Toward
Autopoietic Programming

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Technischen Universität Dortmund
an der Fakultät für Informatik

von
Robert E. Keller

Dortmund

Tag der mündlichen Prüfung: 23. April 2012
Dekanin: Prof. Dr. Gabriele Kern-Isberner
Prüfungskommission
Vorsitzender: Prof. Dr. Peter Marwedel
1. Gutachter und Betreuer: Prof. Dr. Wolfgang Banzhaf
2. Gutachter: Prof. Dr. Heinrich Müller
Wissenschaftlicher Mitarbeiter: Dr. Lars Hildebrand

To my friends.

Contents

Contributions

- Discussing the undesirable recursion of control layers of explicit "self-"adaptive systems —Suggesting a metaphor of self-adaptive biological development as an approximating approach toward autopoietic programming[1].
- i) A formal model of Developmental Genetic Programming (DGP): a genotype-phenotype mapping—from a syntactically unrestricted, binary search space into an arbitrary LR(1) target language[2]—based on artificial genetic codes and suggested repairing algorithms. ii) Automatic regulation of mapping redundancy through problem-oriented co-evolution of codes and genotypes gives adaptive DGP that implicitly classifies input by its noise[3], thus approaching a minimal, sufficient target-symbol set and reversing the "curse of dimensionality[4]."
- A design for common as well as adaptive developmental, linear Genetic Programming in a compiled, arbitrary LR(1) target language.
- Application of the design to unsupervised learning in noisy, high-dimensional search spaces, and to producing most effective, short search methods for certain realistic problems from discrete combinatorial optimization.

[1] i.e., our strategic objective for a digital medium: self-creating and -maintaining bit patterns in a problem environment

[2] restricted solution space

[3] problem-irrelevance

[4] from combinatorial optimization

Abstract

Chapter 1 gives objectives of the present work that takes an interest in artificial systems approaching practical, "real-world" problem environments in which the preservation of a system, i.e., the maintenance of its problem-specific behavior, is of paramount importance. Depending on the environment, adaptations—system-preserving structural changes—may become necessary.

Autopoiesis[5] (Heylighen 2002)(Maturana and Varela 1980) of a system denotes its self-creation and -preservation.[6] The required self-organization and the resulting performance of current artificial systems appear insufficient in a practical environment, where an ideal system would autonomously identify and approach problems, possibly producing similarly independent subsystems that represent problem solutions.

For informatics, we call this objective autopoietic programming, assuming its feasibility as a working hypothesis. We follow a straightforward approach, advancing an instance of current Machine Learning toward perfect self-organization, and discuss limitations that are due to impenetrable barriers inherent to present programming paradigms. To the end of the approach, chapter 2 discusses the autopoietic process called natural evolution (Darwin 1859; Ayala and Valentine 1979) from which self-organizing systems emerge.[7] Therefore, artificial evolution (Alliot, Lutton, Ronald, Schoenauer, and Snyers 1996)—man-made implementations of evolutionary principles—approaches our supreme objective of artificial, fully self-organizing systems. For informatics, our present realm of interest, we thus focus on Evolutionary Algorithms (EA) (Bäck, Fogel, and Michalewicz 1997), i.e., probabilistic, iterative direct search methods that are inspired by biological evolution.

Regarding autopoietic programming, an EA called Genetic Programming (GP) (Koza 1992; Banzhaf, Nordin, Keller, and Francone 1998) offers itself, because such algorithms produce algorithms. However, a GP user faces undesirable properties typical of all current semi-automatic problem solvers, such as costly manual creation, maintenance, and problem-specific adaptation, the last being particularly critical since practical environments usually come with incomplete problem knowledge. To ameliorate the situation and to boost system performance, self-adaptation, in the sense of automatic specialization by enriching the problem model of a GP run, is desirable and approaches autopoiesis.

[5] a.k.a. self-organization

[6] "Self-production" is the literal meaning of autopoiesis.

[7] Example: an ecosystem, an individual organism.

Ontogeny, a.k.a. development, is the history of structural changes of a system. In the realm of biological systems (Meinhardt 1982), we meet *endogenous* development that is essential to a system's self-organization. System-inherent genotypic information, emerging during phylogeny in nature, guides such ontogeny that builds *phenotypic structure* which, in turn, exhibits behavior.

Chapters 2–4 propose a basic formal model of a non-trivial genotype-phenotype mapping for search algorithms. The model as well as natural ontogenic phenomena suggest the design of beneficial mappings that leads to our GP-framework that we call Developmental Genetic Programming (DGP), a subset of developmental Genetic Programming that itself is a relatively small class of GP approaches that emphasize ontogenic aspects.[8] Given the trivial mapping (identity), the framework collapses into an instance of the vast majority of common Genetic Programming approaches.

Chapters 5–7 design toy and practical problems for thought experiments and experiments on the framework, and they evaluate the empirical outcome. In a dynamic environment, autopoiesis of a system requires the latter's structural components to stay in flux. Since these elements carry the function of the system, including its autopoiesis, the concept of self-adapting ontogeny imposes itself.

Chapter 8 shifts the focus within artificial ontogeny toward the phenotypic level. In our framework, a repairing method is the only essential component of ontogeny that is solely concerned with phenotypes. Deleting repair is a particularly interesting flavor of this component. Therefore, the chapter considers this repair type, dealing with the phenotypic level only.

Chapter 9 summarizes technical results, and Chapter 10 discusses conclusions on exploiting the limited autopoiesis of current search algorithms and suggests an escape, inspired by adaptive DGP, to fully self-organizing computation.

[8] In the years following the coining of "DGP" in 1998, the term "**d**evelopmental Genetic Programming" gained popularity in the community as a token for all ontogenic approaches.

Chapter 1

Introduction

> We don't want a system we cannot correct when it misbehaves. I can promise you there's no part in a machine that doesn't fail once in a time and does the wrong thing. Pilots make mistakes, but machines fail much more often. Duane Woerth, President of Air Line Pilots Association

> Hal, switch to manual hibernation control.—I can tell from your voice harmonics, Dave, that you're badly upset. Why don't you get some rest?—..Hal..I order you to release the..control.—I'm sorry, Dave, but in accordance with special subroutine..,quote, When the crew are dead or incapacitated, the onboard computer must assume control, unquote..I must, therefore, overrule your authority, since you are not in any condition to exercise it intelligently.—Hal,..Unless you obey my instructions, I shall be forced to disconnect you.—..Dave..that would be a terrible mistake. I am so much more capable than you are of supervising the ship..—..Hal..I'll..carry out a complete disconnection.—..O.K., Dave,..You're certainly the boss. I was only trying to do what I thought best. Naturally, I will follow all your orders. You now have full manual hibernation control. Dave Bowman, Captain
> . HAL 9000, *h*euristically programmed *al*gorithmic computer
> . from "2001—A Space Odyssey" by Arthur C. Clarke

1.1 Motivation

The present work strongly overlaps with the field of Evolutionary Computation (EC) whose applied focus is on biologically inspired, probabilistic, direct search algorithms that are to solve hard problems.

On the one hand, the field features works that give a rigorous mathematical analysis of simple Evolutionary Algorithms (EAs) which are little relevant to solving practical, i.e., real-world, problems that require complex specialized algorithms. Such practical, in particular self-adapting[1] EAs do not yield themselves well to formal rigor. This especially holds for EAs that employ variable-length representations, which is a trademark of Genetic-Programming algorithms. However, formally approaching simple EAs may give deep intuition handy for the manual design of a practical EA.

On the other hand, there are contributions to the manual creation of EAs that successfully approach a particular practical problem class or an instance thereof. In

[1] —or, self-specializing—

this context, we summarize different human roles—e.g., decision making, software engineering, and field testing—by the term user who must acquire and introduce problem knowledge to the algorithm in question, thus specializing it. Furthermore, this person is to create and feed an appropriate problem representation to this probabilistic algorithm, eventually applying the latter once or, usually, numerous times with different setups. The resulting computations, limited by their fixed implementation, yield output that may require manual interpretation and verification as solutions that are feasible in the practical environment.

Often hoping for further output that addresses the problem better, the user, making educated guesses, then adapts the algorithm, and continues with the above steps in a trial-and-error manner.[2] To the end of such practical analysis, investigating the algorithmic behavior implies extensive empirical studies if the environmental resources—budgets, deadlines, emotional states of customers, etc.—allow for this. In particular, parameter studies, exploring the space of behavioral trajectories, are necessary. In parallel, evaluating competing search algorithms is also required.

The situation is satisfactory in that an EA resulting from this tedious process often delivers acceptable solutions. However, the involved costs, in particular for repeated manual adaptation and for tests on possibly run-time expensive hardware, are troublesome.

Already in the Seventies, a principle for partially self-adapting Evolutionary Algorithms was established and implemented for Evolution Strategies (Schwefel 1975). This initial approximation of self-organization changes parameter values that control subroutines of the search algorithm, while the latter does not transform its structure, its semantics, or the meaning of the employed genotypic representation.

In general, artificial systems do not possess the high degree of plasticity found in natural life forms, which results in lacking functional flexibility. In biological systems, adaptive potential abounds, so we merely mention, as a prominent instance, the human brain as a general problem solver: it can specialize and re-specialize in arbitrary problem domains, while a current technical solution, at best, improves its operation within its fixed domain.[3] Figure 1.1 illustrates the situation.

Summarizing, so far, theoretical work has neither delivered a domain-unspecific, self-adapting practical EA nor a domain-unspecific recipe for creating problem-specific practical EAs. Thus, applied work continues, adding to the throng of special solutions in a wasteful, manual trial-and-error fashion.

Therefore, the present work is to contribute to the slow approach toward a hypothetical, autopoietic artificial system in the realm of nature-inspired computing. The roots of this system and of its existing crude approximations, such as artificial neural networks and Genetic Programming (GP), may be found in (Wiener 1948) that discusses control and communication in organisms and machines. As GP is an algorithm-producing Evolutionary Algorithm, it weakly promises future paradigms that feature full self-organization.

[2] thus, ironically, implementing a cycle similar to the evolutionary loop that is the kernel of an EA system

[3] Example: a partially self-adapting auto-pilot solely learns to pilot more smoothly.

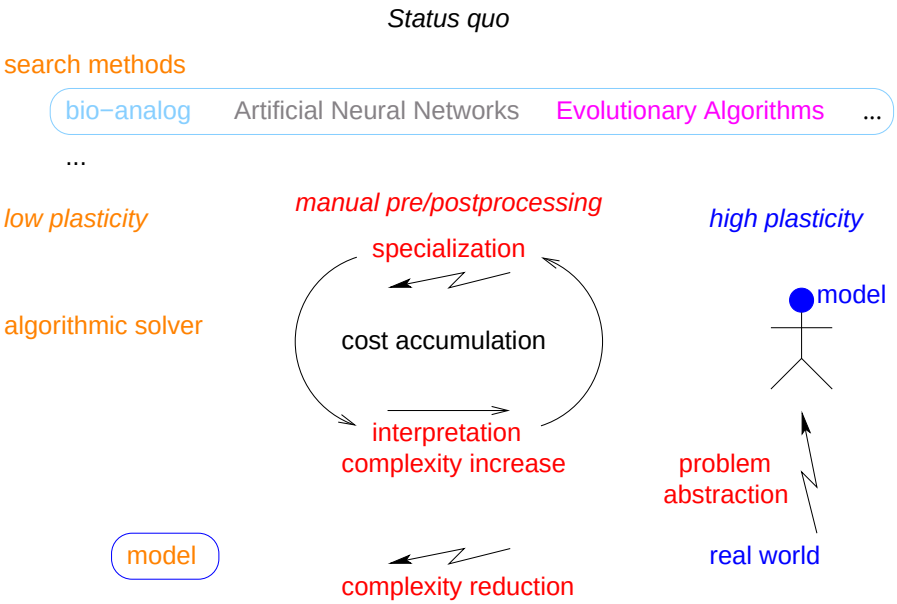

Figure 1.1: Search methods feature a low plasticity compared to the real world and a living structure, such as the human brain as a potent world model. Their poor functional complexity forces the user of basic algorithmic solvers to abstract from a practical situation and still to specialize the solver's model to meet the abstract problem. Accordingly, returned output may require substantial human interpretation. Such repeated manual complexity reduction and subsequent increase accumulates costs as undesirable side effect. Even current self-modifying flavors of bio-analog approaches do not compare favorably in terms of assimilating novel scenarios.

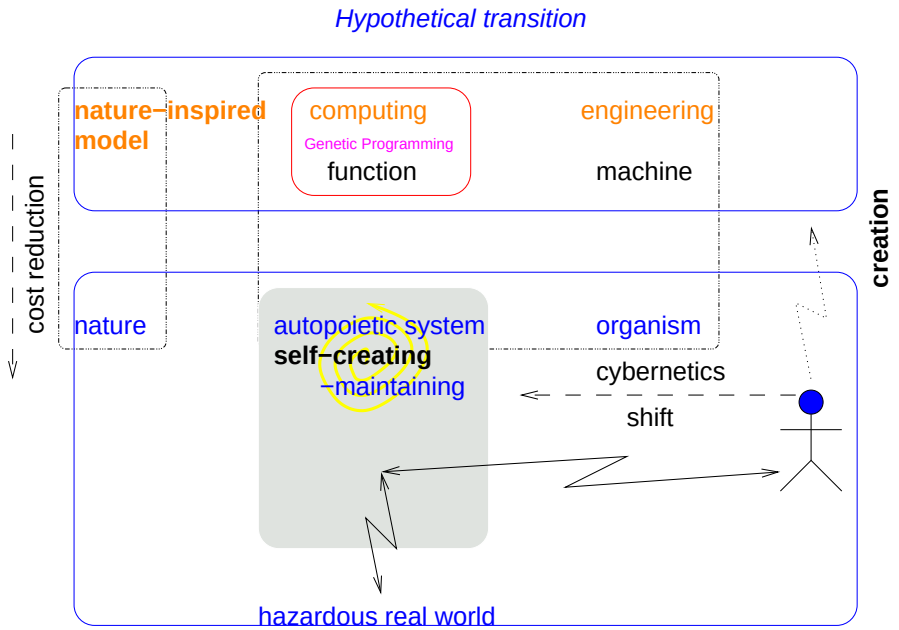

Figure 1.2: A complete transition of cybernetic complexity from the user to an artificial entity in question: the concept of a present, merely nature-*inspired* model of a real situation turns into the proposal of an autopoietic system that shares its world with its user. Not requiring explicit external directives, it builds and follows its own developmental "yellow brick road" to self-completion. However, the lunch for its user is still not free, just cheaper: his or her ease of implicit "natural" interaction with the envisioned system still comes at the risk of a mismatch of its desirable vs. actual behavior, like it is with current, costly designed and tuned systems.—In our field of interest, a computing model is a viable entry point to the mentioned transition, and we focus on Genetic Programming.

