES THE ITERATION NUMBER
15: $Res \leftarrow +\infty$ //INITIALLY, RESIDUAL IS INFINITY
16: **while** $t < t_{\max} \wedge Res > \epsilon$ **do** //PERFORM BACKWARD UPDATES UNTIL AN ACCEPTABLE CHANGE IN THE RESIDUAL IS REACHED
17: $Res \leftarrow$ *External-VI-Backward-Update*$(\mathcal{P}, Open_t)$
18: $t \leftarrow t + 1$
19: **end while**

generated layers. For undirected graphs, looking at two previous layers is enough to remove all duplicates (Munagala & Ranade 1999), but for directed graphs the *locality* $\mathcal{L}(\mathcal{G}(\mathcal{P}))$ of the problem graph $\mathcal{G}(\mathcal{P})$ dictates the number of layers to be looked at (see Chapter 5). In the example graph of Figure 10.1, $\mathcal{L}(\mathcal{G}(\mathcal{P})) = 2$ (8 is generated in the third layer through state 3 and again through 7 and 9 in the fifth layer). Note that as we deal with edges, we might need to keep several copies of a state. In our case, an edge $(u, v, a, h(v))$ is a duplicate, if and only if, its predecessor u, its state v, and the action a match an existing edge. Thus, in undirected graphs, there are two different edges for each undirected edge.

The procedure for DDD can be borrowed either from sorting-based DDD (Munagala & Ranade 1999), hash-based DDD (Korf & Schultze 2005), or structured duplicate detection (Zhou & Hansen 2004b). In our case, sorting-based DDD is the best choice for two reasons. Firstly, it is the most general form of DDD, which makes no further assumptions such as the maximum size of a layer. Secondly, the sorting order is further exploited during the backward phase.

Algorithm 10.2 shows External Value Iteration. The algorithm maintains layers $Open(d)$ on disk in the form of files. The first phase ends up by concatenating all layers into one *Open* list that contains all edges reachable from $\mathcal{I}$. The complexity of this phase is $O(\mathcal{L}(\mathcal{G}(\mathcal{P})) \cdot scan(|\mathcal{R}|) + sort(|\mathcal{R}|))$ I/Os. The first term is obtained by summing up the total I/Os required for successor generation and subtracting $\mathcal{L}(\mathcal{G}(\mathcal{P}))$ previous layers. The second term $sort(|\mathcal{R}|)$ is the accumulative I/O complexity for delayed duplicate detection on all the layers.

Algorithm 10.3 External Value Iteration – Backward Update

Input: State space model $\mathcal{P}$; state space $Open_t$ stored on disk.
Output: $Open_{t+1}$ generated by this update, stored on disk.

1: $Res \leftarrow 0$
2: **Copy** $Open_t$ into *Temp*
3: **Sort** *Temp* with respect to states u //VALUES ARE PROPAGATED FROM *Temp* TO $Open_t$
4: **for all** $(u, v, a, h) \in Open_t$ **do**
5: **if** $v \in \mathcal{T}$ **then** //CASE I: A TERMINAL STATE IS REACHED
6: **Write** (u, v, a, h) to $Open_{t+1}$ //NO UPDATE REQUIRED FOR A TERMINAL STATE
7: **else if** $v = v_{last}$ **then** //CASE II: ONE COPY OF THE STATE HAS JUST BEEN UPDATED
8: **Write** (u, v, a, h_{last}) to $Open_{t+1}$ //USE THE PREVIOUSLY COMPUTED h-VALUE FOR THIS NEW COPY
9: **else**
10: **Read** (x, y, a', h') from *Temp* //READ A NEW SUCCESSOR
11: *Post-condition:* $x = v$ //CASE III IS $x \neq v$
12: *Post-condition:* both files are aligned //CASE IV
13: **for all** $a \in A(v)$ **do** $Q(a) \leftarrow c(a, v)$ //INITIALIZE Q-VALUES FOR APPLICABLE ACTIONS
14: **while** $x = v$ **do** //ITERATE ON ALL THE SUCCESSORS OF v
15: $Q(a') \leftarrow Q(a') + P_{a'}(y|x)\, h'$ //COLLECT ALL THE Q-VALUES FROM THE SUCCESSORS
16: **Read** (x, y, a', h') from *Temp* //GET NEW SUCCESSOR
17: **end while**
18: **Push-back** (x, y, a', h') in *Temp* //WRONG SUCCESSOR READ, PUSH IT BACK TO THE FRONT OF THE STREAM
19: $h_{last} \leftarrow \min_{a \in A(v)} Q(a)$ //NEW h-VALUE IS THE MINIMUM OF ALL THE Q-VALUES
20: $v_{last} \leftarrow v$ //SAVE v TO CHECK AGAINST THE NEXT STATE FROM *Open*
21: **Write** (u, v, a, h_{last}) to $Open_{t+1}$ //STATE v HAS A NEW h-VALUE
22: $Res \leftarrow \max\{|h_{last} - h|, Res\}$ //COMPUTE THE CHANGE IN h-VALUE FOR THE RESIDUAL
23: **end for**
24: **return** *Res*

10.4.2 Backward Phase: Update of Values

This is the most critical part of the approach and deserves special attention. To perform the update (Equation (10.1)) on the value of state v, we have to use the values of its successor states. As they all are contained in one file, and since *there is no arrangement that can bring all successor states close to their predecessor states*, we make a copy of the entire graph (file) and deal with the current state and its successors differently. To establish the adjacencies, the second copy, called *Temp*, is sorted with respect to the node u. Remember that *Open* is sorted with respect to the node v.

A parallel scan of files *Open* and *Temp* gives us access to all the successors and values needed to perform the update on the value of v. This scenario is shown in Fig. 10.2 for the graph in the example. The contents of *Temp* and $Open_t$, for $t = 0$, are shown along with the heuristic values computed so far for each edge (u, v). The arrows show the flow of information (alternation between dotted and dashed arrows is just for clarity). The results of the updates are written to the file $Open_{t+1}$ containing the new values for each state after $t + 1$ iterations. Once $Open_{t+1}$ is computed, the file $Open_t$ can be removed as it is no longer

needed.

Algorithm 10.3 shows the backward update algorithm for the case of MDP models; the other models are similar. It first copies the *Open*$_t$ list in *Temp* using buffered I/O operations, and sorts the new *Temp* list according to the predecessor states u. The algorithm then iterates on all edges from *Open*$_t$ and searches for the successors in *Temp*. Since *Open*$_t$ is sorted with respect to states v, *the algorithm never goes back and forth in any of the Open*$_t$ *or Temp files*. Note that all reads and writes are buffered and thus can be carried out very efficiently by always doing I/O operations in blocks.

We now discuss the different cases that might arise when an edge $(u, v, a, h(v))$ is read from *Open*$_t$. States from Fig. 10.1 that comply with each case are referred in parentheses, while the lines in the algorithm are referred in brackets. The flow of h values for the example is shown in Fig. 10.2.

- *Case I:* v is terminal (states 8 & 10). Since no update is necessary, the edge can be written to *Open*$_{t+1}$ [Line 5].
- *Case II:* v is the same as the last updated state (state 3). Write the edge to *Open*$_{t+1}$ with $h(v)$ same as the one computed for the previous copy of v, h_{last} [Line 7]. (Case shown in Fig. 10.2 with curved arrows.)
- *Case III:* v has no successors. That means that v is a terminal state and so is handled by case I [Line 11].
- *Case IV:* v has one or more successors (remaining states). For each action $a \in A(v)$, compute the value $Q(a, v)$ by summing the products of the probabilities and the stored values. The values are stored in the array $Q(a)$ [Line 12].

For edges (x, y, a', h') read from *Temp*, we have:

- *Case A:* y is the initial state, implying $x = \emptyset$. Skip this edge since there is nothing to do. By taking $\emptyset$ as the smallest element, the sorting of *Temp* brings all such edges to the front of the file. (Case not shown for the sake of brevity.)
- *Case B:* $x = v$, i.e. the predecessor of this edge matches the current state from *Open*$_t$. This calls for an update in the $Q(a)$ values [Lines 13–17].

The array $Q : A \to \mathbb{R}$ is initialized to the edge weight $c(a, v)$, for each $a \in A(v)$. Once all the successors are processed, the new value for v is the minimum of the values stored in the Q-array for all applicable actions.

An important point to note here is that the last edge read from *Temp* in Line 16 is not used. The operation **Push-back** in Line 18 puts this edge back to the front of *Temp*. This operation incurs no physical I/O since the *Temp* file is buffered. Finally, to handle case II, a copy of the last updated node and its value are stored in variables v_{last} and h_{last} respectively.

Theorem 10.1 *The algorithm* External Value Iteration *performs at most* $O(\mathcal{L}(\mathcal{G}(\mathcal{P})) \cdot \mathit{scan}(|\mathcal{R}|) + t_{\max} \cdot \mathit{sort}(|\mathcal{R}|))$ *I/Os.*

Proof The forward phase requires $\mathcal{L}(\mathcal{G}(\mathcal{P})) \cdot \mathit{scan}(|\mathcal{R}|) + \mathit{sort}(|\mathcal{R}|)$ I/Os. The backward phase performs at most $t_{\max}$ iterations. Each such iteration consists of one sorting and two scanning operations for a total of $O(t_{\max} \cdot \mathit{sort}(|\mathcal{R}|))$ I/Os. ■

10.5 Experiments

We implemented External Value Iteration (Ext-VI) and compared it with Value Iteration (VI), and in some cases to the LDFS and LRTDP algorithms, on several problem instances from 4 different domains.