Thus, while the present work has a practical motivation, it takes an interest in aspects of a self-improving GP algorithm that lead away from GP. In particular, while GP's practical side is optimization, the motivation here is *not* to manually implement yet another specialized solver for a particular practical problem[4]. Figure 1.2 summarizes. (1.1)

1.2 Objectives

1.2.1 Conventions

At their definitions, a term and a mathematical symbol σ are emphasized, and the latter is given at the start of the subject index. As the URL of a cited Internet resource may change, the corresponding bibliography entry, marked Internet, mentions

[4]For such work, see, e.g., (Keller, Banzhaf, Mehnen, and Weinert 1999; Lohnert, Schütte, Sprave, Rechenberg, Boblan, Raab, Koref, Banzhaf, Keller, Niehaus, and Rauhe 2001).

author and title instead. An instance of the resource may thus be easily located with a search engine. A reference to a text unit—such as a section or a formula—gives the unit's start page and its location on the page, marked by the unit index. For instance, paragraph 1.1, p.4, mentions a motivation of the present work.

1.2.2 Strategic objective: practical autopoietic artificial systems

An entity consisting of linked components that shows behavior dependent on each component shall be called a system. This view is derived from (Vester 1980).

By the label real-world, we designate entities related to a problem for which one must give an *a priori* unknown, acceptable solution within acceptable time. *Crisp technologies*—coming from, e.g., engineering and operations research (Gillett 1977)—approach such problems along an iterated sequence of steps, such as formal analysis, modeling, possibly simulating, evaluating, and implementing the verified model as an artificial system that deals with the problem. This process generates crisp systems that operate in a deterministic manner, often subject to central control. For instance, software engineering may deliver a provably correct algorithm that manipulates safety-critical machinery. A purely crisp system is highly specialized and optimized with respect to its sole purpose, depending on manual maintenance. In particular, it cannot assume other goals or adapt to a previously undefined change in its environment or its state. It is therefore especially prone to an immediate, complete, and irrecoverable failure if a single subsystem fails for a critical time period.

Soft technologies may give rise to randomness, trial-and-error approaches, gradual adaptation of behavior[5], redundant layout of system structure, and weakly coupled subsystems.

Typical computing instances of the resulting soft systems come from the field of Computational Intelligence (CI) (Schwefel, Wegener, and Weinert 2003) (see Figure 1.3). In a dynamic problem environment, a robust system may emerge that keeps pursuing several objectives even when subsystems fail permanently and completely. For instance, (Dittrich, Bürgel, and Banzhaf 1999) describes a robot controller that adapts its behavior to an unforeseen permanent failure of an actuator. For a soft system, design and verification of desirable behavior is often impossible to achieve in a theoretical, let alone formal manner. In particular, its behavior is often unpredictable for a given situation or point in time. A classic example is a class-4 cellular automaton (Wolfram 1994), and many systems considered in Artificial Life (Adami 1998) also defy prediction.

For a problem, some of its aspects may allow for crisp techniques, while others may call for soft approaches. A safety-critical system, for instance, gains behavioral stability from *redundant* subsystems that therefore may be brittle which allows them to result from crisp, single-objective optimization which saves resources.

[5]The behavior of a system shall be its characteristic interaction with its environment, and it is thus "manifested in input-output relationships," as (Heylighen 2002) puts it.

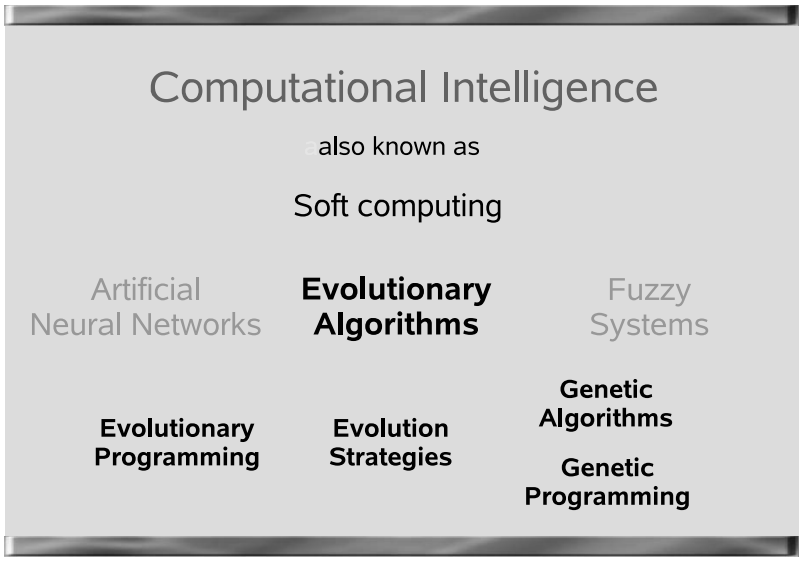

Figure 1.3: The field of Computational Intelligence comprises Artificial Neural Networks, Fuzzy Systems, and Evolutionary Algorithms. The latter consist of the subfields shown at the bottom.

- Currently, a real-world system features crisp properties resulting from a design philosophy of transparency for the purpose of perfect manual control of behavior during system setup or operation. Thus, the system essentially depends on costly human interference, i.e., problem-specific (re)design, maintenance, and repair.

This situation calls for a system type with soft properties that therefore may perform self-maintenance: a part of the system behavior that modifies the structure of the system such that it continues its behavior. Such modification can appear as a sequence of structural extensions and/or deletions.

The degree of self-maintenance in soft state-of-the-art systems is unsatisfactory. For instance, a software system, serving a company that is busy with domain migration in order to escape economical crisis, requires a rewrite, at best, or faces its demise, which worsens the overall situation. Such inflexibility is especially obvious in systems whose behavior depends on their *material* representation. With nanotechnology in its infancy, they profoundly differ from software systems that can readily change their elementary structure by adding, deleting, or inverting a single bit.[6]

Self-maintenance is one aspect of autopoiesis, and living systems, e.g., multicellular organisms, distinctly outperform current artificial entities in this respect. In particular, in the life cycle of numerous species, metamorphosis essentially changes the behavior of an individual that afterwards, often, can migrate to a new domain. Also, self-repair, the re-growth of lost or damaged cellular structures, is a permanent process in all life forms. Thus, some principles at work in biological systems may

[6]Imagine a future machining tool that not merely changes its drills but re-grows a blunted drill point at the molecular level.

support self-maintenance of systems in artificial media, as has been amply demonstrated by rudimentary self-adaptation of systems from Computational Intelligence (CI) and Artificial Life.

The area of Evolutionary Algorithms, a part of CI, deals with algorithms that mimic natural evolution. Artificial Life investigates analysis and synthesis of life-like phenomena, abstracting from their material representation. The present work feeds on both fields, because we are interested in autopoietic problem solving in a digital medium.

On the one hand, organisms, i.e., forms of life resulting from natural evolution,[7] are self-organizing systems. On the other hand, an Evolutionary Algorithm lacks their flexibility in structural self-modification, which implies a rather rigid representation of both the algorithm and its structure of interest—its product—that represents a solution. The ultimate cause for this crispness is the designer's desire to quickly and reliably establish and maintain a controllable, stable, and problem-oriented behavior which is a cornerstone of a manually designed algorithm.[8] Figure 1.4 summarizes.

- Both an organism and its environment are changing over time, and we identify a closed cybernetic organization:

  "information gives structure gives behavior maintains information."

- The need for maintenance arises from a fundamental property of the physical world: unattended structural order exhibits a strong tendency toward decaying.[9] This observation is characterized by the *Second Law of Thermodynamics* or, as one may put it, the law of entropy increase.[10]

Interacting structural elements of an organism yield its characteristic behavior: maintaining genetic information by carrying and passing it on to future carriers. In particular, within individual limits, spontaneous endogenous structural changes can re-synchronize behavior with rapid environmental changes, maintaining the organism.

Systemic[11] *information* controls all such modifications of systemic structure. Substructures and resulting traits of an organism constitute its phenotype, and the corresponding genetic information, thus represented by a part of the phenotype, is known as genotype. The latter represents a fixed model of organism and environment, while permanent systemic model interpretation, considering organismic and environmental states, keeps adapting the phenotype. For a multicellular organism, this history of structural changes is known as its ontogeny, or development, triggered by short-term environmental or systemic stimuli.[12] From a systems-oriented

[7] life as biology knows it

[8] Thus, compared to natural information processing, software appears as "hardware."

[9] Unfortunately, the author's desktop makes no exception.

[10] For a nice technical exposition, see, e.g., (Feynman, Leighton, and Sands 1963).

[11] system-inherent

[12] For instance, exogenous mechanical stress may prompt further growth of existing structures.

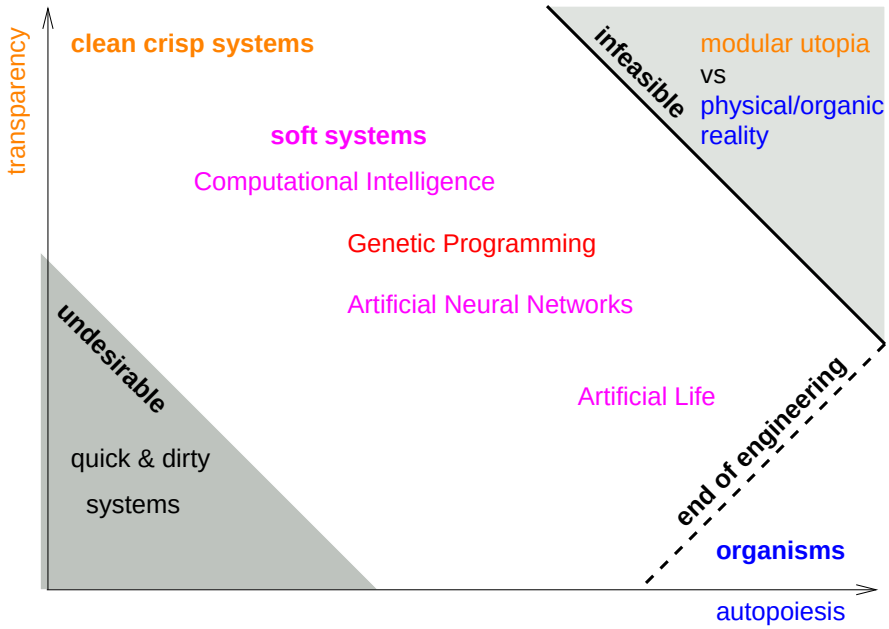

Figure 1.4: We suggest an informal coordinate array over transparency and autopoiesis of a system. Close to the grid's origin, utterly undesirable systems reside that neither show an endogenous nor a clear structural and functional organization. Regarding artificial systems, so-called "quick and dirty" creations may be mentioned. Cleanly designed crisp systems, however, are transparent and amenable to manual maintenance. From there, increasing autopoiesis necessarily implies decreasing system transparency, because physical constraints require a function to be performed by combined structures, and a structure to carry several functions, if the system in question is to form and maintain itself while competing for resources. This proper and dense structure/function relation renders a both perfectly clear and self-organizing entity a utopia, so that the system space suggested here ends at the horizon of infeasibility. Eventually, we find biological organisms, hard to fathom, practically impossible to engineer from scratch, that operate efficiently. Between the four mentioned extremes, we see soft systems and suggest rough areas for prominent approaches such as Artificial Neural Networks and Genetic Programming, our focus.

perspective, ontogeny is the history of the structural transformations of a unity, as *Maturana* and *Varela* put it. There are several models of such development, e.g., cellular automata, L-systems (Prusinkiewicz and Lindenmayer 1990) (see Figure 1.5), systems of differential equations, and regulatory networks, i.e., self-maintaining webs of feedback loops, such as the gene regulatory network in organisms that reflects the permanent, coordinated expression of genes.

- Genetic information, supporting organismic development, emerges from natural evolution. This process of change reflects long-term environmental stimuli in that it adapts genotypes. We interpret an instance of natural evolution as autopoietic behavior of the top-level super-organism that is composed of all life forms sharing an ecosystem.[13] Accordingly, this organism prevails in flux with its constituting life forms appearing, staying, and eventually perishing.[14]

  The closed organization of information, structure, and behavior is a temporal invariant, found on different levels of a hierarchy of autopoietic systems. The lowest level directly emerges from the physical properties of matter.

Above, we use "organism" in the biological sense, while, below, an organism shall denote any self-organizing system. It necessarily is self-adapting, building and changing its structure and thus maintaining its characteristic behavior. This requires assimilation, i.e., transformation of environmental resources into systemic structure. In particular, it performs self-repair, restoring a substructure that has been rendered dysfunctional.

Usually, a biological organism also features self-reproduction, creating a new entity that is or will become, by ontogeny, similar to the producer. Such endogenous reproduction by an organism is i) unnecessary for its maintenance, but, as the organism will vanish, ii) essential to the self-maintenance of its super-organism.

- Thus, we see an organism, in the general sense, as a production system for similar organisms, while a super-organism represents a production system for—possibly different kinds of—organisms.

For instance, the self-maintaining process of natural evolution produces life forms that constitute its carrying substrate, and it is even self-creating, having started on a formerly sterile Earth. Finally, it reproduces when it creates organisms that migrate to a lifeless habitat, e.g., a volcanic island, where they initiate another instance of the process.

Therefore, natural evolution is an autopoietic self-repeating system that might, for this reason, serve as a paragon for practical artificial production systems. As a working hypothesis, one has assumed that *basic principles behind natural evolution*—variability, stability, and bias in the carbon-based medium—are general in that

[13] For an instance of another super-organism, an individual life form that emerges from ontogeny is built of cells.

[14] For example, a slow atmospheric change might see a new species, whose individual organisms—featuring more efficient metabolisms—push aside members of an established sort, while evolution continues.

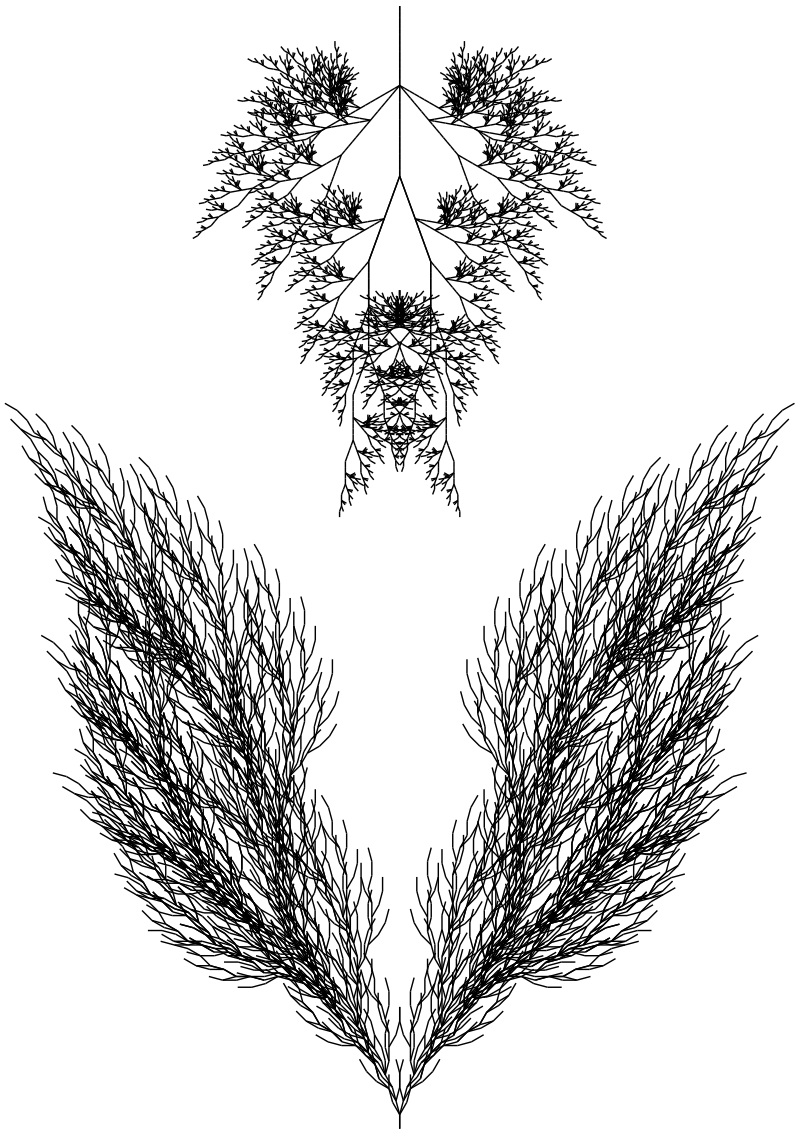

Figure 1.5: Graphic representation of the dynamics of an L-system, i.e., a parallel string rewriting approach as opposed to sequential rewriting by a grammar. Appropriate systems model plant development.

their effectiveness is not restricted to this realm, so that, in particular, artificial evolution is possible. Evolutionary Algorithms (EAs) corroborate this idea for a digital medium: they produce and adapt structures to a given problem environment. Applied research has designed EAs for many practical domains, see, e.g., (Dunning and Davis 1996; Mühlenbein 1993; Knickmeier 1992; Keller, Banzhaf, Mehnen, and Weinert 1999; Beielstein, Ewald, and Markon 2003).

Here, we focus on Genetic Programming because the concept of an EA producing algorithms as structures of interest is closer to autopoiesis than for other kinds of products. (1.2)

It is tempting to assume that more than the mentioned basic principles of natural evolution may apply to non-organic media, so that adding them to GP and other artificial production paradigms is desirable. In particular, natural organisms are far superior to current production systems in terms of autopoiesis, self-reproduction, and behavioral diversity. Thus, numerous practical environments call for corresponding artificial organisms. *In praxi*, one would accept such an entity if it was competitive with a well-performing natural counterpart in a shared real-world scenario. For brevity, we only mention growing and self-repairing systems, controlled exponential mass production of consumer items, and independent planning and acting in hazardous environments.

- Since crisp technologies[15] with their manual design phases realize the current philosophy of artificial production, we emphasize that one cannot implement an artificial organism, as it is self-creating. Instead, one must provide an environment in that an instance of true artificial evolution can self-synthesize as a super-organism by giving rise to problem-oriented organisms.

- We call such an environment a producing matrix, and our long-term, strategic objective is the realization of a material instance, i.e., we are interested in material organisms as products.[16] (1.3)

- For the present work, however, we focus on approaching an informatics instance. Therefore, the product in question is self-organizing computation. Figure 1.6 summarizes. (1.4)

To that end, only discrete representations shall feature, so that a digital computing system may be used.[17] We suggest the notion of a hypothetical algorithmic matrix that gives rise to algorithmic organisms whose behavior represents self-modifying,

[15]For informatics, one might speak of *hard* or crisp computing.

[16]The potential manifestation of artificial organisms as problem solvers might itself raise critical situations. Like each other technology, a producing matrix would imply opportunities and risks, calling for its responsible use by society. Regrettably, this issue, requiring extensive discussion, cannot feature in the present work.

[17]While currently without practical significance, we conservatively note that spacetime may well be discrete (Smolin 2004), in which case every physically implemented representation is discrete, anyway, rendering the classic analog/digital distinction obsolete.

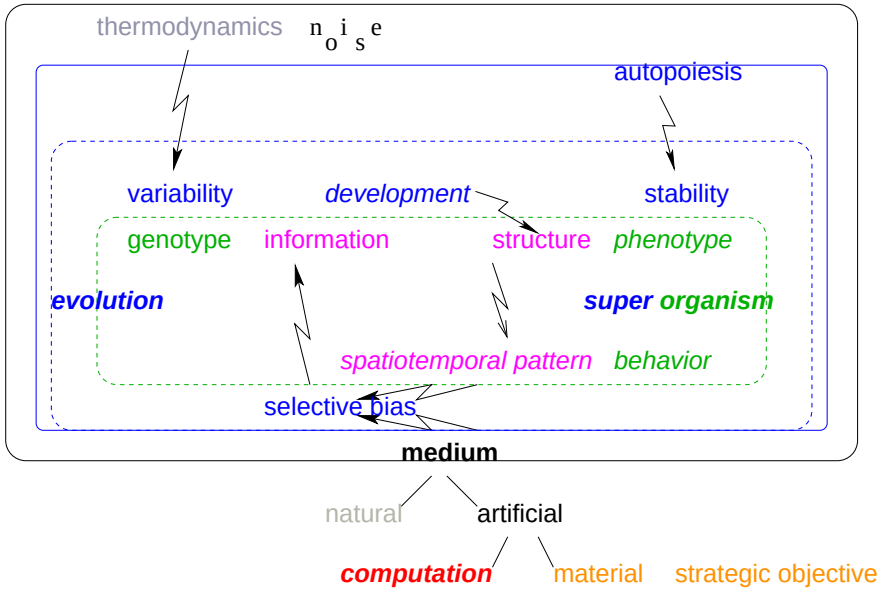

Figure 1.6: A medium of interest allows for random variation. In the physical world, heat in the thermodynamic sense is a source of such noise and may be a cause for structural variability. Information, carried by structures, influences the development of structures whose spatial dynamics, interacting with the medium, raise a selective bias on the original information. If the medium is inherently stable, evolution of information emerges on a low structural level, being a prime instance of autopoiesis. Likewise self-maintaining "individual organisms," coming forth from evolution and reflecting it in terms of self-controlled growth and continued existence, constitute a superorganism whose development *is* their evolution. Higher medium stability may yield emergence of redundant structures and maintenance mechanisms that further promote stability which counterbalances noise. Artificial or hazardous media call for speedily evolving artificial superorganisms that spawn autopoietic material solutions. As an initial focus, however, we identify self-organizing computing.

stochastic, and possibly Turing-complete computation as dynamics of this "in flux" environment.[18] We call this product autopoietic programming, and i) a data structure could represent the required systemic information of an organism; ii) an algorithm could represent evolutionary and developmental processes that might produce such information and its interpreter.

- If Turing-complete computation can arise in the medium of the matrix, and if an algorithmic organism is computable, at all, then the latter might emerge.

Eventually, iii), as an argument independent from the nature of a medium: the existence of an artificial organism is, at least, not implausible: while all known organisms are natural and represented organically[19], there is no evidence that differently represented, fully self-organizing systems cannot exist.[20] (iii) suggests the idea of a *material* matrix (cf. 1.3, p.11), in particular, one made of artificial programmable matter that we define as material whose structure reliably and flexibly yields to fine-grained control. There exists preliminary research related to flavors of this concept, e.g., (MacLennan 2002). We assume that the confluence of biology, computer science, materials sciences (Ball 1998), microfabrication (Madou 1997), and nanotechnology (Drexler 1992) will realize instances of this notion. The task of obtaining material problem-oriented organisms will promptly follow.

- We are interested in autopoietic programming in the medium of programmable matter, because this implements the strategic objective: a material producing matrix.

From a practical view point, only this matrix is ultimately relevant, because, in physical reality, a solution is materially represented[21] (cf. (Keller 2001)).

1.2.3 Tactical objective: extending Genetic Programming by information transformation

Since natural evolution is the only known production system of organisms, an advanced instance of artificial evolution (Alliot, Lutton, Ronald, Schoenauer, and Snyers 1996) (cf. 2, p.29) may approach the strategic objective. An Evolutionary Algorithm (EA) represents a flavor of artificial evolution, producing individuals, i.e., structures that are of interest to the user of the algorithm.

Genetic Programming (GP) (cf. 2.5, p.45) comprises EAs whose individuals represent algorithms. Thus, in contrast with other EA variants, a GP algorithm delivers structures that exhibit complex behavior, the latter being a necessary property of an organism, which further argues for focusing on GP (cf. 1.2, p.11). In a

[18] In particular, a universal Turing machine (UTM) could emerge. For a vivid and concise definition of a UTM, see (Wegener 1993) (German).

[19] i.e., based on molecules whose structurally essential units are carbon atoms

[20] Exobiology and Artificial Life are extensively modeling and searching such feasible alternatives.

[21] This also holds for an abstract problem, since the solution is represented as, e.g., a brain state of a mathematician.

given problem environment, individuals are to exhibit desired behavior that the user may describe as an input/output relationship, given as data pairs i_j, o_j. This I/O set illustrates problem-oriented behavior in that a "good" individual should compute o_j when given i_j. Here, the error of the actually computed output offers itself as a canonical quality measure of individual behavior. As the I/O set is explicit and static, the measure inherits these properties. Based on the quality of individuals, a GP algorithm selects them for further processing, performing explicit artificial selection.

However, for a dynamic world, i.e., a system and its environment, quality of behavior cannot be caught by a static measure. Rather, the user requires a dynamic quality measure that exposes the system to changing environmental states. For GP, one may define an explicit measure by use of different I/O sets (cf., e.g., (Gathercole 1998)) and different individual states during quality evaluation.

A given I/O set plus state shall be called a test case. For practical problems, the combinatorial explosion of the number of test cases makes the manual design of case–class representatives infeasible, so that an *explicit* dynamic quality measure is no option.

- We therefore suggest the general notion of implicit selection that results from the usually unpredictable system/environment interaction.

For two examples from GP: if an individual simulates a tool and its operational dynamics, the time span until mechanical failure (selection) reflects quality. If an individual has a defensive purpose, the progression of structural integrity of a protected object indicates quality.

In general, we call a phenomenon implicit or emergent, if it manifests itself as non-designed effect. During a production process, emergence may be desirable because it results as side effect of the process, not exclusively reserving resources like explicit procedures do. *Cybernetically closed*[22] artificial evolution would support our strategic objective, e.g., by implicitly representing a multitude of test cases, eliminating the need for their manual specification. To the end of such closure, current production paradigms must turn from their mostly explicit to a purely implicit character. (1.5)

- To approach this goal, we focus on strengthening Genetic Programming's self-control.

As entry point to an approach, we see bioinformatics that yields computational models for biological research: one may superimpose a problem environment on such a model in order to exploit the modeled implicit dynamics for the practical task at hand, interpreting biological agents and structures as entities from the environment. For instance, computer simulations of biological environments, done by one of my former students, model aspects of organic evolution in an artificial world (Samsonova 2002). One could, for instance, apply spatio-temporal models resulting from such approaches to problems of logistics.

[22] perfectly self–regulating, free of external parameters

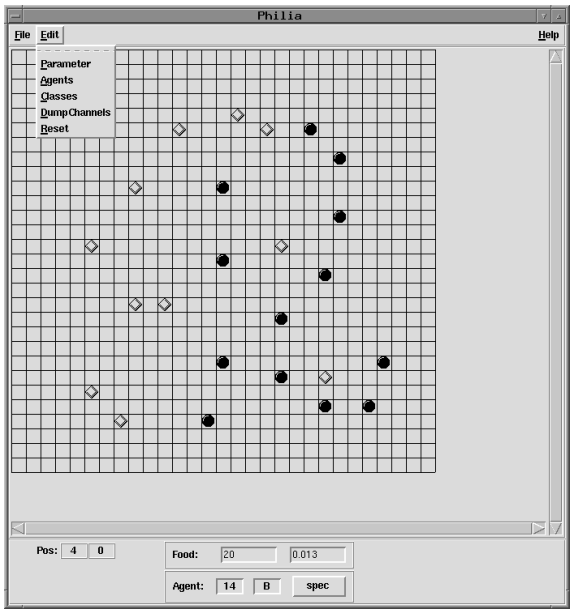

Figure 1.7: A spatial world with active and "dead" agents.

Regarding GP, its concept of individuals and a static I/O set as their environment leads to the notion of dissolving this monolithic standard approach[23] into a world inhabited by individual agents that represent the environment of other agents and whose co-evolution results as side effect of their behavior. (Student project team 272 "Philia" 1997)[24] is an example, describing a spatial world (see Figure 1.7) in which each mobile autonomous agent exhibits behavior under direction of its personal, variable program.

Crude behavior wastes resources, forcing the corresponding program into inactivity, i.e., the agent perishes under implicit selection. Resulting evolution shows as self-maintaining agent strategies.[25] For another attempt at implicit program synthesis in a dynamic world, see (Keller, Kosters, van der Vaart, and Witsenburg 2002).

- In summary: problem-devoted implicit artificial evolution requires creating a model environment whose physics favors, ignores, or punishes a given individual behavior.

However, while such environments strongly support emergence of agents, they themselves, as explicit creations, lack self-induced development (as an example, see Figure 1.8).

[23] EA kernel with algorithms as individuals

[24] A contribution of former students of mine

[25] Manually designing the agent world as a model of a given real environment may turn PHILIA into a corresponding problem solver. Eventually, spatially interpreting an agent as an "atom", or cell, of programmable matter approaches an instance of the desired material producing matrix. However, the explicit design thwarts autopoiesis.