Algorithm	$\|\mathcal{S}\|/\|\mathcal{R}\|$	RAM	Time	$\|\mathcal{S}\|/\|\mathcal{R}\|$	RAM	Time
	barto-small			barto-big		
VI	9,398	**7.0M**	1.2s	22,543	16M	4.9s
Ext-VI	139,857	7.6M	19.4s	337,429	16M	82.5s
	hansen-bigger			hansen-biggest		
VI	51,952	37M	22.1s	150,910	77M	256.8s
Ext-VI	780,533	**34M**	325.2s	2,314,967	**67M**	1,365.3s
	ring-4			ring-5		
VI	33,238	34M	6.5s	94,395	86M	28.9s
Ext-VI	497,135	**33M**	92.2s	1,435,048	**80M**	193.1s
	square-5			square-6		
VI	1,328,820	218M	2,899s	out-of-memory		
Ext-VI	21,383,804	**160M**	3,975s	62,072,828	**230M**	19,378s

Table 10.1: Performance of External Value Iteration on the racetrack domain with parameters $p = 0.7$ and $\epsilon = 0.0001$.

The first domain is the racetrack benchmark used in a number of works. An instance in this domain is characterized by a racetrack divided into cells such that the task is to find the control for driving a car from a set of initial states into a set of goal states minimizing the number of time steps. Each applied control achieves its intended effect with probability 0.7 and no effect with probability 0.3. We report results on two instances from (Barto, Bradtke, & Singh 1995) (*barto-small* and *barto-big*), on an instance from (Hansen & Zilberstein 2001) (*hansen-bigger*, and on one new instance made by enlarging the map of the previous instance (*hansen-biggest* in the Table).

We also extended the ring and square instances from (Bonet & Geffner 2006). The results, obtained on an Opteron 2.4 GHz Linux machine with a memory bound of 512MB, are depicted in Table 10.1. The table shows the size of the problem in states for VI and edges for Ext-VI, the amount of RAM used (in MB), and the total time spent by the algorithm in seconds.

Algorithm	RAM	Time	Result
VI	$>$ 2 GB	$>$ 30 minutes	out-of-memory
LRTDP	$>$ 2 GB	$>$ 12 hours	out-of-memory
LDFS	$>$ 1.5 GB	$>$ 118 hours	out-of-time
External-VI	1.6 **GB (**16 **GB Disk)**	91 **hours**	**Converged in** 56 **iterations**

Table 10.2: Evaluation on Square-7 race-track problem. Total edges generated: $|\mathcal{R}|$ = 518,843,406; $\epsilon = 10^{-4}$

For *square-6*, the state space could not be generated by VI within the memory bound. For

Ext-VI, the edges took about 2GB of hard disk and another 6GB for the backward phase. We also started a much larger instance of the square model on an Athlon X2 with 2GB RAM. The instance consists of a grid of size 150×300 with the three start states at the top left corner and three goal states at bottom right corner. We will use the name *Square-7* for this instance in the subsequent discussion. Internal memory VI consumed the whole RAM and could not finalize. Ext-VI generated the whole state space with 518,843,406 edges consuming 16GB on the disk while just 1.6GB on RAM. After 91 hours, the algorithm finished in 56 iterations with a value of $h^*(s_0) = 29.233$ and residual $< 10^{-4}$. We also evaluated the algorithms LDFS and LRTDP on the same instance. LRTDP consumed 2GB while running for 12 hours and was canceled. LDFS consumed 1.5GB while running for 118 hours and was also canceled. The results are summarized in Table 10.2

The difference in RAM for Ext-VI observed in the table is due to the internal memory consumption for loading the instances. Moreover, external sorting of large files require opening of multiple file pointers each allocating a small internal memory buffer.

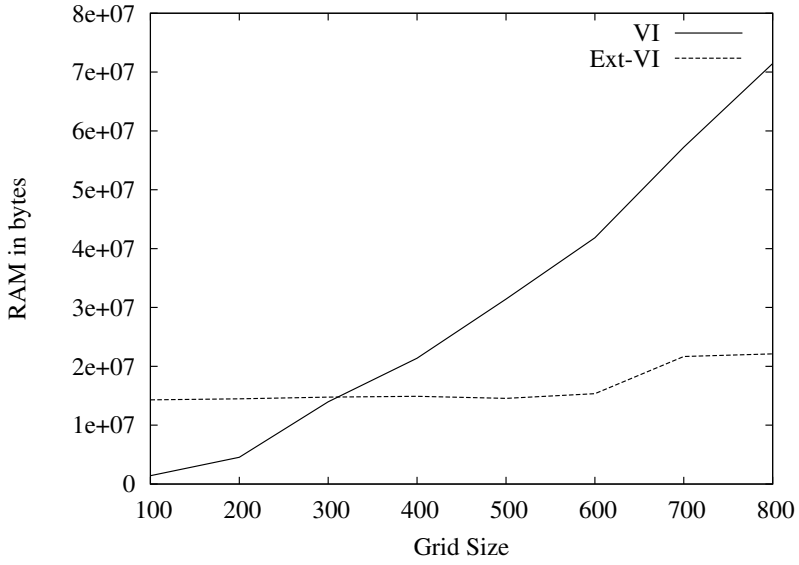

Figure 10.3: Memory consumption in the domain wet-floor

In the second set of experiments, we used the wet-floor domain from (Bonet & Geffner 2006) which consists of a navigation grid in which cells are wet, and thus slippery, with probability $p = 0.4$. The memory consumption in bytes is shown in Fig. 10.3 for grids ranging from 100×100 to 800×800. As demonstrated, the memory consumption of VI grows without control, whereas for Ext-VI the memory consumption can be adjusted to fit the available RAM. Indeed, we also tried a large problem of 10,000×10,000 on an Athlon X2 machine with 2GB RAM. The internal version of VI quickly went out of memory since the state space contains 100M nodes. Ext-VI, on the other hand, was able to generate the whole state space with diameter 19,586 and 879,930,415 edges taking 16GB of space to store the BFS layers and about 45GB of space for the backward updates. Each backward-update iteration takes about half an hour, and thus we have calculated that Ext-VI will take about 2 years to finish; we stopped the algorithm after 14 iterations. Fig. 10.4 (left) shows the increase in size of the optimal policy for the wet-floor domain in instances from 100×100 to 2000×2000. As can

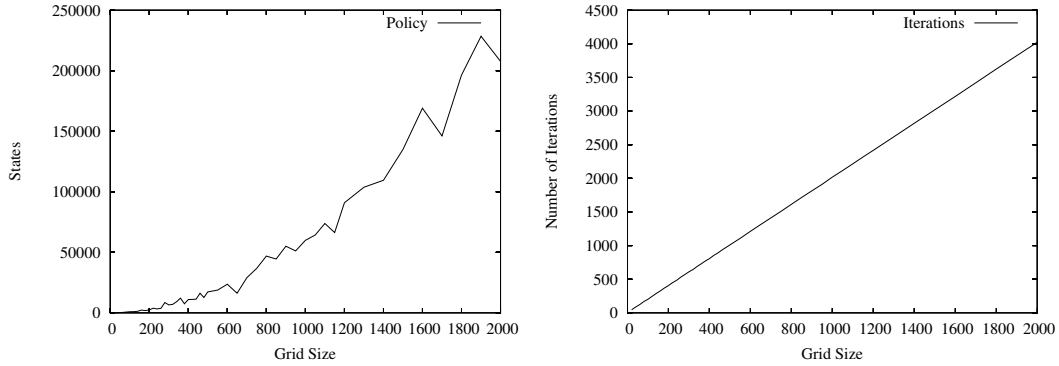

Figure 10.4: Growth of policy size in wet-floor domain (left). Growth of iterations in wet-floor domain by Internal VI (right).

be seen, the policy size grows quadratically and since any internal memory algorithm must store at least all those states that are part of the optimal policy, one can predict that no RAM based algorithm will be able to solve the 10,000×10,000 instance. Fig. 10.4 (right) shows the number of iterations needed by internal Value Iteration on the wet-floor domain over the same instances.

Algorithm	p	$\lvert\mathcal{S}\rvert/\lvert\mathcal{R}\rvert$	Iter.	Updates	$h(\mathcal{I})$	$h^*(\mathcal{I})$	RAM	Time
VI($h=0$)	1.0	181,440	27	4,898,880	0	14.00	21M	6.3
Ext-VI($h=0$)	1.0	483,839	32	5,806,048	0	14.00	**11M**	71.5
VI(h_{manh})	1.0	181,440	20	3,628,800	10	14.00	21M	4.4
Ext-VI(h_{manh})	1.0	483,839	28	5,080,292	10	14.00	**11M**	65.2
VI($h=0$)	0.9	181,440	37	6,713,280	0	15.55	21M	8.7
Ext-VI($h=0$)	0.9	967,677	45	8,164,755	0	15.55	**12M**	247.4
VI(h_{manh})	0.9	181,440	35	6,350,400	10	15.55	21M	8.3
Ext-VI(h_{manh})	0.9	967,677	43	7,801,877	10	15.55	**12M**	237.4

Table 10.3: Performance of External Value Iteration on deterministic and probabilistic variants of 8-puzzle.

The third domain considered is the $n \times m$ sliding tile puzzle. We performed two experiments: one with deterministic moves, and the other with noisy operators that achieve their intended effects with probability $p = 0.9$ and no effect with probability $1 - p$. Table 10.3 shows the results for random instances of the 8-puzzle for both experiments. Note the differences in the total iterations performed by the two algorithms. This is due to the fact that during an iteration, VI can also use the new values of the successors updated in the same iteration. On the other hand, Ext-VI does not have constant time accesses to these new values, as they might have already been flushed to the disk.

We also tried our algorithm on the 3×4-puzzle with $p = 0.9$. The problem cannot be solved with our internal VI because the state space does not fit in RAM, there are $12!/2 \approx 239 \times 10^6$ states. Ext-VI generated a total of 1,357,171,197 edges taking 45GB of disk space. An additional 90GB were used to keep the temporary files generated during updates. Fig-

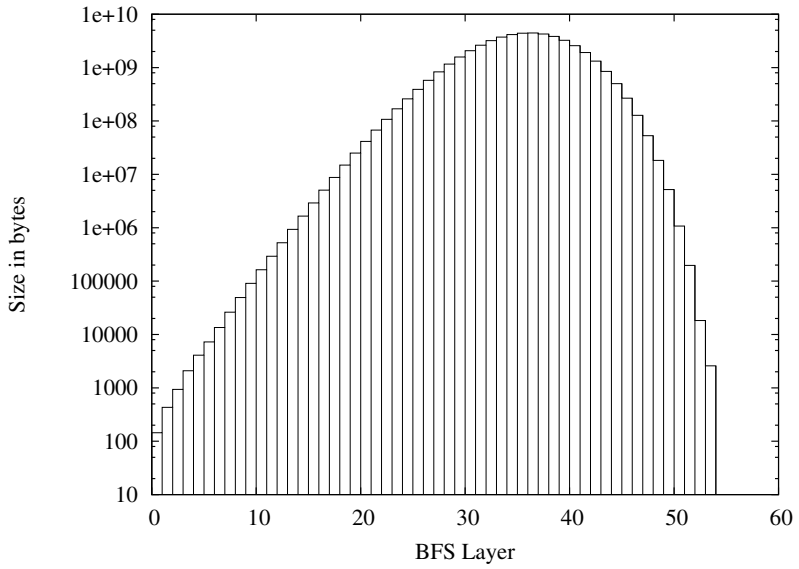

Figure 10.5: Edge space of 11-puzzle with $p = 0.9$ as found by External-VI

$\|\mathcal{R}\|$	p	ϵ	RAM	Disk	Iter.	$h^*(\mathcal{I})$	Time
$1,357,171,197$	0.9	10^{-4}	1.4GB	45GB	72	28.8889	437 hours

Table 10.4: Performance summary of Ext-VI on 3×4-puzzle.

ure 10.5 shows the memory consumption for the edges in each layer of the BFS exploration. The backward update took 437 hours and *finished in 72 iterations* until the residual became less than $\epsilon = 10^{-4}$. The internal memory consumed is 1.4GB on a Pentium-4 3.2GHz machine. The h-value of the initial state converged to 28.8889. The results are summarized in Table 10.4.

Finally, we implemented Ext-VI for deterministic settings within the planner MIPS-XXL (Edelkamp, Jabbar, & Nazih 2006), which is based on the state-of-the-art planner *MetricFF* (Hoffmann 2003b). As an example, we chose the propositional domain TPP from the recent planning competition (IPC-5). For instance 6 of TPP, 3,706,936 edges were generated. In 30 iterations the change in the average h-value was less than 0.001. It took 5 hours on a 3.2GHz machine taking a total of 3.2GB on the disk, while 167MB on RAM. The convergence behavior of average h-value is illustrated in Figure 10.6.

10.6 Summary

Wingate & Seppi (2004) proposed to integrate a disk-based cache mechanism into Value Iteration and Prioritized Value Iteration. The paper provides empirical evaluation on the number of cache hits and misses for both the algorithms and compares the effects of different cache sizes. However, External Value Iteration provides the first implementation of a well-established AI algorithm, able to solve different state space models, that exploits secondary storage in an *I/O efficient* manner. Contrary to internal Value Iteration, the external

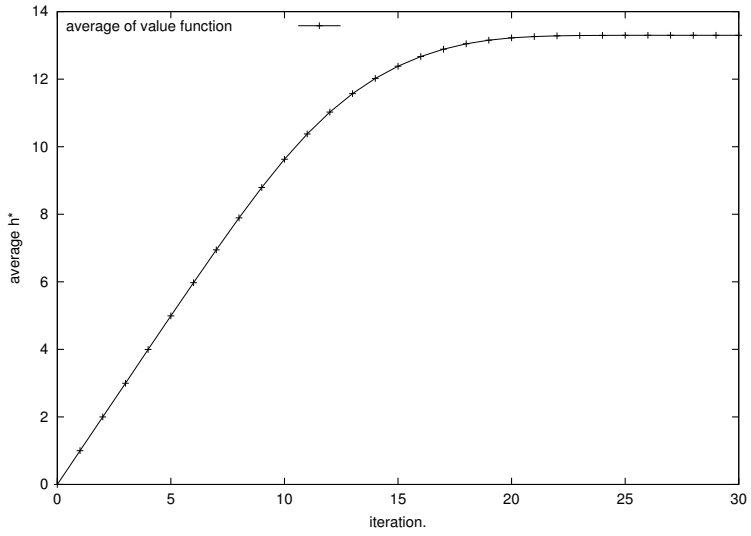

Figure 10.6: Convergence of External-VI on a deterministic planning problem from 5^{th} International Planning Competition. Implemented in the planner MIPS-XXL.

algorithm works on edges rather than on states. We provided an I/O complexity analysis that is bounded by the number of iterations times the sorting complexity. This is in itself a non-trivial result, as backward induction has to connect the predecessor states' values with the current state's value by connecting two differently sorted files in order to apply the Bellman update. We provided empirical results on known benchmark problems, showing that the disk space can overcome limitations in main memory. The largest state space we have been able to *generate and converge* in the probabilistic setting took 45.5GB.

The approximation approaches for solving MDPs, as we have seen in Section 10.2 do not compute exact solutions and thus are not directly comparable with our approach. Furthermore, some of them transform the problem into a simpler one which is solved with the Value Iteration algorithm, and thus amenable to be treated with the techniques in this chapter.

The obtained I/O complexity bound is remarkable: we cannot expect a constant number of iterations, since, in contrast to the deterministic case, there is currently no (internal) algorithm known to solve non-deterministic and MDP problems in a linear number of node visits.

Having a working externalization, it is worthwhile to try parallelizing the approach, e.g., on a cluster of workstations or even on a multi-core machine. Many techniques that are developed in Chapter 7 are directly applicable on this setting. It would also be interesting to extend the approach to heuristic search planning in non-deterministic and probabilistic models.

Part III

Conclusions

Chapter 11

Conclusions and Future Work

State space explosion problem lies at the heart of model checking and graph-based action planning. The limited amount of RAM hinders the application of these methods to real-world sized problems. In recent years, significant progress in magnetic media has made hard disks with large capacities widely available and very cheap. Unfortunately, when it comes to data access, RAM is a clear winner due to its fast random access ability. For each data access, hard disks have to introduce a latency time for moving the head to the desired data location. But once the head is there, not only the data that are directly under the head, but the whole circular track, can be read very fast.

External Memory (EM) algorithms are designed exactly with this phenomenon in mind. They require that the algorithm should be able to process data in blocks – each of which can be read or written with just one I/O operation involving a single head movement. The efficiency of EM algorithms is measured in the asymptotic number of I/Os they perform until they terminate.

The aim of this dissertation was to propose algorithms that are able to explore a variety of state spaces that cannot fit into the RAM. State spaces correspond to graphs that are not given beforehand, but are generated on-the-fly through a repetitive application of a set of transformation rules on a set of initial states. For a coherent treatment of these different state spaces, we introduced a formal characterization that allows one to instantiate state spaces that appear in single-agent games, LTL model checking, real-time model checking, action planning, Markov decision processes, AND/OR graphs, and Game trees. The set of proposed algorithms are analyzed for their asymptotic worst-case I/O complexities. The behaviors of these algorithms have been empirically evaluated on a large set of benchmark problems.