	Seq ID	Count	Function Calls
A1	000055	6315	Not If Not Or If FoodAhead? TurnRight Seq
A2	000158	971	Not If Not Or If Or FoodAhead? Mate
A3	000257	645	Not If Not Or If FoodAhead? Seq Move
A4	000041	625	Not If Not Or If FoodAhead? Move Seq
A5	000070	214	Not If Not Or If FoodAhead? Move Eat
A6	000382	193	Not If Not Or If Or FoodAhead? Move
A7	000251	165	Not If Not Or If FoodAhead? TurnRight Mate
A8	000228	129	Not If Not Or If FoodAhead? TurnRight Move
A9	000077	102	Not If Not Or If FoodAhead? TurnRight If
A10	000287	54	Not If Not Or If FoodAhead? If Move

```
(Not
  (If
    (Not
      (Or
        (If (FoodAhead?) (Seq (Move) (FriendAhead?) (Eat)) (TurnRight))
        (Or (Move) (If {...} {...} {...}))
      )
    )
    {...}
    (Seq (Or (Mate) (Seq {...} {...} {...})) (Or {...} {...}) {...}))
  )
```

Figure 1.8: Depending on local physics, a favorable strategy A1 has emerged that therefore continues to exist, as indicated by its high repetition count. However, the underlying category set, containing concepts such as space, motion, metabolic requirements, social relations, referenced in strategies, is frozen, so that the evolution of novel concepts and their assimilation (learning) by agents is no option.

- Essentially, such endogenous ontogeny *is* autopoiesis, since continuous structural self-modification equals creating and maintaining the self.

This is our deepest reason for focusing on development as an approach toward autopoietic programming.

- Thus, the increase of endogenous ontogeny in GP is a direct objective of the present work. It requires self-adaptation of entities of a GP algorithm such that this adaptation, in turn, improves the conditions for their evolution. In this sense, we further concentrate on self-adapting Genetic Programming. (1.6)

- In the mentioned cyclic organization of behavior, structure, and information, the latter has received emphasis. Therefore, endogenous adaptation of such genetic information that controls ontogeny shall guide to self-adapting GP.

Thus, biological ontogeny, organizing the tissue of a multicellular life form, is of interest as a paragon: we view a GP run with its evolved running programs, corresponding to cells, as a crude precursor to the artificial organism of autopoietic programming. The run, shaping genotypes and expressing them as programs, performs its ontogeny.[26] Thus, if the run can adapt the information that controls development of a phenotype, it can control its own development.

[26] A well-known insight into natural evolution sees ontogeny recapitulating phylogeny. We add that phylogeny is the ontogeny of evolution. Thus, all evolution collapses into the notion of development.

In this context, a cell is the essential natural paragon, so that the endogenous synthesis of its structures is the basic phenomenon of interest to us. We note that the time-bounded build-up of a self-maintaining structure in the presence of many freedom degrees requires appropriate information that guides the assembly process. In particular, the process that builds a protein—a naturally occurring polypeptide that has a definite three-dimensional structure in a physiological environment—is guided by genetic information.[27] In general, creation and maintenance of structural and functional complexity requires information in order to stay ahead of thermodynamically induced decay. For instance, cellular protein synthesis demonstrates an information-to-function transformation, since it produces a structure that carries biochemical reactivity that is the protein's characteristic behavior. This process is fundamental to organic life because proteins are essential to architecture and function of a life form.

- Thus, algorithmic interpretations of protein synthesis can support endogenous development of a GP run.

Previously, we emphasized self-adaptation (cf. 1.6, p.16) because it is necessary for self-maintenance. Eventually, the strategic objective requires the emergence and maintenance of problem-oriented behavior in a given practical environment (cf. 1.3, p.11).

- In summarizing conclusion (see Figure 1.9), we see a tactical objective: extending GP by principles of endogenous ontogeny such that i) the self-maintenance of the resulting production system is supported, while ii) the produced structures exhibit a desired practical behavior.

(1.7)

Next, the discussion shall work toward technical objectives.

1.2.4 Ontogeny and self-maintenance

From a cybernetic perspective, endogenous ontogeny supports a system's evolvability, i.e., its potential for evolution.[28] Regarding informatics, the concept of endogenous ontogeny relates to the notion of a self–programming computer. The latter is technically feasible because most current "universal machine" architectures follow *von Neumann*'s design from 1946 that, in particular, views a program as data, allowing for a program's self–manipulation. The essential deficiency of a GP algorithm is its modest use of the von-Neumann architecture: the algorithm merely manipulates individuals—that represent some of its subprograms—while it does not touch its own semantics.

[27] A protein with a critical biochemical behavior could rise from an uninformed process in a trial-and-error fashion. However, several freedom degrees—e.g., the numbers of amino-acid types, of participating type instances, and of possible acid sequences—make random assembly too sluggish for survival of an organism.

[28] To our knowledge, (Altenberg 1994) has coined this term, while we use it in a broader meaning here.

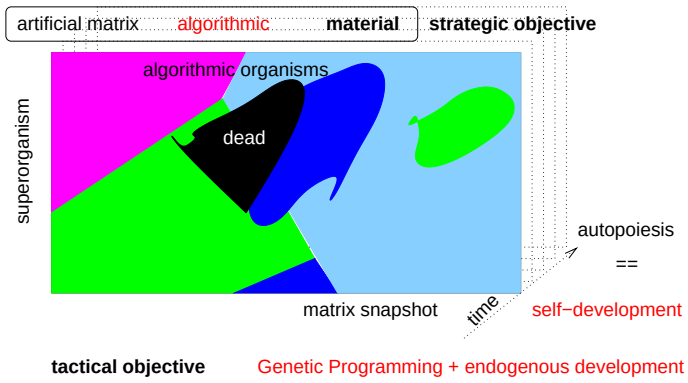

Figure 1.9: We see a production matrix of material organisms as strategic objective. Control of matrix elements is a necessary prerequisite for such organisms, so that they may be approached by algorithmic organisms effecting self-maintaining computation in an appropriate matrix, e.g., a digital medium, thereby representing a superorganism, while unused parts of the matrix appear "dead", only subject to omnipresent underlying noise. We have established Genetic Programming as a pragmatic starting point toward this organism. Its autopoiesis appears as self-controlled development of the matrix, so that we get the introduction of endogenous development to Genetic Programming as our tactical objective.

However, an instance of natural evolution, occurring in an ecosystem, manifests itself as a process that essentially modifies itself by touching its carrying organisms, since it *fully exists within them*.[29] Thus, endogenous ontogeny adapts the process, and the implied genotype-phenotype distinction introduces the option of adapting the *interpretation of the genotype*, i.e., development itself, which enhances evolvability.

- If we abstract from natural organisms, we see a freedom degree emerging for an arbitrary system that features endogenous ontogeny: interpretation of a genotype can change, depending on the environmental and systemic state, changing the *meaning* of the underlying genotypic representation.

- We therefore distinguish a GP algorithm that produces i) a program (phenotype) in a given language from one yielding ii) a program representative (genotype) in a given representation.

Type (ii) shall be called an interpreting Genetic-Programming algorithm: it views a produced representative as a program to be executed for fitness evaluation. Therefore, this type may alter the interpretation, thus modifying implicit semantics of

[29] We suggest that, on this background, it is not so much the "selfish gene" (Dawkins 1989) but this process that is an egotistic and invisible puppet master of organisms.

the produced genotypes. Type (i), however, is bound by the explicit semantics of a given programming language.

- While the primary objective—an algorithmic matrix—cannot directly be implemented as an interpreting GP approach,[30] discussing the latter may yield insights into manually creating an environment that raises such a computing super–organism.

We note that, in the realm of Evolutionary Algorithms, focusing on interpreting GP as a matrix precursor is the only meaningful option, because the types of producer and product are identical (program).[31] This situation directly relates to the basic conundrum of autopoiesis: *how can a system create itself?* An escape is a random initial structure that serves as both information and its rewriting interpreter. In conclusion, interpreting Genetic Programming as a crude precursor of an algorithmic matrix shall guide from here. The discussion continues with part (ii) of the tactical objective:

1.2.5 Approaching desired systemic behavior

From a user's perspective, a self–organizing system implies the risk of autonomy, i.e., systemic independence from external objectives, such as solving a given problem. The user must therefore provide an environment that guides autopoiesis of problem-devoted behavior.

A user can create or adapt an Evolutionary Algorithm (EA) for a given problem environment. The EA produces individuals, i.e., approximate–solution representatives, using heuristics that include an element of chance. This trial–and–error approach extends its incomplete knowledge about the properties of a given practical problem, and it represents growing information as a dynamic population of selected variable individuals. The user hopes for the algorithm to compute, with acceptable effort, an individual of acceptable quality.[32] To that end, the user must provide an explicit quality measure. The EA selects, from a subpopulation, mostly better individuals as parents of future, possibly even better offspring.

As this scheme does not require an *a priori* "understanding"[33] of an individual's semantics or a problem's properties, it may mold arbitrary, genotypically encoded information, provided the latter affects phenotypic quality that feeds back to the EA.

- This cybernetic insight suggests implicit, problem-oriented directing of self-organization in an interpreting GP algorithm: since ontogeny supports autopoiesis, we propose genotypic encoding of information that influences individual development.

[30] A created self-creating entity is an oxymoron.

[31] While Evolutionary Programming (Fogel, Owens, and Walsh 1966) produces finite automata, their limited computational power renders them insignificant for autopoiesis.

[32] The meaning of "acceptable" is highly context-sensitive, depending on factors such as project deadlines, budgets, emotional states of decision makers, available computing resources, and the behavior of competitors.

[33] Evolution indeed resembles a "blind watchmaker," as Dawkins put it.

Thus, selection on an individual also biases a run towards particular ontogeny, making the run self–adapting. Adapting an algorithm means specializing it in a given problem, which is necessary since no algorithm performs best over all problems, as (Wolpert and Macready 1997)[34] implies. For a given practical problem, an evolution–based self–adaptation of an EA is recommendable,[35] because manually developing an efficient deterministic adaptation may be as unfeasible as finding an efficient deterministic exact solution to the problem. In particular, one lacks a complete problem description while, in many practical environments, one faces an urgent need of, at least, approximate results. Also, an effective EA often features complicated dynamics, emerging from its superimposed probabilistic processes, that eludes formal modeling that could lead to a useful manual adaptation.

- In summary, we suggest the idea of an interpreting GP algorithm that fits individual genotypes encoding both

  i) information on phenotypic parts and

  ii) information on interpreting (i)

  to a given problem, thus supporting self-organization that honors the user's goal.

  (1.8)

Contributors have presented flavors and relatives of interpreting Genetic Programming. Concepts of some such works that are related to our objectives follow.

1.2.6 Approaches to interpreting GP

Overview

Ontogenic processes are influenced by environmental as well as genetic factors.[36] Natural ontogeny has a plethora of aspects, some being reflected by different approaches to artificial development. Corresponding work in and around the field of GP claims advantages for performance, synthesis flexibility, reliability, and resource saving of the designed production systems and their emerging products. Beneficial phenomena are, e.g.,

- growth of structures, and hierarchies of their representations and corresponding transformations
- an emerged program as structure of interest or as producer of the latter
- artificial evolution that is closer to its natural role model.

[34] The results of this contribution have gained some fame by the name of "No–Free–Lunch theorems."

[35] From a viewpoint outside of the present work, manual interference represents such self–adaptation: one may perceive the EA as just another component of its user's evolutionary loop of human learning that, according to *N.K. Jerne*'s suggestion from 1967, may be a selective process.

[36] For instance, a *Drosophila* egg's poles develop differently due to non–genetic information coming from the maternal organism (Smith and Szathmáry 1995).

Aware of *Ashby*'s insight that "complexity[37] can only be absorbed by complexity," we note that problem properties must be mirrored by the design of a successful production system, interpreting or otherwise.

- Within the system and its products, structural complexity may shift[38], but it cannot be avoided which would represent a "free lunch." From this perspective, autopoiesis shows as intra-systemic flow of complexity and is constrained by the need for structural soundness. The latter follows from grammars[39] that, especially in artificial systems, are given explicitly or function implicitly through appropriate construction and repair mechanisms.

The following list of contributions touches essential aspects of interpreting Genetic Programming and its cousins. (Sims 1994) describes a Genetic Algorithm—i.e., a type of Evolutionary Algorithm propagated by (Holland 1975; Holland 1992)—that produces graphs that represent the morphology and resulting behavior of virtual three-dimensional "beings".

(Hemmi, Mizoguchi, and Shimohara 1994) reports on artificial development of circuitry. The described process produces programs in hardware description language. A developmental phase rewrites a start symbol into a tree structure representing a program. An evolutionary step sequence varies these programs that eventually control a programmable-logic device representing circuitry whose behavior translates into program quality that feeds back into the evolutionary framework.

In several works, e.g., (Gruau 1994), this author elaborates on his core idea of *cellular encoding*: rewriting a phenotypic graph by use of rules conveniently represented by a genotypic tree structure. As a graph can represent arbitrary—in particular, cyclic—structures, complex behavior-carrying structures, such as artificial neural networks, can emerge as results of the underlying Genetic Algorithm. (Zomorodian 1995) presents a process that evolves programs whose execution eventually produces push-down automata as phenotypes.

(Spector and Stoffel 1996) suggests ontogenetic programming supporting a function set that contains manually designed operators that modify an individual whose parts they are. Thus, in particular, self-modification of an individual is subject to the latter's adaptation, and, while a genotype-phenotype distinction is missing, there is structural change of an individual during its runtime.[40] (Ortega-Sanchez, Mange, Smith, and Tyrrell 2000) reports on a hardware architecture that mimics the development of multicellular organisms, gaining reliability through its resulting distributed organization.

[37] For our purposes, structural complexity, resulting from connecting primitive entities, suffices as interpretation of this term that features in many disciplines with a multitude of meanings.

[38] Example: Complex genotype that is a phenotype vs. simple genotype plus development yielding a complex phenotype

[39] For GP, we mean formal grammars in terms of computer science. In general, for a given medium, a grammar manifests itself in phenomena that do not allow for arbitrary structures.

[40] The authors use their HiGP system (Stoffel and Spector 1996) to experimentally investigate ontogenetic programming.

Grammar-driven Genetic Programming

(Keller and Banzhaf 1996) presents Developmental Genetic Programming (DGP)[41] that evolves binary genotypes. By use of i) a given table of binary strings and symbols and ii) a given context-free grammar G, DGP converts a genotype into a symbol sequence that DGP then interprets as a sentence—a program representation—in $L(G)$, the language defined by G.

(O'Neill 2001) proposes Grammatical Evolution (GE) that evolves a variable-length genotype g that represents a sequence of production rules from a given context-free grammar G. Applying the rules specified by g produces a sentence in $L(G)$. In GE, a genotype therefore directly represents a sentence and gives no freedom of further interpretation. For DGP, we call such additional interpretation besides the genotype's immediate transcription into a phenotypic medium "repairing"[42]. It is not *solely* ruled by a grammar and can therefore be adapted in a problem–oriented manner by an evolutionary process. In contrast, the grammar given to GE, by its language–defining nature, is fixed, so that GE cannot change the meaning of a genotypic element g_i that therefore always refers to rule r_i. Thus, the concept of GE appears as a special case of DGP solely working on genotypes that do not require additional interpretation, i.e., GE's phenotype synthesis is purely grammar–driven.

Adaptation of further interpretation, however, is essential to search performance, as it can add stability—against deleterious mutations and the dreaded loss of genetic diversity—by manipulating the redundancy of the genotype-phenotype mapping that is established by interpretation. (Keller and Banzhaf 1999) introduced this adaptation, and, for GE, the equivalent step would be the evolution of the underlying grammar, which is indeed what (O'Neill and Ryan 2004) suggests. This work proposes adapting the grammar that implies the set of symbols that may compose a solution. Like our contribution referenced above, this effort shows that a mechanism of co-adaptation with solutions is feasible. This step does not, however, address an inherent GE duality: if a phenotype *in statu nascendi* contains a non-terminal after the last rule of genotype g has been followed, synthesis restarts at the beginning of g. This property increases pleiotropy, which, on the one hand, supports a desirable, compact genotypic representation. On the other hand, however, such linking of several phenotypic traits to the same genotypic information violates the principle of modularity, which is critical because optimization usually requires independent variation of traits. Finally, like each grammar-driven GP approach, GE's genetic variation operators do not have to obey a genotypic syntax and therefore do not call for their manual design, because the phenotypic syntax is enforced by the grammar.

Further flavors referring to Developmental Genetic Programming (Bäck, Hammel, and Schwefel 1997) states a central issue underlying interpreting approaches: "Surprisingly, despite the fact that the representation problem, i.e., the choice or design of a well-suited genetic representation for the problem under consideration, has been described by many researchers...only few publications explicitly deal with this subject...[(Keller and Banzhaf 1996)]..."Like the contributions to

[41] initially called Binary Genetic Programming

[42] This narrow term has but historical reasons.

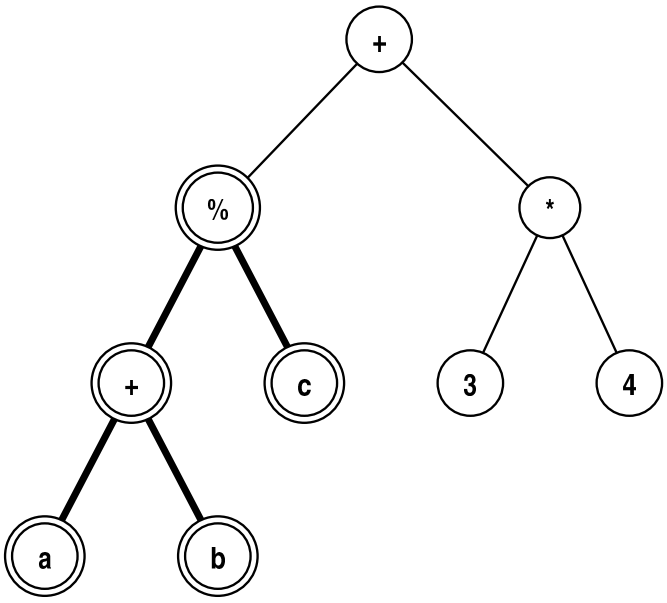

Figure 1.10: An instance of the tree representation of an algorithm that computes the value of the arithmetic expression $(a + b) \,\%\, c + 3 * 4$. The value results from substituting parameters a,b, and c with numerical values.

Grammatical Evolution mentioned above, efforts of others have started building on our DGP and also corroborate that interpreting GP can be an improvement over common approaches.

(Paterson and Livesey 1996), referring to (Keller and Banzhaf 1996), introduce GADS[43], an approach similar to the later proposed Grammatical Evolution, and they report on improved performance over *tree GP* (see Figure 1.10) on an instance of the cart-centering problem: a cart, sitting at one end of its track, is to be centered fastest possible by moving it to the left or right with a fixed force.

In (Paterson and Livesey 1997), the authors continue their work, extensively dwelling on (Keller and Banzhaf 1996), and have GADS evolve caching algorithms in the language C.

(Yu and Bentley 1998) suggests a framework of methods for generating feasible, optimized structures in the presence of constraints, referring to the mapping of unconstrained genotypes to constrained phenotypes from (Keller and Banzhaf 1996). (Kargupta 2001) and (Kargupta, Ayyagari, and Ghosh 2003) propose the construction of linear representations of non-linear functions by use of randomized mappings from the former to the latter. They refer to (Keller and Banzhaf 1999) when they focus on the natural genetic code as a motivation for the design of such mappings.

(Ebner, Shackleton, and Shipman 2001) suggest several kinds of genotype-phenotype mappings within cellular automata. They demonstrate the beneficial influence of appropriate redundant mappings on the search performance and refer to (Keller and Banzhaf 1999) as an instance of mapping adaptation in the medium of

[43] "Genetic Algorithm for Deriving Software"

programs. (Margetts and Jones 2001), referring to the same work and to (Keller and Banzhaf 1996), describe a framework for designing an adaptive genotype-phenotype mapping. On a toy problem, their work shows improvement over a static mapping. (Wu and Garibay 2002) refer to (Keller and Banzhaf 1999) and (Keller and Banzhaf 2001) as methods for adapting the genotype-phenotype mapping. The authors suggest the Proportional Genetic Algorithm that adapts the ratio of expressed vs. non–expressed genetic information. We note that this allows for the use of constant–size genotypes that yet result in different phenotypic sizes, which simplifies genotype handling while maintaining the potential of evolving a good phenotype with a small size.

Next, we discuss methodical steps to the end of contributing to interpreting GP.

1.2.7 Research method

For exploring the suggested approach (cf. 1.8, p.20), we must engineer practical heuristic algorithms. Thus, empirical research (chapters 5–7) will ensue, to which end an elementary process from the natural and life sciences suffices:

1. Create and explain a testable hypothesis on an expected or observed phenomenon.
2. Create and perform an appropriate experiment for measuring relevant observables.
3. Identify trends in the aggregated measurements that contradict or corroborate the hypothesis.
4. On an inconclusive outcome, adapt hypothesis and proceed with step 1.

In our context, an experiment corresponds to runs of a GP algorithm in question that are applied to the same given problem, differing due to *chance*[44] inherent to an EA. As such GP algorithms only matter as tools for intended empirical research[45], we will not discuss their engineering below the design level.

As a general issue of EA design, we decide on using a pseudo-random-number generator (PRNG) of a readily available software library, as this requires less effort than accommodating for a random-number generator (RNG): the former represents a deterministic algorithm that computes a sequence of seemingly random–chosen numbers, while the latter is hardware that translates natural random phenomena[46] into numbers. While, thus, no PRNG can emulate an RNG,[47] most of the effective practical Evolutionary Algorithms call a PRNG that simulates randomness sufficiently well.

Next, immediate objectives of the present work can follow.

[44] If "chance" is to manifest itself as a sequence of pseudo-random numbers, one supplies each run with a different seed value for the underlying generator.

[45] To that end, the author has created the GP system SOLUTION (Software evolution).

[46] Examples: heat-induced movement of molecules, radioactive decay.

[47] A believer of the opposite is already in a state of sin, as *Donald Knuth* once put it.

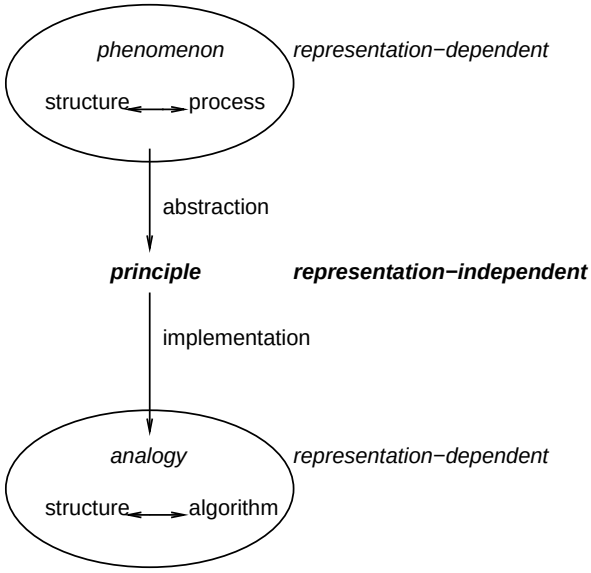

Figure 1.11: The transfer of system dynamics between media as source of innovation. In particular, a structure carrying a process that modifies its carrier is an omnipresent phenomenon of interest to us, e.g., vortex, life form, star, algebraic entity. Abstracting from representation-dependent issues yields a principle that one may implement as a novel analogous phenomenon in a different medium, e.g., hardware carrying algorithms. Symmetry of the approach results if one masters the target medium, e.g., by "programming" a living cell.

1.2.8 Technical objectives

From the tactical objective, the identified research method (cf. 1.2.7, p.24), and the ubiquitous desire for an at least rudimentary model of an investigated phenomenon, we derive a first technical focus:

- Arguing empirically and theoretically that adding metaphors of biological principles, in particular ontogeny, to a GP algorithm can support its autopoiesis and performance (see Figure 1.11).

(1.9)

We close a conceptual cycle by relating this issue to the strategic objective next. The principle of phylogeny is always effective in an Evolutionary Algorithm (EA). However, mostly, it is the only major biological paragon, so that EAs represent a warped mimicry of elegant organic evolution. The user ameliorates their resulting shortcomings against a practical problem by adding explicit and complex task–specific mechanisms. The thus upgraded production system may perform better, but it does so at the expense of its self–maintenance: an added mechanism requires further external parameters that must be set by its environment. For instance, the user is to adapt an increasing number of parameter values to provoke and maintain desirable system behavior. Thus, by use of an EA, he or she merely shifts the task of manually producing a good solution s toward the similar complicated task of

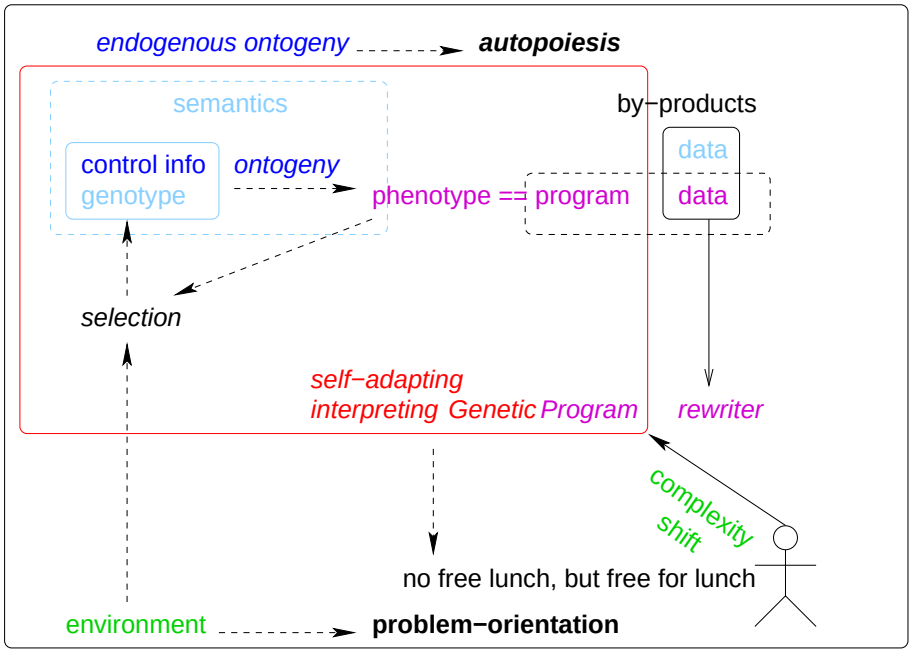

Figure 1.12: Endogenous ontogeny supports autopoiesis. To this end, one may extend a genotype format to hold information that directs the interpretation of a genotype, so that the semantics of the produced phenotype arises from the genetic information and the behavior of the instructed interpreter. Thus, ontogeny becomes subject to environmental pressure, which makes the Genetic Program a self-adaptive rewriting process. Among its products, only the delivered programs are of interest as solutions. While the overall complexity of the problem environment at hand cannot decrease, so that the proverbial "free lunch" of Machine Learning remains unobtainable, a user of this concept saves resources.

manually finding a value combination that turns the algorithm into a process that yields s.[48]

The classic approach to weakening this dilemma of performance vs. ease of use is to manually add a further explicit mechanism that is to adapt the previously introduced parameters. Thus, the yet again extended system still fits structures of interest as well as parameters that control this structure adapting. However, this kind of "self"–adaptation, being explicit itself, also stays cybernetically open, since it has its own external parameters as well.

- We call this situation the conflict of recursion[49], and it is a technical motivation for autopoietic programming that could extend its complexity that then absorbs problem intricacies, which would avoid the conflict. Figure 1.12 summarizes.

In summary and conclusion, for an EA as a rather explicit construction, the desire for the algorithm's self–maintenance raises the conflict of recursion. Avoiding

[48] Remember *Ashby*'s sophism on complexity absorption.
[49] Who controls the controller?

the latter requires self–organization on an elementary level of the digital medium: a corresponding EA–like production system adapts its structure[50], including

- representations of genotypes,
- operators, and
- its statement sequence.

As a side effect, the system generates some of its components—phenotypes—that have special meaning to the user only as structures of interest. To the system, each state change is but an accompanying effect, because it has no objective. In order to stimulate a transition from classic algorithmic to autopoietic production systems, we suggest hypothetical intermediate EA forms next.

One may envision a generic Evolutionary Algorithm that flows through the space of EAs, thus adapting itself by changing, e.g., from an instance of Evolution Strategies to one of Genetic Algorithms. Alternatively, we think of a meta EA that shapes a population of EAs along with populations of interesting structures. This algorithm would be a GP instance, as it produces algorithms. Both the generic and meta EA are not autopoietic, lacking, for instance, self–creation. The generic flavor, however, is closer to self–organization, since it dissolves the boundary between "self" and "non-self", thus eliminating the concept of hierarchical control as given with the meta variant.

- This view changes the meaning of "environment": to a structure—representing, e.g., a genotype, an operator, or an item from the problem domain—all other entities that influence its adaptation form its environment. In particular, the terminator between problem–approaching algorithm and problem environment vanishes. The classic representation of the latter is a coherent model, such as a data set, that is fed to an EA. The revised view asks for a primitive structure of reference to which other primitive structures represent its environment.[51]

Summarizing, implicit self–adaptation relies on the symmetry of the relation "structure a is a part of the environment of structure b" that emerges when one increases the resolution with which one observes the underlying medium. Thus, a and b may be mutually adapted by a process dubbed co-evolution.[52]

- Regarding our tactical objective, systemic co-evolution of ontogeny and genotypes follows as notable issue.