11.1 Assessment of Contributions

Let us briefly enumerate the results contributed in this thesis:

Large-scale Heuristic Search with External A*
A*, the famous heuristic search algorithm, requires a priority queue data structure to be able to pick the best state for expansion – for large state spaces an external priority queue is hence

required. External A*, the EM variant of A*, instead uses the properties of heuristic estimates and the *total* nature of the heuristic function to divide the whole exploration into sets. A total ordering on these sets induces an implicit priority queue. It serves two main purposes: each set can grow arbitrarily beyond the available RAM, and no upper bound on the total number of sets is required. The algorithm has been analyzed for its worst-case I/O complexity, as well as for the maximum number of state sets that would be expanded before the solution is reached. External A* has been extensively evaluated on Korf's famous 100 instances of 15-puzzle. The most difficult of them (instance 88) took 17 GB of hard disk while using a mere 1.2 GB of RAM.

External A* with Data Pipelining
Pipelining targets at reducing the number of I/O operations by feeding the input of one stage directly into the other stage, without creating an intermediate file. We have integrated the concept of pipelining into External A*. Compared to the original External A*, pipelining saves almost *half* of the I/O operations.

External Memory Directed Model Checking for Safety Properties
There are two main advantages of directed model checking: shorter error trails and space savings. With External A* extended for directed and weighted graphs as they appear in LTL model checking, we report on the first integration of EM algorithms with directed model checking. Empirical evaluation is done by implementing the new approach in our experimental model checker IO-HSF-SPIN. Explorations as large as 3 TB while using only 3.2 GB of RAM are documented.

Upper Bound on Duplicate Detection Scope
Duplicate detection with respect to previous layers is an essential part of EM search algorithms. We have contributed a novel procedure to identify the duplicate detection scope through a structural analysis of the Extended Finite State Machines (provided beforehand), and by dynamically analyzing their unfolding (generated on-the-fly). The bounds are provided for the general case, where the sub-systems communicate with each other through a set of variables from finite domains. The new bounds are tighter than the ones presented earlier by Jabbar & Edelkamp (2005). The main advantage of these bounds is that, through a static analysis at the beginning of the algorithm, it can be decided whether previous layers should be looked at or not for duplicates' removal. This optimization would save a large number of I/O scanning operations required for subtraction.

I/O Complexity Analysis of Earlier Disk-based Algorithms for Model Checking
None of the earlier disk-based algorithms for model checking were analyzed for their asymptotic I/O complexities. In this dissertation, we have also reported the I/O complexities of those algorithms, on the EM model by Aggarwal & Vitter (1988).

External Memory Directed and Breadth-First Model Checking for LTL Liveness
Previous research in disk-based algorithms was limited only to safety checking, due to the inherent difficulty in checking for an accepting cycle in a large graph. Our contribution is the first ever in this field. We have put forward an EM algorithm based on BFS traversal and

an External A* extension for liveness checking. The algorithms have also been implemented and evaluated on our experimental model checker IO-HSF-SPIN.

Heuristics For External Directed Liveness Checking
A set of heuristic functions to guide external liveness checking is proposed. These functions are purely based on the structural characteristics of the EFSMs, which are available beforehand. We furthermore proved the admissibility and consistency of these heuristics.

Distributed External A*
With the advent of multi-core systems, there is an indisputable need for parallel algorithms. Probing of our external model checker revealed some interesting bottlenecks. We found out that, in most cases, almost 80% of the total time was spent on internal work like successor generation and heuristic computation. This observation led to the development of a distributed extension of External A*. It is reported that not only the expansions can be distributed, but the duplicates removal too can be delegated to multiple processors. For LTL model checking, Distributed External A* has shown a significant gain in performance in a distributed environment. For the largest exploration (3 TB), the total time was reduced from 20 days to just 8 days when four processors were employed. In the area of model checking, this is the first algorithm that successfully combined external and distributed search for full LTL model checking.

EM Algorithms for Timed Automata and Priced Timed Automata
Timed automata provide the necessary formalism to model systems where the behavior is bounded to a set of clocks. The state spaces that result from the unfolding of timed automata are peculiar in the sense that no *total order* can be defined on them. However, by considering only the discrete part, one can define a total ordering on the *subsets* of a state space. To bring them under the EM framework, the external duplicate detection had to be redefined to accommodate this property. For priced timed automata, one is not only interested in just finding any path, but the one that also minimizes the cost. An EM branch-and-bound procedure is suggested for the exploration in priced TA. To accelerate the search, branch-and-bound is further equipped with iterative broadening that explores only a fraction of the state space to arrive at a (possibly) non-optimal solution. The solution is then gradually improved, until a fix-point is reached. It is further proved that if, while broadening, certain rules are observed, continuous convergence towards the optimal solution is guaranteed. The algorithms are implemented in UPPAAL-CORA and are evaluated on aircraft landing scheduling problems.

External Action Planning Graph-based action planning aims at delivering a sequence of actions that transform a given state to a state where given goal criterion is fulfilled. The new planning language, PDDL3, introduced temporal preferences and constraints that have resulted in non-monotonic cost functions in the state space. We have presented an EM algorithm for cost-optimal planning in such domains. For non-optimal but fast planning, an EM algorithm that follows Enforced Hill Climbing strategy has been suggested. To bound the duplicate detection scope in planning, a novel method is put forward that uses only the grounded description of the problem to find an upper-bound.

External Value Iteration for optimally solving MDPs, AND/OR graphs and Game trees
One of the major contributions of this thesis is the External Value Iteration algorithm that is able to deal with both non-deterministic and probabilistic state spaces. It optimally solves Bellman equations not only for Markov decision processes (MDPs), but also for AND/OR graphs and Game trees. In search algorithms, including the EM search algorithms presented in this dissertation, the information is always propagated forwards. In Value Iteration (VI), on the contrary, the flow of information is backward – each state receives information from its immediate successors. Unfortunately, for large graphs that do not fit into the RAM, the successors are not accessible in constant time. Due to this inherent difficulty, no I/O-efficient solution to this problem had been reported. A novel method is contributed that works on edges rather than on states. In External VI, by investing one sorting operation per iteration, all the updates in an iteration are available through just one scan of the edge list.

11.2 Future Work

I can see several ways in which the results presented in this dissertation might be extended or combined with other techniques. We enumerate some of them in the following.

Real-valued Weights in External A*
In External A*, we have only considered integer weights. The advantage of this approach is the fact that it yields a good partitioning of the whole search space into buckets. For the case of real-valued weights, the number of buckets required for a successful exploration increases. Many of them might contain just a few states, which in turn, will result in I/O operations with less than B elements. Using number intervals instead of concrete integers in the External A* matrix distribution, could lead to external heuristic search for real-valued weighted graphs.

External Partial Order Reduction in Model Checking
One major improvement that can be foreseen in external model checking would be the integration of partial order reduction (POR) techniques (Peled 1998). POR exploits the commutivity of actions to prune the state space. In asynchronous systems that involve extensive interleavings of many sub-systems, POR has been found to be particularly useful in reducing the number of states. Using a Breadth-First order, there is a significant chance of success for this integration.

Efficient Updates in External Value Iteration
Currently, External Value Iteration is very exhaustive: the whole state space must be scanned for updates. Recently, de Alfaro & Roy (2007) presented a technique to iterate only on predefined regions of the state space in order to accelerate Value Iteration. Accommodating this technique into External VI may increase the overall time efficiency of the algorithm.

External Probabilistic Model Checking
As the name implies, probabilistic model checking deals with the verification of systems that involve a probability distribution. Three types of probabilistic systems are mentioned in probabilistic model checking literature (Hinton *et al.* 2006; Baier, Ciesinski, & Grer 2005):

discrete-time Markov chains, continuous-time Markov chains, and MDPs. For continuous-time Markov chains, a disk-based algorithm has been presented by Mehmood (2004) and successfully integrated into the PRISM model checker. We believe our External Value Iteration can be used to alleviate the scope of probabilistic model checking, to large MDPs.

External Game-based Model Checking The algorithm External Value Iteration – with some modifications in the update function – can be used to solve large parity games (van de Pol & Weber 2008). In a nutshell, parity games are played on graphs, where the set of states is divided into two disjoint sets: states where moves are performed by the player *Eve*, and those where moves are performed by the player *Adam*. States are tagged with a priority which is an integer number from $[0, d)$ for a given d. An infinite sequence of states, or a play, is pronounced as 'won by *Eve*', if the least priority that appears infinitely often is even. The concept of a 'strategy' is analogous to the concept of policy in MDPs. A strategy is termed *winning on* a state u for the player α, if any play starting from u and following the strategy is won by α. To solve parity games, a widely used algorithm is by Jurdziński (2000) that associates a d-sized vector to each state. Iteratively, the vectors are updated by back-propagating the information based on the vectors of the successor states, until a fix-point is reached. The iterative updates are quite similar to the Bellman updates, except that here instead of a single real-number, a vector of integers is updated.

Parity games have their applications in μ-calculus model checking, where the system to be verified is modeled as a player, and a parity-game is played against an *adversary* that tries to falsify a property specification. A winning strategy for the adversary corresponds to the falsification of the property. A potential research direction for introducing EM exploration in μ-calculus model checking would be to extend the GEAR tool (Bakera *et al.* 2007). GEAR provides supports for Specification Patterns (Dwyer, Avrunin, & Corbett 1999), CTL and full modal μ-calculus. For solving parity-games, it implements the algorithm by Yoo (2007). The tool is developed within the jABC (Steffen *et al.* 2007) framework that facilitates a seamless integration of multiple levels of business logics to model a complete work-flow.