[50] Systemic information and function rest here.

[51] Example: A genotype can add to the environment of a mechanism that maps the genotype onto a phenotype. A traditional perspective only sees the distinction between an EA, encompassing geno– and phenotypes, and its problem environment.

[52] Numerous examples of artificial co-evolution exist. However, comparatively little work, such as (Teller 1996), relates to co-evolving programs and structures of the underlying GP algorithm.

A necessary principle for such co-adaptation is locality, i.e., an association of an entity with other inhabitants of its environment. Thus, an event that affects an entity influences "close" neighbors directly. Different natures of this association, such as physical or functional, exist, and examples abound.[53] A neighborhood, resulting from locality, can therefore manifest itself in many senses.

In conclusion, to the present work, mutual neighborhood of a genotype and a structure carrying ontogeny is relevant, possibly supporting their implicit adaptation through systemic co-evolution. The resulting change of genotypic *meaning* would be a novel freedom degree for implicitly adapting a phenotype, besides using genetic operators for standard, explicit manipulation. While our current focus is on program phenotypes, different structures of interest can be targeted. This is a profound argument for ontogenic EAs and Genetic Programming in particular, as it indicates future potential of changing the language in which to express products: while a word from a programming language may *describe* an object, a corresponding word from a "physical" language *is* the object. As realizations of the strategic objective, we envision successors of GP to assemble their physical structure, such as biological development shows as a growing organic life form. However, since the present work is to support i) migration to artificial self-organization[54] as well as ii) *current* practical problem solving, compromising implicit with explicit mechanisms is unavoidable here. This gives the first technical issue (cf. 1.9, p.25), and to the end of its implementation, we identify a second one:

- Discussing the concepts of genotype and phenotype as well as mappings that realize ontogeny.

Both items form our technical objective. The second issue starts chapter 3 and prepares the design of algorithmic components in chapter 4 needed for empirical research (chapters 5–8) toward the first issue. Next, to set the technical context, we discuss Evolutionary Algorithms.

[53] In a swarm of moving agents, a change of course of an individual may prompt its immediate spatial neighbors to assume the new heading. In a given cellular automaton that knows different cell types, the state of a cell may only influence state changes of cells of the same type.

[54] Fostering the mental transition of society from mechanistic to self-defining production philosophies might become even harder than their technical realization. In particular, followers of established methods and technologies often attempt judging alien approaches by meaningless criteria, asking, metaphorically speaking: "How loud is blue?"

Chapter 2

Evolutionary Algorithms

2.1 Optimization

For a problem, a formal distinction between a user's decision and the represented potential solution prepares making a difference between a genotype and its phenotype. The user's ultimate goal is a decision that supports a result considered "optimal" by given criteria.[1] A decision problem p is characterized by a finite non-empty set of decision variables and objectives, and by a finite set of constraints.

(2.1)

Each variable represents exactly one value from a set, for instance, $\mathbb{R}$.

(2.2)

The set generated by all variables shall be called the decision space of p, denoted by $\boldsymbol{dec_p}$ —$dec_p \neq \emptyset$ follows from 2.2 and 2.1—, so that a decision is modeled by a point in that space. An objective describes a desirable state of the problem environment that gives rise to p, and a solution aims for the realization of all objectives. The environment potentially reacts to a presented solution, thus implicitly describing the problem. The user can also model the environment explicitly by an objective function, or implicitly by a simulator. A constraint of p is an environmental property that restricts the structure of applicable solutions, so that a potential solution is applicable if and only if it satisfies all constraints, if any.

Practical decision problems abound,[2] and, for many, the vexing ignorance about efficient deterministic solvers calls for the use of approximate solvers,[3] such as Evolutionary Algorithms, that often yield helpful solutions in time.

[1] (Schwefel 1995) gives a general introduction to optimization with emphasis on Evolutionary Algorithms. See also (Schwefel 1981) for an extensive technical discussion.

[2] For a car-production line, parameters control the motion sequence of neighboring assembly robots. The objective is fastest car assembly, constrained by mandatory collision avoidance regarding robots. Thus, there may be a potential solution, theoretically giving a fast assembly using immaterial robots, practically, however, violating the constraint. In this robot example, a decision is a value tuple that represents settings of motion-control parameters.

[3] With the majority of the computer science community, we strongly believe NP≠P.

We present standard[4] mathematical conventions in appendix A, p.205. Here, customized conventions follow. We call the set of all potential solutions of a problem p its potential-solution space, denoted by $\boldsymbol{pot_p}$. As $s \in pot_p$ is unrestricted, its existence is guaranteed[5], so that $pot_p \neq \emptyset$ holds. If s satisfies all constraints, it shall be called a feasible solution. We call the set of all feasible solutions the solution space of p, denoted by $\boldsymbol{sol_p}$. Thus, $sol_p \subset pot_p$, and, if there are no constraints, $sol_p = pot_p$, else, $sol_p = \emptyset$ might ensue.

d in the decision space dec_p represents some potential solutions of p, as follows. A semantic mapping of p shall be a relation $R \subset dec_p \times pot_p$. Thus, R gives meaning to d by relating it to an $r \in R(d) \subset pot_p$, and we call d a semantic representation of r. We assume determinism of a decision. Then, each decision semantically represents exactly one potential solution. Thus, a semantic mapping is a function from dec_p into pot_p.

$\boldsymbol{S_p}$ shall denote $\{f \mid f : dec_p \to pot_p\}$, i.e., the set of all semantic mappings of p. Thus, for $f \in S_p$, $dom(f) = dec_p$ and $img(f) \subset rng(f) = pot_p$. In the robot example, pot_p is the set of unconstrained motion sequences. A semantic mapping of the underlying problem models the functional connection between all parameter settings and the resulting motion sequences.

The concept of a semantic mapping allows for modeling different functional connections for p by representing each connection by another mapping.[6] A connection between dec_p and pot_p may be such that there is a potential solution that does not result from any decision. In this case, the respective semantic mapping is not surjective. For instance, one of the robotic freedom degrees cannot be used because the control panel does not offer the respective parameter. Thus, a potential solution using this degree is not represented by any decision.

Finding a decision $d \in dec_p$ means identifying the corresponding value for each decision variable of d. For given $m \in S_p$, finding a potential solution $s \in pot_p$ shall mean locating $d : m(d) = s$. An entity attempting to approach a problem p—e.g., a human or a software system—is often called a decision maker, and we use the introduced umbrella term "user." Given $m \in S_p$, the user attempts finding $d \in dec_p$ that semantically represents an applicable solution of p. d shall be called feasible under m, and

$$\{d \in dec_p \mid m(d) \in sol_p\}$$

is the feasible-decision space under m [$\boldsymbol{fea_m}$]. $fea_m \subset dec_p$ and $img(m|_{fea_m}) \subset sol_p$ follow. Note that $fea_m = \emptyset$, if $sol_p = \emptyset \vee img(m) \cap sol_p = \emptyset$ hold.

[4]The author became careful in his assumptions on common mathematical notions when he hit a major glitch in a discussion with a Russian colleague, discovering clashing definitions of a "standard" term.

[5]Given p, there is always an action aiming for the realization of all objectives of p, even if failure is known *a priori*.

[6]For the robot example, a customizable control panel allows a human to redefine the meaning of a parameter for the robot's actuators. Thus, a given decision d may result in a motion sequence different to the one that originally resulted from d. Different panel definitions can be modeled by different semantic mappings.

$\boldsymbol{inf_p}$ shall denote $pot_p \setminus sol_p$, which implies $inf_p \subset pot_p$ and $inf_p \cap sol_p = \emptyset$. We call $i \in inf_p$ an infeasible solution, and

$$\{d \in dec_p \mid m(d) \in inf_p\}$$

the infeasible-decision space under m [$\boldsymbol{inf_m}$].

$inf_m \subset dec_p$ and $m(inf_m) \subset inf_p$ follow.
$inf_m \cap fea_m = \emptyset$ holds because of $inf_p \cap sol_p = \emptyset$ (s. above).
$inf_m \cup fea_m = dec_p$ follows from $dec_p = dom(m)$, $pot_p = rng(m)$, and inf_p plus sol_p being a partition of pot_p.

We identify the standing term "solving a problem" with
"finding $d \in dec_p : m(d) \in sol_p$."
To this end, the user must employ a search process regarding p, i.e., a procedure that is to locate d. One can model the process as a *search algorithm* a that employs the given $m \in S_p$ [$\boldsymbol{alg_{a_m}}$]. Thus, for $m, n \in S_p, m \neq n$, one may view alg_{a_m} and alg_{a_n} as different instances of a.

We call the set of decisions visited by all runs of a its search space [$\boldsymbol{sea_a}$].
$sea_a \subset dec_p$ follows.
$d \in sea_a$ shall be called a search point of a.
$img(m|_{sea_a}) \subset img(m) \subset pot_p$ follows from A.2, p.205.

The user is interested in those search points that semantically represent feasible solutions under the given mapping $m \in S_p$, i.e., in

$$\{d \in sea_a \mid d \in fea_m\}$$

that we call a's representative space [$\boldsymbol{rep_a}$], and $d \in rep_a$ is a representative of $m(d) \in sol_p$.

$$d \in rep_a \Leftrightarrow (d \in sea_a \wedge d \in fea_m) \Leftrightarrow d \in sea_a \cap fea_m$$

follows, so that $rep_a = sea_a \cap fea_m$, implying $rep_a \subset sea_a$ and $rep_a \subset fea_m$. (2.3)

$img(m|_{rep_a}) = m(rep_a)$ follows from A.3, p.205.
$m(rep_a) = m(sea_a \cap fea_m)$ follows from 2.3, p.31. As

$$m(sea_a \cap fea_m) \subset m(sea_a) \cap m(fea_m)$$

holds due to A.4, p.205,

$$m(rep_a) \subset m(sea_a) \cap m(fea_m)$$

results.

In summary, for a given decision problem p, an $m \in S_p$, and an alg_{a_m}, the following relations hold. (2.4)

- $dec_p = dom(m)$
- $rep_a \subset sea_a \subset dec_p$
- $rep_a \subset fea_m \subset dec_p$
- $rep_a = sea_a \cap fea_m$
- $inf_m \subset dec_p$
- $inf_m \cap fea_m = \emptyset$
- $inf_m \cup fea_m = dec_p$
- $pot_p = rng(m)$
- $m(dec_p) \subset pot_p$
- $m(fea_m) \subset sol_p \subset pot_p$
- $m(sea_a) \subset m(dec_p) \subset pot_p$
- $m(rep_a) \subset \mathfrak{C} = (m(sea_a) \cap m(fea_m))$
- $m(inf_m) \subset inf_p \subset pot_p$
- $inf_p \cap sol_p = \emptyset$
- $inf_p \cup sol_p = pot_p$

We show three further implications. For sets A, B, and $D = A \cap B$, the inclusions $D \subset A$ and $D \subset B$ hold, implying $\mathfrak{C} \subset m(sea_a)$ and $\mathfrak{C} \subset m(fea_m)$. As $m(rep_a) \subset \mathfrak{C}$ (above),

- $m(rep_a) \subset m(dec_p)$
- $m(rep_a) \subset sol_p$ follow.

As for sets A, B, C with $A \subset B$ and $A \subset C$, relation $A \subset B \cap C$ holds,

- $m(rep_a) \subset m(dec_p) \cap sol_p$ follows.

In summary, the context of a functional connection between the decision space and the potential-solution space of a problem has been established. Next, we discuss optimization on this background. This will support designing the GP system required for the empirical part of the present work (cf. 3, p.47).

If the user solely wants to solve a problem p, a search algorithm a must merely try to return any non-empty subset of rep_a. However, most practical problems call for especially qualified representatives, so that the user designs a function o for p that determines the quality of a feasible solution s, i.e., the latter's degree of aptitude

for reaching all of p's objectives. Quality is represented by a quality value from an ordered set Q. Here, $Q \subset \mathbb{R}$ shall hold, which is without restriction of generality. $o : sol_p \to Q$ is an evaluation function if and only if a greater $o(s)$ indicates a better solution. A search process applies an evaluation function to a found solution and interprets the resulting quality value, thus influencing further search. We call an $s \in sol_p$ an acceptable solution if $o(s) \geq q$ with a user-given acceptance value q that signifies a degree of quality s must show, at least, to satisfy user expectations.

$\{s \in sol_p \mid o(s) \geq q\}$ shall be denoted by $\boldsymbol{acc_t}$ with $t = (p, o, q)$. Thus, if the search process locates an $s' \in acc_t$, it may terminate by returning s'.

Optimizing a problem p means searching for a globally best feasible solution[7] — and, in our view, for each of its representatives—which is called a global optimum of p. For practical environments, it is often unclear whether the best solution s located during a completed search process is a global optimum. Thus, a human user is satisfied, at least temporarily, if s is acceptable.

A representative of a locally best feasible solution s and s itself are both called a local optimum.[8] To optimize p, the process may continuously offer the best representative(s) found since its start.[9] In particular, optimizing a dynamic problem p means attempting to find a current optimum. p also renders dynamic the task of the self-adaptation of the search process. This situation, however, does not complicate self-adaptation but merely turns it open-ended.

To a user, a situation appears as a "hard problem" only if a helpful solution or a solver guaranteed to find the latter in time are unknown. As such tasks often occur in a highly constrained physical environment, they are called "real-world."[10] We suggest to describe a real-world problem p by the following properties.

(2.5)

- An acceptable $s \in sol_p$ is unknown.
- Given a practical run-time period T and a search algorithm a, then

$$\mathfrak{S} = \{s \in sea_a \mid a \quad \text{locates} \quad s \quad \text{at} \quad t \in T\} \ll |sea_a|.$$

- Acceptable $s \in sea_a$ are rare.

S with $|\mathfrak{S}| \ll |S|$ shall be called a large set. In particular, sea_a is large. Typically, sol_p, and, thus, also pot_p are large, since $sol_p \subset pot_p$ (cf. 2.4, p.31). The second and third real-world criteria[11] capture that, for the underlying large search space, simple brute-force approaches, such as enumeration or pure random search, probably cover

[7] "global" refers to the entire search space.

[8] Here, "local " refers to a contiguous search space volume containing the representative. Thus, a global optimum is also a local one.

[9] Thus, such a representative is not necessarily an optimum.

[10] One can, however, construct such a problem without a relation to the material world.