External C++ Program Model Checking
We have seen the verification of *models* of the systems. A parallel approach is to verify the actual implementations. Verification systems like StEAM (Mehler 2006) provide the facility of checking a multi-threaded C++ program directly. The advantages are two-fold. Firstly, a programmer is not burdened with the transformation of the system to a model. Secondly, the errors that could appear due to back-and-forth transformations can be avoided. StEAM has been extended for external and parallel search (Edelkamp, Jabbar, & Sulewski 2007). But the external search used in this extension is not very I/O efficient. Several alternative approaches presented in this dissertation can be applied directly on StEAM for I/O efficient program model checking.

External Guided Real-time Model Checking
Recently, a wide body of results for directed model checking in real-time systems has been contributed, such as (Kupferschmid *et al.* 2006; S. Kupferschmid & Larsen 2008). Some of the heuristics are based on pattern databases, and hence, are both admissible and consistent. These heuristics make External A* a good candidate for integrating directed and external search into real-time model checking.

External Model Checking on Flash Memory
A hard disk is a slow and fragile medium. With the new advancements in Flash memory, it can be foreseen that, in future, hard disks might be completely replaced by Flash memory devices such as USB sticks, and Solid State Disks (SSD). Flash memory provides faster random reads than magnetic hard disks, but at the cost of slow writes. Writing a data record, sometimes, involves initialization of a whole block from the memory area and then rewriting the old contents and the new data record on that region. The idea of using flash memory for large-scale search was first coined by Ajwani *et al.* (2008), where the MM-BFS algorithm for explicit graphs (cf. Chapter 2) has been extended to work on Flash memory. They suggested to keep the adjacency list on the Flash memory, which allows a quick retrieval of all the neighboring nodes.

At present, SSDs are not available in large enough sizes (more than 64 GB) to be able to contain a large model checking state space. However, Flash memory can be used to accelerate the duplicate detection process by acting as a small read-only hash table. We observed that most duplicates are found in the previous layer just expanded, which suggests to keep only that layer in the Flash memory to avoid external set subtraction. Since the layer has to be scanned once for expansion, this does not impose any extra I/O overhead.

11.3 Final Words

The field of External Memory algorithms for implicit graphs is a relatively new field, as opposed to the more mature body of research for explicit graphs. These implicit graphs differ from their explicit counterparts in three major ways:

- On-the-fly: The data is generated on-the-fly and is not given beforehand. In most of the cases, the actual space requirement is not even known in advance or is too complicated to compute.
- Large state sizes: A single state is a snapshot of a software or hardware system under consideration. For large systems, the state sizes can increase to several Kilobytes. Consequently, the time per state for internal memory operations, such as expansion and comparison, can even surpass the I/O time.
- Dense state spaces: As many operators are applicable on a state, the average branching factor encountered is often quite high – more than 50 has been observed for some of the model checking graphs. This density, in turn, produces a large number of duplicate states that affect both external and internal computational efforts.

Sometimes, the internal memory is not sufficient to even keep the result. Such types of scenarios are highly unlikely to appear in path search problems, but in policy search, where there may be many states involved, it is a plausible scenario. We have demonstrated that in some of the models, the policy size grows quadratically. Since any internal memory algorithm must store at least all states that are part of the optimal policy, all internal memory algorithms are destined to fail.

The focal point of this dissertation has been to explore the horizons of External Memory algorithms by investigating the possible applications in state space traversals. I hope, that these contributions will lead to a broadening of the scope of state space based approaches in a wide variety of disciplines. Still, *the search should continue – exploring the unexplored!*

List of Figures

List of Tables

Bibliography

Aggarwal, A., and Vitter, J. S. 1987. The I/O complexity of sorting and related problems. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 267 of *Lecture Notes in Computer Science (LNCS)*, 467–478. Springer. 39

Aggarwal, A., and Vitter, J. S. 1988. The input/output complexity of sorting and related problems. *Communications of the ACM* 31(9):1116–1127. 3, 9, 12, 196

Aggarwal, S.; Alonso, R.; and Courcoubetis, C. 1987. Distributed reachability analysis for protocol verification environments. In *Discrete Event Systems: Models and Application*, volume 103 of *Lecture Notes in Control and Information Sciences*, 40–56. Springer. 125, 140

Ajwani, D.; Malinger, I.; Meyer, U.; and Toledo, S. 2008. Characterizing the performance of Flash memory storage devices and its impact on algorithm design. Technical Report MPI-I-2008-1-001, Max-Planck Institut für Informatik, Germany. 200

Ajwani, D.; Dementiev, R.; and Meyer, U. 2006. A computational study of external memory bfs algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 601–610. ACM Press. 20, 40, 41

Ajwani, D.; Meyer, U.; and Osipov, V. 2007. Improved external memory BFS implementations. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 3–12. 20

Alur, R., and Dill, D. L. 1994. A theory of timed automata. *Theoretical Computer Science* 126:183–235. 145, 146, 147, 151

Arge, L.; Hinrichs, K.; Vahrenhold, J.; and Vitter, J. S. 1999. Efficient bulk operations on dynamic R-trees. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, volume 1619 of *Lecture Notes in Computer Science (LNCS)*, 328–348. Springer. 23

Arge, L.; Knudsen, M.; and Larsen, K. 1993. A general lower bound on the i/o-complexity of comparison-based algorithms. In *Workshop on Algorithms and Data Structures (WADS)*, volume 709 of *Lecture Notes in Computer Science (LNCS)*, 83–94. Springer. 39

Arge, L.; Procopiuc, O.; and Vitter, J. S. 2002. Implementing I/O-efficient data structures using TPIE. In *European Symposium on Algorithms (ESA)*, volume 2462 of *Lecture Notes in Computer Science (LNCS)*, 88–100. Springer. 23

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence Journal* 116:123–191. 169

Baier, C.; Ciesinski, F.; and Grer, M. 2005. ProbMela and model checking markov decision processes. *ACM Performance Evaluation Review* 32(4):22–27. 179, 198

Bakera, M.; Margaria, T.; Renner, C. D.; and Steffen, B. 2007. Verification, diagnosis and adaptation: Tool supported enhancement of the model-driven verification process. In *ISoLA Workshop*. 199

Bao, T., and Jones, M. 2005. Time-efficient model checking with magnetic disks. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science (LNCS)*, 526–540. Springer. 104

Barnat, J.; Brim, L.; Černá, I.; Moravec, P.; Ročkai, P.; and Šimeček, P. 2006. Divine – a tool for distributed verification (tool paper). In *International Conference on Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science (LNCS)*, 278–281. Springer. 142

Barnat, J.; Brim, L.; Šimeček, P.; and Weber, M. 2008. Revisiting resistance speeds up I/O-efficient LTL model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. To appear. 123

Barnat, J.; Brim, L.; and Černá, I. 2002. Property driven distribution of nested DFS. In *International Workshop on Verification and Computational Logic (VCL)*, 1–10. 68

Barnat, J.; Brim, L.; and Chaloupka, J. 2003. Parallel breadth-first search LTL model checking. In *International Conference on Automated Software Engineering (ASE)*, 106–115. 69, 107, 142

Barnat, J.; Brim, L.; and Chaloupka, J. 2005. From distributed memory cycle detection to parallel model checking. *Electronic Notes in Theoretical Computer Science* 133:21–39. 69

Barnat, J.; Brim, L.; and Rockai, P. 2007. Scalable multi-core LTL model-checking. In *Workshop on Model Checking Software (SPIN)*, volume 4595 of *Lecture Notes in Computer Science (LNCS)*, 187–203. 142

Barnat, J.; Brim, L.; and Stříbrná, J. 2001. Distributed LTL Model-Checking in SPIN. In *Workshop on Model Checking Software (SPIN)*, volume 2057 of *Lecture Notes in Computer Science (LNCS)*, 200–216. Springer. 141

Barnat, J.; Brim, L.; and Šimeček, P. 2007. I/O efficient accepting cycle detection. In *International Conference on Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science (LNCS)*, 281–293. Springer. 103, 122

Barto, A.; Bradtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence Journal* 72(1):81–138. 181, 188

Beasley, J. E.; Krishnamoorthy, M.; Sharaiha, Y. M.; and Abramson, D. 2000. Scheduling aircraft landings – the static case. *Transportation Science* 34(2):180–197. 159

Behrman, G.; Hune, T.; and Vaandrager, F. 2000. Distributed timed model checking - how the search order matters. In *International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science (LNCS)*, 216–231. Springer. 162

Behrmann, G.; Bengtsson, J.; David, A.; Larsen, K. G.; Pettersson, P.; and Yi, W. 2002. Uppaal implementation secrets. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, volume 2469 of *Lecture Notes in Computer Science (LNCS)*, 3–22. Springer. 151, 152

Behrmann, G.; Hune, T. S.; and Vaandrager, F. W. 2000. Distributed Timed Model Checking – How the Search Order Matters. In *International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science (LNCS)*, 216–231. Springer. 140

Behrmann, G.; Larsen, K. G.; and Rasmussen, J. I. 2005. Optimal scheduling using priced timed automata. In *ICAPS Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems*. 159