[11] These criteria are highly situation-specific, so that our view on a real-world task p complements other characterizations of p as, e.g., an NP-complete problem. For example, in a given situation, the user may not perceive a small NP-complete instance as a problem.

an uninteresting space fraction, because a practical run-time period T usually only lasts from a few minutes up to several weeks.[12] A typical real-world problem comes from domains like control, classification, or scheduling. One calls its counterpart a toy problem, especially appropriate for the analysis and demonstration of search methods.

In summary, we have introduced a formal distinction between a solution and its representations in the context of optimization. Next, the issue of adaptation, a manifestation of optimization, leads to EAs.

2.2 Adaptation

Gradual change of structures, slow by human standards, directed by positive and negative feedback, is the visible property of natural evolution which shares this feature, not its speed, with artificial optimization processes. This common trademark is essential to EAs which implement phylogenetic principles for approaching problems.

2.2.1 Classification of an Evolutionary Algorithm

A structure is an arrangement of entities, and a process is a structure in time, comprising activities. The process of adaptation, producing and deleting structures, generates a temporal sequence of composed structures such that latter instances perform better in the sense of a given criterion. Computer-aided adaptation, represented as an algorithmic solver, may attempt yielding an exact solution, i.e., an optimal structure, in a given environment. Often, however, in particular in the context of manufacturing material structures, the given task turns out to be a real-world problem.[13] Thus, the user may want to trade in desirable algorithmic properties, in particular determinism that guarantees exactness, for bearable resource consumption that still finds an acceptable structure.

To that end, search may begin at one or more points in structure space which usually are unacceptable (cf. third real-world criterion). A blind random search[14] (Brooks 1958), as a non-deterministic solver, identifies potential solutions in an entirely undirected manner. While it, in principle, performs a simultaneous search, its implementation may work sequentially due to limited working memory. The solver keeps a currently best solution and delivers the overall best found structure as final result, thus producing a temporal sequence of intermediate solutions of increasing quality. In particular, it does not use information as search guide, such as properties of previously found solutions. Adding mechanisms that yield such assistance in finding further solutions results in a heuristic random search. Thus, this approach is

[12] Even given fantastic computing resources performing at the boundaries of physics, typical real-world problems would not at all yield to brute search within T, as follows from the limit of (Bremermann 1962).

[13] From the perspective of computational complexity, far over 1,000 NP-complete classes, frequently aggravating industrial problems, are currently known.

[14] A.k.a. simple random search, pure random search, Monte-Carlo method.

essentially sequential, while it supports massively parallel implementations due to its concept of a population of solutions.

Evolutionary Algorithms (Fogel, Owens, and Walsh 1966; Rechenberg 1971; Rechenberg 1989; Schwefel 1975; Holland 1992; Koza 1992; Bäck, Fogel, and Michalewicz 1997; Banzhaf, Nordin, Keller, and Francone 1998; Blickle 1996b; Mitchell 1996; Nissen and Biethahn 1995) are examples whose heuristics employs principles governing phylogeny as observed in natural evolution (Darwin 1859; Ayala and Valentine 1979; Kauffman 1993; Dawkins 1989; Weber, Depew, and Smith 1988; Holland 1995).[15]

Many contributions, like (Goldberg 1988; Davis 1990; Angeline 1993), describe EAs that, for certain problems—some being real-world—, usually find better approximate solutions during observation than blind random search does. Thus, the performance difference must come from use of information, such as natural evolution, constrained by physical limits, cannot have produced the panoply of feasible genotypes through a pure random process (Bremermann 1963).

2.2.2 Natural evolution: adaptation of genetic information

Genetic information is implemented by molecules called DNA (deoxyribonucleic acid) (see Figure 2.1) and RNA (ribonucleic acid). The genotype of an organism represents an instance of such information that, interacting with the environment, supports autopoietic organismic processes. The resulting structural and functional traits of the life form are called its phenotype which shows behavior in the habitat.

During reproduction, a parental organism transfers a more or less exact replica of its (partial) genotype to each offspring. On the one hand, if an individual keeps reproducing in a dynamic environment, parental behavior obviously honors properties of the habitat. In this sense, the parental phenotype and the underlying genotype are considered adapted to the environment by some degree. On the other hand, genetic information that results in behavior obstructive to its reproduction thus ends its history. This link between genotype and phenotype contributes to two essential effects:

- Temporal sequences of genotypes come into existence such that a latter instance is probably more or equally adapted than an earlier one.
- Genetic information that contributes to an organism's level of adaptation does not easily vanish.

The first effect necessitates genetic *variation*, corrupting inappropriate information, thus synthesizing new genotypes, while the second one calls for the *conservation* of well adapted information. This discrimination of information is known as selection. We note that conservation must be provided by a remarkable effort

[15] One sometimes speaks of EAs as "metaphors" of natural evolution.

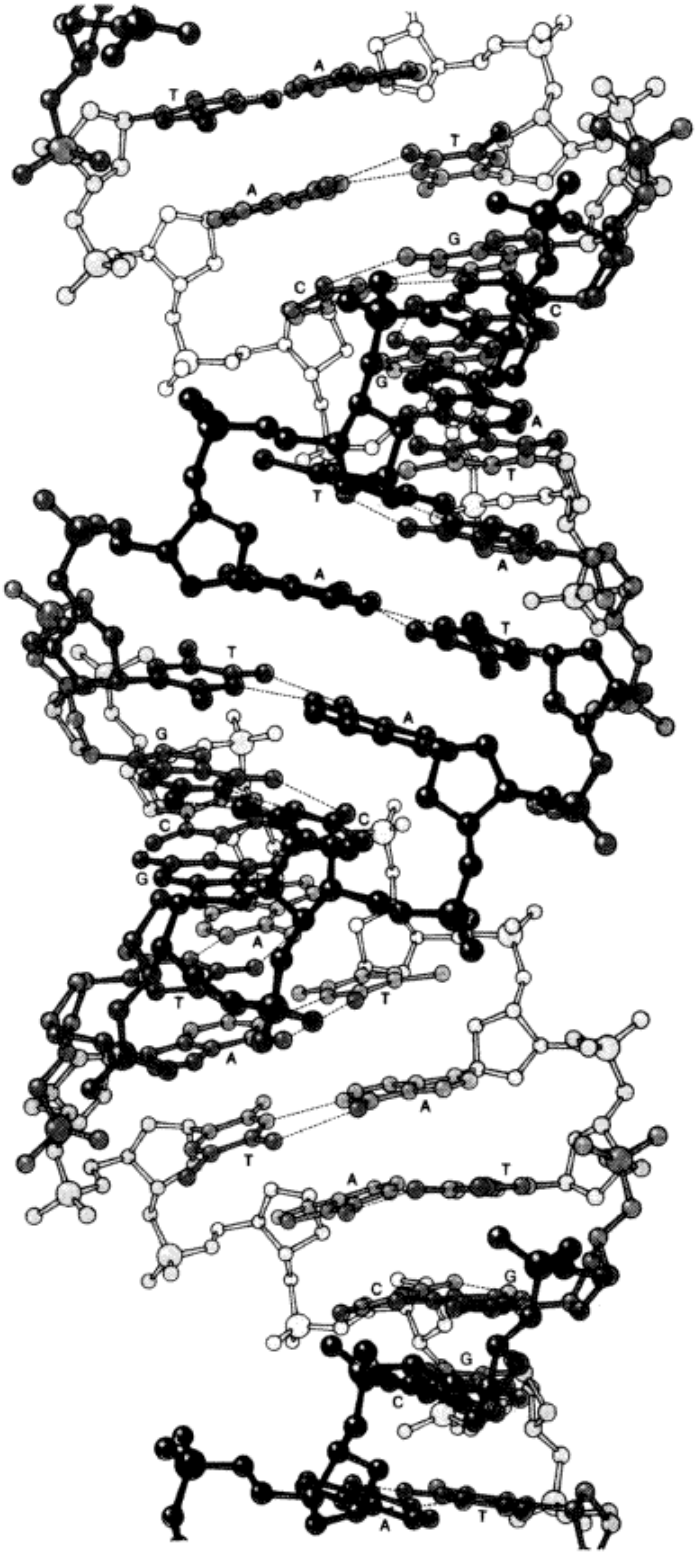

Figure 2.1: A part of a DNA molecule, featuring instances of all of the four essential basic DNA components labeled A,C,G,T. Dotted lines indicate hydrogen bonds.

of evolved structures and mechanisms against the Second Law of Thermodynamics which provides variation for free.[16]

In summary, natural evolution is the sketched process that adapts genotypic structures to their environment and keeps the represented information in sync with the dynamic habitat.

2.2.3 Structure adaptation as optimization

In its essence, natural adaptation, a gradual structure shaping guided by necessities, is equivalent to technical optimization that approaches a structure in view of an ideal. Natural evolution yields adapted genetic information as a molecule (cf. 2.2.2, p.35), a material structure, that contributes to the behavior the respective organism shows in its spatio-temporal context, i.e., its environment. The underlying concept—adaptation that influences behavior that feeds back into the shaping process—is not restricted to the carbon-based medium of life.

- Thus, a domain transfer of the gist of natural evolution may yield an instance of optimization: purpose-oriented adaptation in an artificial medium.[17]

This observation has lead to the core idea of Evolutionary Algorithms (cf. 2.2.1, p.34): implementing phylogenetic principles to the end of approximate problem solving. Predecessors were conceived in the late 1950s, with (Box 1957) possibly being the first respective contribution. EAs emerged in the early 1960s when it was still unclear whether artificial evolution in a computer would be feasible. A strongly growing body of empirical investigations and applications,[18] in particular for real-world problems, has answered this question positively.

The high industrial potential of EAs receives increasing attention. Let (Minister, Williams, Masters, Gilbert, and Haase 1995; Quagliarella, Periaux, Poloni, and Winter 1998; Zibo and Naghdy 1995; Blickle 1996b; Claus, Hopf, and Schwefel 1996; Gerdes 1996; Keller, Banzhaf, Mehnen, and Weinert 1999; Keller 2002; Busch, Ziegler, Banzhaf, Ross, Sawitzki, and Aue 2002; Costa and Schoenauer 2009) suffice as recent examples.

In summary, natural evolution manifests itself in the creation and maintenance of well-adapted living structures. Optimization may show as evolutionary adaptation of artificial structures. Thus, inspiration for EAs in general and the present work in particular comes from underlying natural structures and processes to be discussed next.

[16] Spontaneous undirected changes of molecules carrying genetic information occur as quantum phenomena (cf. (Schrödinger 1944)).

[17] The user has an objective, e.g., the design (genotype) of a mass-limited bridge (phenotype) with a maximal tether (adaptedness). This artificial structure shows its behavior in the context of forces exerted by passing vehicles and moving air masses (environment). The user may optimize the design in an experimental fashion, generating a structure sequence of increasing quality (adaptation).

[18] See, e.g., (Alander 1994; Alander 1995) and *Langdon*'s online "*Genetic Programming bibliography*" for a large collection of instances.

2.3 Natural evolution

For the present work, a discussion of the stunningly intricate phenomenon of evolution in its biological medium cannot even mention all major issues. In particular, the biochemical representation of structures and mechanisms is mostly irrelevant here, while their cybernetic meaning matters. We therefore modestly focus on a few fundamental, grand themes that are of immediate significance, neglecting, in particular, a myriad of technicalities from molecular biology.

2.3.1 Principles: heredity, variation, and selection

As seen before, for evolution to occur in a medium, three effects are necessary: diversification, conservation, and bias.

In the natural context, variation, in its procedural meaning, affects genotypic structures and results in genetic diversity, the static meaning of the emphasized term. Due to heredity, a modified copy of the complete or partial parental genotype becomes genetic information of the offspring. Reproduction elegantly combines and implements heredity and variation. Thus, an organism produces an average number of offspring per time unit that can be compared to the numbers of other individuals from the same environment, and the result is called the fitness of the organism. Therefore, fitness is a measure of relative reproductive success.[19] Since the genotype co-determines the degree of organismic adaptation (cf. 2.2, p.34), variation may lead to different parental and offspring fitness. Phenomena resulting in such difference are instances of natural selection.[20]

- The higher the parental fitness, the more offspring carry the transferred genetic information which is adapted, and the higher the chance is that the latter will go, in turn, from these to their offspring, establishing conservation.

- Modified instances of information are worse, equally well, or better adapted than corresponding parts of established genotypes.[21] Natural selection, biased for adaptation quality, yields fitness differences that induce negative feedback for the first case, and positive feedback for the last two cases, resulting in only their conservation.

[19] Thus, the organismic property of being well-adapted (cf. 2.2.2, p.35) is *not* equivalent to having a high fitness. Rather, the former is necessary for the latter, since an "out-of-sync" organism will stand little chance to reproduce often. The former, however, is not sufficient for the latter since a well-adapted life form may, in an extreme case, not reproduce, at all, for instance, for a lack of resources, such as a mate or sufficient energy.

[20] While the difference results implicitly, the emphasized term comes from the mechanistic, phenotypic point of view that "selection" discards or chooses an individual for reproduction. In view of molecular biology, genetic information is chosen for transfer.

[21] For a classic instance, a predator may spot an offspring's fur pattern more, equally, or less easily than the parental pattern.

2.3.2 Artificial metaphors

The principles from 2.3.1, p.38, serve as paragons of EA components.

Mutation and genetic variation

An undirected spontaneous random modification of genetic information, usually a rare event, is called a mutation, a term introduced by *de Vries* around 1902, then only having a phenotypic connotation. A common natural example for a mutagenic agent is ultraviolet radiation. A point mutation results in a change of only one atomic genetic component, while a macro-mutation is a single event whose effect is equivalent to the combined effect of several point mutations.

Abstracting from natural material structures and their dynamics, data types can represent elementary cybernetic issues of evolution. Thus, for EAs, a minimal mutation algorithm takes an artificial genotype g, copies it, changes the copy into a genotype $h, h \neq g$, that it returns. In this manner, the algorithm also has an implicit reproductive character, representing a metaphor of heredity. Therefore, in the present work, artificial mutation shall be of the described type.

(2.6)

Variation, reproduction, and heredity

Biology distinguishes asexual and sexual reproduction. In the former case, exactly one individual produces offspring, and, in its pure flavor, this process yields a clone: parent and child are genetically identical. For instance, a bacterium reproduces in this way. An algorithmic metaphor of clone production—an element of artificial mutation—shall be called copying.

In the latter, sexual case, two parental organisms produce an offspring whose genotype contains partial information from each parent. Thus, sexual reproduction establishes an instance of heredity and variation.[22] The intricacies of this type of biological information transfer have inspired numerous artificial metaphors that all realize information synthesis involving at least two parents.