Bellman, R.; Kalaba, R.; and Kotin, B. 1963. Polynomial approximation – a new computational technique in dynamic programming. *Mathematical Computing* 17(8):155–161. 181

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press. 179

Bengtsson, J., and Yi, W. 2004. Timed automata: Semantics, algorithms and tools. In Desel, J.; Reisig, W.; and Rozenberg, G., eds., *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science (LNCS)*. Springer. 413–438. 148

Bérard, B.; M. Bidoit, A. F.; Laroussine, F.; Petit, A.; Petrucci, L.; Schoenebelen, P.; and McKenzie, P. 2001. *Systems and Software Verification*. Springer. 60, 63

Bergstra, J., and Klop, J. 1984. The algebra of recursively defined processes and the algebra of regular processes. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 172 of *Lecture Notes in Computer Science (LNCS)*, 82–95. Springer. 54

Bertsekas, D., and Tsitsiklis, J. 1996. *Neuro-Dynamic Programming*. Athena Scientific. 181

Bertsekas, D. 1995. *Dynamic Programming and Optimal Control, (2 Vols)*. Athena Scientific. 183

Biere, A.; Cimatti, A.; Clarke, E.; Strichman, O.; and Zhu, Y. 2003. Bounded model checking. In *Advances in Computers (volume 58)*. Academic Press. 113

Bollig, B.; Leucker, M.; and Weber, M. 2001. Parallel model checking for the alternation free μ-calculus. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science (LNCS)*, 543–558. Springer. 140

Bonet, B., and Geffner, H. 2005. An algorithm better than AO*. In *National Conference on Artificial Intelligence (AAAI)*, 1343–1348. 181

Bonet, B., and Geffner, H. 2006. Learning depth-first: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 142–151. AAAI Press. 180, 181, 188, 189

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* 11:1–94. 179

Brim, L.; Černá, I.; Moravec, P.; and Simsa, J. 2004. Accepting predecessors are better than back edges in distributed LTL model-checking. In *Formal Methods in Computer Aided Design (FMCAD)*, volume 3312 of *Lecture Notes in Computer Science (LNCS)*, 352–366. 69, 70, 71, 107

Burch, J. R.; Clarke, E. M.; McMillian, K. L.; and Hwang, J. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98(2):142–170. 70

Burkart, O., and Steffen, B. 1997. Model checking the full modal mu-calculus for infinite sequential processes. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1256 of *Lecture Notes in Computer Science (LNCS)*, 419–429. Springer. 56

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence Journal* 165–204. 167

Cassez, F.; David, A.; Fleury, E.; Larsen, K. G.; and Lime, D. 2005. Efficient on-the-fly algorithms for the analysis of timed games. In *Concurrency Theory (CONCUR)*, volume 3653 of *Lecture Notes in Computer Science (LNCS)*, 66–80. Springer. 146

Černá, I., and Pelánek, R. 2003a. Distributed explicit fair cycle detection (set based approach). In *Workshop on Model Checking Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science (LNCS)*, 49–73. Springer. 71, 107, 122, 142

Černá, I., and Pelánek, R. 2003b. Relating hierarchy of temporal properties to model checking. In *Mathematical Foundations of Computer Science (MFCS)*, volume 2747 of *Lecture Notes in Computer Science (LNCS)*, 318–327. Springer. 69

Chiang, Y.-J.; Goodrich, M. T.; Grove, E. F.; Tamasia, R.; Vengroff, D. E.; and Vitter, J. S. 1995. External memory graph algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 139–149. ACM Press. 108, 109

Christensen, S.; Kristensen, L. M.; and Mailund, T. 2001. A sweep-line method for state space exploration. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science (LNCS)*, 450–464. Springer. 104

Clarke, E. M.; Grumberg, O.; and Peled, D. A. 1999. *Model Checking*. MIT Press. 1, 53, 62, 67

Courcoubetis, C.; Vardi, M. Y.; Wolper, P.; and Yannakakis, M. 1992. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design* 1(2/3):275–288. 67, 68

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(4):318–334. 29

de Alfaro, L., and Roy, P. 2007. Magnifying-lens abstraction for markov decision processes. In *International Conference on Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science (LNCS)*, 325–338. Springer. 198

DeGiacomo, G., and Vardi, M. Y. 1999. Automata-theoretic approach to planning for temporally extended goals. In *European Conference on Planning (ECP)*, 226–238. 169

Dementiev, R. 2006. *Algorithm Engineering for Large Data Sets*. Ph.D. Dissertation, Saarland University, Germany. 23, 41

Dial, R. B. 1969. Shortest path forest with topological ordering. *Communications of the ACM* 632–633. 29

Dierks, H. 2005. Time, Abstraction and Heuristics – Automatic Verification and Planning of Timed Systems using Abstraction and Heuristics. Habilitation thesis. 160

Dijkstra, E. W. 1959. A note on two problems in connection with graphs. *Numerische Mathematik* 1:269–271. 26

Dwyer, M. B.; Avrunin, G. S.; and Corbett, J. C. 1999. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering (ICSE)*. 69, 199

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *European Conference on Planning (ECP)*, 135–147. 177

Edelkamp, S., and Jabbar, S. 2005. Accelerating external search with bitstate hashing. In *KI'05 (German Conference on AI) 19. Workshop on New Results in Planning, Scheduling and Design, Koblenz*. 106

Edelkamp, S., and Jabbar, S. 2006a. Cost-optimal external planning. In *National Conference on Artificial Intelligence (AAAI)*, 821–826. AAAI Press. 170

Edelkamp, S., and Jabbar, S. 2006b. Directed model checking petri nets. *Electronic Notes in Theoretical Computer Science (ENTCS)* 149(2):3–18. 167

Edelkamp, S., and Jabbar, S. 2006c. Large-scale directed model checking LTL. In Valmari., ed., *Workshop on Model Checking Software (SPIN)*, volume 3925 of *Lecture Notes in Computer Science (LNCS)*, 1–18. Springer. 101, 122, 123

Edelkamp, S., and Jabbar, S. 2007. Real-time model checking on secondary storage. In *ECAI'06 (European Conference on AI) Workshop on Model Checking and Artificial Intelligence (MoChArt'06), Revised Selected and Invited Papers*, volume 4428 of *Lecture Notes in Computer Science (LNCS)*, 67–83. Springer. 145

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, 81–92. 44

Edelkamp, S.; Jabbar, S.; and Bonet, B. 2007. External memory value iteration. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 28–135. AAAI Press. 179

Edelkamp, S.; Jabbar, S.; and Lluch Lafuente, A. 2005. Cost-algebraic heuristic search. In *National Conference on Artificial Intelligence (AAAI)*, 1362–1367. AAAI Press. 30

Edelkamp, S.; Jabbar, S.; and Nazih, M. 2006. Cost-optimal planning with constraints and preferences in large state spaces. In *ICAPS'06 (International Conference on Automated Planning and Scheduling) Workshop on Preferences and Soft Constraints in Planning*. 8, 170, 191

Edelkamp, S.; Jabbar, S.; and Schrödl, S. 2004. External A*. In Biundo; Frühwirth; and Palm., eds., *German Conference on Artificial Intelligence (KI)*, volume 3238 of *Lecture Notes in Artificial Intelligence (LNAI)*, 226–240. Springer. 25

Edelkamp, S.; Jabbar, S.; and Sulewski, D. 2007. Distributed verification of multi-threaded C++ programs. In *Parallel and Distributed Methods in Verification (PDMC)*. To appear in Electronic Notes in Theoretical Computer Science. 141, 199

Edelkamp, S.; Leue, S.; and Lluch Lafuente, A. 2004. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer* 5(2–3):247–267. 7, 71, 75, 99

Edelkamp, S.; Lluch Lafuente, A.; and Leue, S. 2001. Directed model-checking in HSF-Spin. In *Workshop on Model Checking Software (SPIN)*, Lecture Notes in Computer Science (LNCS), 57–79. Springer. 72

Edelkamp, S. 1999. *Data Structures and Learning Algorithms in State Space Search*. Ph.D. Dissertation, University of Freiburg, Germany. 34

Edelkamp, S. 2005. External symbolic heuristic search with pattern databases. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 51–60. AAAI Press. 45

Edelkamp, S. 2006. On the compilation of plan constraints and preferences. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 374–377. AAAI Press. 170

Ernst, A. T.; Krishnamoorthy, M.; and Storer, R. H. 1999. Heuristic and exact algorithms for scheduling aircraft landings. *Networks* 34(3):229–241. 159

Etessami, K. 2002. Automata-theoretic model checking. Lecture slides from 2nd International School on Formal Methods for the Design of Computer, Communication and Software Systems: Model Checking. 55, 56

Farias, D. P. D., and Roy, B. V. 2004. On constraint sampling in the linear programming approach to approximate dynamic programming. *Mathematics of Operations Research* 29(3):462–478. 182

Fernandez, J.; Mounier, L.; Jard, C.; and Jéron, T. 1992. On-the-fly verification of finite transition systems. *Formal Methods in System Design* 1:251–273. 54, 67

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence Journal* 2:189–208. 166

Fisler, K.; Fraer, R.; Kamhi, G.; Vardi, Y.; and Ynag, Y. 2001. Is there a best symbolic cycle detection algorithm. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science (LNCS)*, 420–434. Springer. 70, 122

Floyd, R. W. 1972. Permuting information in idealized two-level storage. In Miller, R., and Thatcher, J., eds., *Complexity of Computer Calculations*, 105–109. Plenum. 12

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124. 168

Garavel, H.; Mateescu, R.; and Smarandache, I. 2001. Parallel State Space Construction for Model-Checking. In *Workshop on Model Checking Software (SPIN)*, volume 2057 of *Lecture Notes in Computer Science (LNCS)*, 216–234. Springer. 140

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3. Technical report, Department of Electronics for Automation, University of Brescia. 169

Ginsberg, M., and Harvey, W. 1992. Iterative broadening. *Artificial Intelligence Journal* 55:367–383. 161

Godefroid, P. 1991. Using partial orders to improve automatic verification methods. In *Computer-Aided Verification (CAV)*, number 531 in Lecture Notes in Computer Science (LNCS), 176–185. Springer. 60

Godfrey, M. D., and Hendry, D. F. 1993. The computer as von Neuman planned it. *IEEE Annals of the History of Computing* 15(1):11–21. 10

Guestrin, C.; Koller, D.; and Parr, R. 2001. Max-norm projections for factored mdps. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 673–680. Morgan Kaufmann. 182

Hammer, M., and Weber, M. 2006. To store or not to store reloaded: Reclaiming memory on demand. In *Formal Methods: Applications and Technology, Post-proceedings of Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, volume 4346 of *Lecture Notes in Computer Science (LNCS)*, 51–66. Springer. 105

Hansen, E., and Zilberstein, S. 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129:35–62. 181, 188

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on Systems Science and Cybernetics* 4:100–107. 26, 27, 29

Haslum, P., and Jonsson, P. 2000. Planning with reduced operator set. In *Artificial Intelligence Planning and Scheduling (AIPS)*, 150–158. 174

Haslum, P. 2006. Improving heuristics through relaxed search. In *Journal of Artificial Intelligence Research*, volume 25, 233–267. 181

Haverkort, B. R.; Bell, A.; and Bohnenkamp, H. C. 1999. On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. In *Workshop on Petri Net and Performance Models*, 12–21. IEEE Computer Society Press. 140

Helmert, M. 2002. Decidability and undecidability results for planning with numerical state variables. In *Artificial Intelligence Planning and Scheduling (AIPS)*, 303–312. 168

Henzinger, T. A.; Kopke, P. W.; Puri, A.; and Varaiya, P. 1998. What's decidable about hybrid automata? *Journal of Computer and System Sciences (JCSS)* 57(1):94–124. 145

Hinton, A.; Kwiatkowska, M.; Norman, G.; and Parker, D. 2006. PRISM: A tool for automatic verification of probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science (LNCS)*, 441–444. Springer. 179, 198

Hirschberg, D. S. 1975. A linear space algorithm for computing common subsequences. *Communications of the ACM* 18(6):341–343. 38

Hoare, C. 1978. Communicating sequential processes. *Communications of the ACM* 21(8):666–677. 54

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302. 170, 171

Hoffmann, J. 2003a. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research* 20:291–341. 169, 170

Hoffmann, J. 2003b. The Metric FF planning system: Translating "Ignoring the delete list" to numerical state variables. *Journal of Artificial Intelligence Research* 20:291–341. 191

Holloway, C. M. 1997. Why engineers should consider formal methods. In *16th AIAA/IEEE Digital Avionics Systems Conference*, volume 1, 16–22. 53

Holub, W., and Tuma, P. 2007. Streaming state space: A method of distributed model verification. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE '07)*, 356–368. IEEE Press. 143

Holzmann, G. J., and Bosnacki, D. 2007. The design of a multi-core extension of the spin model checker. *IEEE Transactions on Software Engineering* 33(10):659–674. 140, 142

Holzmann, G. J.; Peled, D.; and Yannakakis, M. 1996. On nested depth first search. In *The Spin Verification System*, 23–32. American Mathematical Society. 68

Holzmann, G. J. 1990. *Design and Validation of Computer Protocols*. Prentice Hall. 98

Holzmann, G. J. 1998. An analysis of bitstate hashing. *Formal Methods in System Design* 13(3):287–305. 106

Holzmann, G. J. 2004. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley. 71

Howey, R.; Long, D.; and Fox, M. 2005. *Plan Validation and Mixed-Initiative Planning in Space Operations*, volume 117 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. chapter 6, 60–70. 175

Inggs, C., and Barringer, H. 2006. CTL* Model Checking on a Shared Memory Architecture. *Formal Methods in System Design* 29(2):135–155. 140

Jabbar, S., and Edelkamp, S. 2005. I/O efficient directed model checking. In Cousot., ed., *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3385 of *Lecture Notes in Computer Science (LNCS)*, 313–329. Springer. 101, 196

Jabbar, S., and Edelkamp, S. 2006. Parallel external directed model checking with linear i/o. In Emerson, and Namjoshi., eds., *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3855 of *LNCS*, 237–251. Springer. 125, 143

Jan, H. 1978. Progress report on the automatic and proven protocol verifier. *Computer Communication Review ACM SigComm* 8(1):15–16. 71

Jensen, R. M.; Bryant, R. E.; and Veloso, M. M. 2002. SetA*: An efficient BDD-based heuristic search algorithm. In *National Conference on Artificial Intelligence (AAAI)*, 668–673. 44

Jurdziński, M. 2000. Small progress measures for solving parity games. In *Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1770 of *Lecture Notes in Computer Science (LNCS)*, 290–301. Springer. 199

Kabanza, F., and Thiebaux, S. 2005. Search control in planing for termporally extended goals. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 130–139. 169

Kamel, M., and Leue, S. 2000. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *International Journal on Software Tools for Technology Transfer* 2(4):394–409. 101, 137

Katriel, I., and Meyer, U. 2002. Elementary graph algorithms in external memory. In *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science (LNCS)*. Springer. 62–84. 17

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning propositional logic, and stochastic search. In *National Conference on Artificial Intelligence (AAAI)*, 1194–1201. 113

Knuth, D. E. 1981. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Reading, Massachusetts: Addison-Wesley, second edition. 12

Korf, R. E., and Felner, A. 2002. *Chips Challenging Champions: Games, Computers and Artificial Intelligence*. Elsevier. chapter Disjoint Pattern Database Heuristics, 13–26. 29

Korf, R. E., and Schultze, P. 2005. Large-scale parallel breadth-first search. In *National Conference on Artificial Intelligence (AAAI)*, 1380–1385. AAAI Press. 142, 185

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence Journal* 27(1):97–109. 43

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence Journal* 42:189–211. 181

Korf, R. E. 2003. Breadth-first frontier search with delayed duplicate detection. In *Workshop on Model Checking and Artificial Intelligence (MoChArt)*, 87–92. 21

Korf, R. E. 2004. Best-first frontier search with delayed duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*. AAAI Press. 650–657. 46

Korf, R. E. 2005. Frontier search. *Journal of the ACM* 52(5):715–748. 21, 46

Krcal, P. 2003. Distributed explicit bounded ltl model checking. In *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier. 113

Kristensen, L. M., and Mailund, T. 2003. Path finding with the sweep-line method using external storage. In *Formal Methods and Software Engineering: 5th International Conference on Formal Engineering Methods, (ICFEM)*, volume 2885 of *Lecture Notes in Computer Science (LNCS)*, 319–337. Springer. 104

Kupferschmid, S.; Hoffmann, J.; Dierks, H.; and Behrmann, G. 2006. Adapting an AI planning heuristic for directed model checking. In *Workshop on Model Checking Software (SPIN)*, volume 3925 of *Lecture Notes in Computer Science (LNCS)*, 35–52. Springer. 162, 199

Lago, U. D.; Pistore, M.; and Traverso, P. 2002. Planning with a language for extended goals. In *National Conference on Artificial Intelligence (AAAI)*, 447–454. AAAI Press. 169

Larsen, K. G.; Larsson, F.; Petterson, P.; and Yi, W. 1997. Efficient verification of real-time systems: Compact data structures and state-space reduction. In *IEEE Real Time Systems Symposium*, 14–24. 145

Larsen, K. G.; Behrmann, G.; Brinksma, E.; Fehnker, A.; Hune, T. S.; Petterson, P.; and Romijn, J. 2001. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In *International Conference on Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science (LNCS)*, 493–505. Springer. 7, 145, 151

Lerda, F., and Sisto, R. 1999. Distributed-memory model checking with spin. In *Workshop on Model Checking Software (SPIN)*, volume 1680 of *Lecture Notes in Computer Science (LNCS)*, 22–39. Springer. 140

Lluch Lafuente, A.; Leue, S.; and Edelkamp, S. 2002. Partial order reduction in directed model checking. In *Workshop on Model Checking Software (SPIN)*, volume 2318 of *Lecture Notes in Computer Science (LNCS)*, 112–127. Springer. 73, 99

Lluch Lafuente, A. 2002. Simplified distributed LTL model checking by localizing cycles. Technical report, Institute of Computer Science, University of Freiburg. 68

Lluch Lafuente, A. 2003a. *Directed Search for the Verification of Communication Protocols*. Ph.D. Dissertation, University of Freiburg, Germany. 71, 72, 115, 116, 120, 153

Lluch Lafuente, A. 2003b. Symmetry reduction and heuristic search for error detection in model checking. In *Workshop on Model Checking and Artificial Intelligence (MoChArt)*. 99

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20:1–59. 166

May, J. M. 2001. *Parallel I/O for High Performance Computing*. Morgan Kaufmann. 9

McDermott, D., et al. 1998. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Committee. 165

Mehler, T. 2006. *Challenges and Applications of Assembly-Level C++ Model Checking*. Ph.D. Dissertation, University of Dortmund, Germany. 199

Mehlhorn, K., and Meyer, U. 2002. External-memory breadth-first search with sublinear I/O. In *European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science (LNCS)*, 723–735. Springer. 19

Mehlhorn, K. 1984. *Data Structures and Efficient Algorithms*. EATCS Monographs. Springer. 26

Mehmood, R. 2004. Serial disk-based analysis of large stochastic models. In *In Validation of Stochastic Systems: A Guide to Current Research*, volume 2925 of *Lecture Notes in Computer Science (LNCS)*, 230–255. Springer. 199

Meyer, U.; Sanders, P.; and Sibeyn, J. 2003. *Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science (LNCS)*. Springer. 1

Milner, R. 1980. *A Calculus of Communicating Systems*. Springer. 54

Müller-Olm, M.; Schmidt, D.; and Steffen, B. 1999. Model-checking: A tutorial introduction. In *Static Analysis Symposium (SAS)*, volume 1694 of *Lecture Notes in Computer Science (LNCS)*, 330–354. Springer. 1, 53, 60

Munagala, K., and Ranade, A. 1999. I/O-complexity of graph algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 687–694. ACM Press. 16, 40, 48, 185

Nilsson, N. J. 1980. *Principles of Artificial Intelligence*. Tioga Publishing Company. 181

Pearl, J. 1985. *Heuristics*. Addison-Wesley. 25, 37, 71, 73

Pednault, E. P. D. 1991. Generalizing nonlinear planning to handle complex goals and actions with context-dependent effects. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 240–245. 166

Peled, D. A. 1998. Ten years of partial order reduction. In *Computer-Aided Verification*, 17–28. 198

Penna, G. D.; Intrigila, B.; Tronci, E.; and Zilli, M. V. 2002. Exploiting transition locality in the disk based murϕ verifier. In *Formal Methods in Computer Aided Design (FMCAD)*, volume 2517 of *Lecture Notes in Computer Science (LNCS)*, 202–219. Springer. 103

Penna, G. D.; Intrigila, B.; Melatti, I.; Tronci, E.; and Zilli, M. V. 2003. Integrating RAM and disk based verification within the murphi verifier. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *Lecture Notes in Computer Science (LNCS)*, 277–282. Springer. 104

Petri, C. A. 1962. *Kommunikation mit Automaten*. Ph.D. Dissertation, University of Bonn (Germany). 54

Pistore, M., and Traverso, P. 2001. Planning as model checking for extended goals in non-deterministic domains. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 479–486. 169

Plaat, A.; Schaeffer, J.; Pijls, W.; and de Bruin, A. 1996. Best-first fixed-depth minimax algorithms. *Artificial Intelligence Journal* 87(1-2):255–293. 181

Pnueli, A. 1977. The temporal logic of programs. In *IEEE Symposium on Foundation of Computer Science*, 46–57. IEEE Press. 60

Rasmussen, J. I.; Larsen, K. G.; and Subramani, K. 2004. Resource-optimal scheduling using priced timed automata. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science (LNCS)*, 220–235. Springer. 145

Reif, J. H. 1985. Depth-first search is inherently sequential. *Information Processing Letters* 20:229–234. 141

Roscoe, A. W. 1994. Model-checking CSP. In *A classical mind: essays in honour of C. A. R. Hoare*. Prentice Hall International Ltd., Hertfordshire, UK. 353–378. 103

Russell, S., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Prentice Hall. 2

S. Kupferschmid, J. H., and Larsen, K. G. 2008. Fast directed model checking via russian doll abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. To appear. 199

Safra, S. 1998. On the complexity of omega-automata. In *Annual Symposium on Foundations of Computer Science*, 319–237. IEEE Computer Society.

Schuppan, V., and Biere, A. 2004. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer* 5(2–3):185–204. 107, 109, 110

Schuppan, V., and Biere, A. 2005. Liveness checking as safety checking for infinite state spaces. In *International Workshop on Verification of Infinite-State Systems (INFINITY)*, volume 149 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, 79–96. Elsevier. 110

Schuppan, V. 2006. *Liveness Checking as Safety checking to find shortest counterexamples to linear time properties*. Ph.D. Dissertation, ETH Zürich, Switzerland. 110

Sistla, A. P.; Vardi, M. Y.; and Wolper, P. 1987. The complementation problem for bchi automata with appplications to temporal logic. *Theoretical Computer Science* 49:217–237.

Steffen, B.; Margaria, T.; Nagel, R.; Jörges, S.; and Kubczak, C. 2007. Model-driven development with the jABC. In *Haifa Verification Conference (HVC'06): Revised Selected Papers*, volume 4383 of *Lecture Notes in Computer Science (LNCS)*, 92–108. Springer. 199

Steffen, B. 1991. Data flow analysis as model checking. In *Theoretical Aspects of Computer Software (TACS)*, volume 526 of *Lecture Notes in Computer Science (LNCS)*, 346–365. Springer. 60

Stern, U., and Dill, D. 1997. Parallelizing the murφ verifier. In *International Conference on Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science (LNCS)*, 256–267. Springer. 125, 128, 140

Stern, U., and Dill, D. 1998. Using magnetic disk instead of main memory in the murphi verifier. In *International Conference on Computer Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science (LNCS)*, 172–183. Springer. 103

Tarjan, R. E., and Fredman, M. L. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* 34(3):596–615. 29

Tarjan, R. 1972. Depth-first search and linear graph algorithms. *SIAM Journal of Computing* 1:146–160. 70, 107

Tesauro, G. 1995. Temporal difference learning and TD-Gammon. *Communications of the ACM* 38:58–68. 181

Tronci, E.; Penna, G. D.; Intrigila, B.; and Zilli, M. V. 2001. Exploiting transition locality in automatic verification. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *Lecture Notes in Computer Science (LNCS)*, 259–274. Springer. 104

Tsitsiklis, J., and Roy, B. V. 1996. Feature-based methods for large scale dynamic programming. *Machine Learning* 22:59–94. 181

van de Pol, J., and Weber, M. 2008. A multi-core solver for parity games. In *Parallel and Distributed Methods in Verification (PDMC)*. To appear. 199

Vardi, M., and Wolper, P. 1986. An automata-theoretic approach to automatic program verification. In *Logic in Computer Science (LICS)*, 332–344. 64

Wagner, D., and Willhalm, T. 2003. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *European Symposium on Algorithms (ESA)*, volume 2832 of *Lecture Notes in Computer Science (LNCS)*, 776–787. Springer. 26

Weber, M. 2007. An embeddable virtual machine for state space generation. In *Workshop on Model Checking Software (SPIN)*, volume 4595 of *Lecture Notes in Computer Science (LNCS)*, 168–186. Springer. 105

Wijs, A. J. 2007. *What to Do Next? Analysing and Optimising System Behaviour in Time*. Ph.D. Dissertation, Vrije Universiteit Amsterdam, The Netherlands. 161

Wingate, D., and Seppi, K. D. 2004. Cache performance of priority metrics for mdp solvers. In *AAAI Workshop on Learning and Planning in Markov Processes*, 103–106. 191

Wolper, P. 1983. Temporal logic can be more expressive. *Information and Control* 56:72–99. 65

Yang, C. H., and Dill, D. L. 1998. Validation with guided search of the state space. In *Conference on Design Automation (DAC)*, 599–604. 71

Yoo, H. 2007. *Error diagnosis during model-checking through animated strategy synthesis*. Ph.D. Dissertation, University of Dortmund, Germany. In German. 199

Zhang, W. 1999. Model checking operator procedures. In *Workshop on Model Checking Software (SPIN)*, volume 1680 of *Lecture Notes in Computer Science (LNCS)*, 200–215. Springer. 121

Zhou, R., and Hansen, E. 2004a. Breadth-first heuristic search. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 92–100. AAAI Press. 161

Zhou, R., and Hansen, E. 2004b. Structured duplicate detection in external-memory graph search. In *National Conference on Artificial Intelligence (AAAI)*, 683–689. AAAI Press. 47, 185

Zhou, R., and Hansen, E. A. 2005. Beam-stack search: Integrating backtracking with beam search. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 90–98. 161

Zhou, R., and Hansen, E. 2006a. Breadth-first heuristic search. *Artificial Intelligence Journal* 170:385–408. 48, 78, 82

Zhou, R., and Hansen, E. 2006b. Domain-independent structured duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*. AAAI Press. 177

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*, 1217–1222. AAAI Press. 143