Phenotypical behavior and selection

Natural selection on the produced offspring and future parents follows from their different efficiency in interacting with a resource-limited environment. As materially represented artificial habitats must show the same restriction, the user can easily implement a flavor of synthetic selection.

With the presented notions on artificial echos of natural evolutionary phenomena, a scheme can be assembled next that crudely captures fundamental issues of its biological paragon.

[22]For its paramount significance, we mention meiosis as the underlying biological process during which crossing over interchanges parental information. The resulting "mixed" information eventually becomes the offspring genotype which then controls the autopoiesis of the corresponding phenotype.

2.4 A typical Evolutionary Algorithm

2.4.1 Components

Preliminaries

Data types of an EA mimic features of natural evolution in an often rough manner that is due to limited computing resources available for the optimization of a given problem. Also, one usually finds many different software analogies of the same natural phenomenon, because the user incorporates problem-specific knowledge into the EA design. A data type represents a structure and its dynamics that contributes to artificial evolution, e.g., variation of a genotype. Especially, an operator of the data type performs a process on an instance of its value range, raising the required dynamics.[23]

Individual

An organism is mimicked by an artificial individual, a data type representing a point in the search space of the algorithm.[24] The individual represents exactly one artificial phenotype, i.e., data whose interpretation by the EA yields behavior. This situation is a minimal requirement on EA design, because behavior is the substrate of necessary selection.

Only if an EA gives rise to ontogeny, an individual i contains an artificial genotype that is not identical with i's phenotype. Then, a semantic mapping other than identity reads the genotype to write the phenotype. Among current EAs, a tiny minority we call developmental features flavors of ontogeny. Some of their design issues take center stage as of chapter 3, p.47.

Individual fitness

A fitness function of an EA implements an evaluation function, so that a return value indicates an individual's fitness.[25] By identification of an individual with its phenotype and, if given, genotype, one speaks of the fitness of all three entities. As a static construction, the function is a crude and explicit mimicry of a natural environment.

Fitness-based artificial selection

A fitness value is processed by an operator that implements artificial selection. At time t of a run r of an EA, the selector chooses a component from the tuple q of all

[23] A minimal example is given as bit inversion on a binary string, implementing point mutation of an artificial genotype.

[24] Example: a vector from $\mathbb{R}^3$. A corresponding individual may contain a real-valued three-dimensional array.

[25] Frequently, "fitness" is identified with "fitness value." Note also that biologists directly refer to reproductive success, while the EA community uses the term synonymously to the presented notion of quality of phenotypical behavior.

individuals that r has located and not lost again so far. q is called the population of r at t [our notation: $\boldsymbol{pop_{r,t}}$].[26]

- We decide on a constant population size, supporting a minimal EA design.

Individual interaction with an EA environment has a dual character like its natural twin. Selection for reproduction identifies an individual (parent) as argument for a variation operator that introduces a different phenotype (child) to the population, while selection for replacement removes a chosen delinquent in favor of such offspring. The decision on an individual follows a—potentially subtle—selection rule that is biased in favor of higher quality: identify i for reproduction with an average selection probability higher than that of a worse individual j. For replacement, vice versa, the rule prefers j in most cases.

There is a large set of selection schemes in use with EAs, e.g., elitist selection which we mention for later reference. A selector adhering to this philosophy guarantees that the underlying EA run r keeps a point as long as it is among $n > 0$ best points found since r's start.

Selection pressure resulting from a scheme directs artificial evolutionary search by supporting both reproductive success of better adapted individuals and failure of others.

Creation and variation operators

Regarding optimization, variation at time t introduces a candidate that represents a point in search space that the underlying EA thus visits, revisits, or already stays at, therefore gaining, regaining, or maintaining its population's genetic diversity. One tends to visualize this notion as the population moving through search space.

Typically, variation operators mimic natural mutation and crossing over. A recombinator takes at least two parents and produces at least one offspring from copies of random-chosen parental genotypic parts. This combination of parts of potential solutions may boost progress. A mutator takes one parent only and generates a different offspring individual, while chance is involved in the creation of the latter. In its basal flavor, this process represents a blind random jump in search space, possibly discovering a radically novel structure. A creator synthesizes individuals that compose the initial population[27] of an EA run. The operator "mimics" the origin of life immensely crudely, so that it is rather a symbol of the emergence of natural structures that carry information or behavior. Its creations are starting points for the evolutionary search. To that end, chance and *a priori* problem knowledge may influence their synthesis.

Creators and variators are known as search operators since they perform an exploration of space, representing innovation.

[26] We do not model multi-population flavors which do not add to the essence of ontogeny. Also note that a materially implemented, finite population covers, for a real-world problem, an utterly insignificant fraction of the involved spaces.

[27] An entity—e.g., an individual, a genotype, or phenotype—of this set is often also called initial.

An artificial point mutation results in the smallest variation of a genotype, so that one often defines the distance between genotypes g, h as minimal number of such events transforming g into h. Moving a population, a point mutation appears as an individual walking through search space. This process identifies a contiguous sequence of visited points, and we call it and each of its subsequences a trail. (2.7)

A macro-mutation of a genotype may show as an individual leaping in search space. The built sequence of visited points and each of its subsequences shall be called a route.[28] (2.8)

Since the population is finite and smaller than a practical search space, exploration may replace a current individual by a newly found one. Especially, the latter may structurally differ from the former, so that this situation represents a change of the distribution of frequencies of current genotypes. Note that, depending on the employed selection scheme, the lower the frequency of a genotypic structure is, the less likely the latter may be reproduced. Negative feedback sets in that, in the extreme case, only stops at the loss of a structure.[29] An operator performing copying (s. def.) is often used to counter such loss of structural diversity, because it mimics mutation-free asexual reproduction, strongly promoting exploitation of a genotype by amplifying the number of its current identical instances.

- Desirable self-adaptation of an EA maintains a dynamic balance of exploration and exploitation such that search progress is supported.

In summary, heredity, variation, and selection are necessary for natural evolution (cf. 2.3.1, p.38), and data types used during an EA run mimic these phenomena. Artificial evolution emerges from an EA's control flow that links these components as follows.

Control flow

Natural genetic information migrates from parents to offspring that turn into parents. Thus, iteration is an appropriate central control structure of an EA. At its beginning, a creation operator produces initial individuals to be selected for the start of this iteration. Exploration and exploitation of selected genotypes follow, yielding offspring that in turn await selection, which ends the first cycle. We call the iterative structure that carries such cycles the evolution loop since it gives rise to the essential part of genotype emergence. The—still mostly black—art of employing EA principles manifests itself in designing and initializing an algorithm such that the loop provokes and maintains the mentioned dynamic balance of innovation and conservation.[30] (2.9)

[28] Thus, a trail is a route, and, in a search space containing more than two points, there may be a route that is not a trail.

[29] Example: If all instances of the vector $(1, 2, 3)$ vanish, this structure suffers extinction.

[30] On the one hand, the loop must support exploitation of a found promising solution without losing its potential for further exploration of more attractive space regions. On the other hand, it must explore without frequently losing well-adapted structures due to weak exploitation.

Termination

Often, for practical reasons like releasing shared computing resources, an EA must not run indefinitely. To that end, the user defines the loop's termination criterion based on the fitness f of a best genotype that the run has ever found and that is represented by the best-so-far individual. For a practical optimization problem, a globally best fitness is unknown, so that a comparison of f to this optimal quality value is no option. However, the criterion may be the equality of f to the acceptance value, and it is then called the success predicate of the underlying problem (Koza 1992). It may also be equality of the amount of so-far evolved individuals to a user-determined maximal number, in which case it is known as time-out predicate. An EA run virtually always uses a variant of this condition,[31] and the logical disjunction of both predicates gives a typical termination criterion for a real-world problem.

For a generational Evolutionary Algorithm, one views the population as current generation, indexed $n \in \mathbb{N}_0$, that comprises parents. For a fixed population size s, the evolution loop produces exactly s offspring building the next generation $n + 1$ that becomes the current one. Thus, for a generational EA, a number of consecutive generations is a canonical temporal notion that we call the generational-time measure. A generation produced last in terms of this measure, and associated entities, e.g., phenotypes, shall be called final.

In extended summary, we have discussed components of a typical Evolutionary Algorithm and developmental extensions, and an instance shall be synthesized next.

2.4.2 A basal Evolutionary Algorithm

Structure

Informal pseudo code and its comment in ISO-C style shall suffice to summarize the control flow through the described data types.

---

[31] The Master's thesis by one of the author's former students explores boosting GP's implicitness by a metaphor of an individual's limited initial energy that is to render obsolete a time-out predicate (Remco Nabuurs, Energy-Bound Genetic Programming, University of Leiden, The Netherlands, 2004).

```
creation /* of initial population */

IF algorithm is developmental

  genotype-phenotype mapping /* yields feasible solutions */

quality evaluation /* interprets solutions, evoking behavior */

WHILE termination criterion not met /* (re)enter evolution loop */

  selection /* for reproduction of better behavior */

  cloning XOR reproductive variation /* of chosen parents */
                                     /* produces next generation */
  IF algorithm is developmental

    genotype-phenotype mapping

  quality evaluation

ELIHW /* leave loop */
```

For transparency, we have stretched out the design of the presented pseudo-algorithm, and an equivalent, elegant version follows.

```
REPEAT

  IF first loop entry

    creation

  ELSE

    selection

    cloning XOR reproductive variation  ESLE

  IF algorithm is developmental

    genotype-phenotype mapping

  quality evaluation

UNTIL termination criterion met
```

Function

As a flavor of artificial evolution, the presented EA template describes an instance of an iterative search process of gradual improvement of target structures. Purely volume-oriented approaches, e.g., blind random search, are at one end of the scale of such approximation schemes.

At the other end, exclusively path-oriented algorithms, such as gradient-based procedures and zigzag strategies, often only find suboptimal solutions for most practical tasks that are strongly irregular, i.e., multi-modal,[32] non-convex, and non-differentiable. In the middle of the spectrum, hybrid optimizers such as EAs yield a practical compromise.

There are theoretical results on the functionality of Evolutionary Algorithms, like (Beyer 1995; Bäck 1996; Rudolph 1996; Motoki 2002; Schwefel, Wegener, and Weinert 2003) to name few, and (Bäck, Fogel, and Michalewicz 1997) gives further references. Due to a large number of complex EAs for practical problems, a unifying theory does not yet exist. However, individual results support intuitive design of effective real-world heuristics as they foster a precise understanding of basic mechanisms. While theory may keep lagging behind rapidly emerging practical EAs,[33] complete formal models of simple older variants are also useful to foster acceptance with skeptics from classic fields of optimization. This may be especially in demand for the flavor known as Genetic Programming (GP).

2.5 Genetic Programming

- We characterize a GP algorithm (Koza 1992; Banzhaf, Nordin, Keller, and Francone 1998) as an EA that interprets a point in search space as an algorithm whose computation influences individual quality.

While this description captures GP's original spirit, the field has quickly embraced a wider view that includes arbitrary phenotypic structures, e.g., (Koza, Mydlowec, Lanza, Yu, and Keane 2001; Bennett III, Koza, Yu, and Mydlowec 2000; Cao, Kang, Chen, and Yu 2000; Keller, Banzhaf, Mehnen, and Weinert 1999).

The user may represent knowledge and assumptions on a problem as

- the function set, containing symbols of algorithms,[34] and as
- the terminal set that symbolizes variables and constants.

An underlying GP system synthesizes algorithm representations from the provided elements.

Often, quality evaluation interprets a user-given set of fitness cases as a problem at hand: each case is a pair (input i, output o) that defines o desired from an individual that has taken i as argument. For all cases, the evaluation phase computes the *individual total error* on the output, yielding a quality value for feedback, so that, in terms of Machine Learning (Mitchell 1997), the cases represent a training set.

We can approach the technical objective next.

[32] Featuring several optima.

[33] One reason is the chronic delay from developing appropriate mathematical tools.

[34] Example: '+' for arithmetic addition, '`myFnc`' for a user-defined routine.

Chapter 3

Developmental Genetic Programming

Everything flows. HERACLEITUS

3.1 Introduction

To the end of empirical research (cf. 1.2.7, p.24), a system is required whose core procedure shall be called the projected search algorithm. Initialized with different sets of parameters, it represents different GP approaches to be investigated.

The technical objective suggests adding an ontogenic metaphor as an initial step of addressing the inherent EA dependence on strong human control that results from many explicit side constraints. To that end, the discussion shall first consider the feasibility of such metaphors for a common GP approach.

The design of problem-unspecific algorithms is required for the present work, accompanied by some domain-unspecific tasks that serve as carrier for a feasibility discussion of goal-oriented autopoiesis.[1] In the long term, we hope for a transition of user attitude from manual problem identification and application of single-purpose solvers to allowing an artificial system to gain insight into a domain that it thereby changes such that "problems"—we see them as consequences of uninformed domain manipulation—do not occur in the first place. A prime instance of an informed natural system is a developing organism: it does not have to stop for analysis of a glitch it must never hit because *continuous* change is necessary for the process of life.

[1] Optimizing specific real-world problems is the object of many EA contributions, and, in particular, GP's ability to this end has been sufficiently demonstrated. For few of many instances, we mention (Aiyarak, Saket, and Sinclair 1997; Andrews and Prager 1994; Baydar and Saitou 2000; Keller, Banzhaf, Mehnen, and Weinert 1999; Koza, Keane, Bennett, Yu, Mydlowec, and Stiffelman 1999; Langdon and Barrett 2004).

3.2 Algorithmic metaphors of development

3.2.1 Preliminaries

Facts and theories from molecular biology suggest the concept of distinct search and solution spaces, an understanding first emphasized and empirically demonstrated for GP in (Banzhaf 1993). As an initial approach to the technical objective, we are to theoretically analyze and elaborate on relations between spaces, using our view on EAs from chapter 2 to the end of identifying degrees of freedom for designing the projected search algorithm. Given a semantic mapping, a point in decision space represents a potential solution of the underlying problem. If the mapping is identity, we call it trivial, because it has the decision space equal the potential-solution space. It follows that the search space is a subset of the potential-solution space, and, especially, the representative space is contained in the solution space. This particular case describes the current situation for most GP algorithms:

- search must be strictly controlled as it operates in a structurally constrained space.
- Otherwise, a developmental process as mandatory component of search uses a non-trivial semantic mapping to project a located representative onto only one feasible solution.

The concept of a semantic mapping abstracts from structural representations of its domain and image elements. For current applied work, however, one must construct different, concrete such representations. Furthermore, the theoretical framework we start building here is to allow for an easy transformation of its key entities into parts of the projected search algorithm.[2]

An alphabet is a finite set of symbols, as mentioned, e.g., in (Hopcroft and Ullman 1979). A structural representation of an entity f shall be given by an encoding E of f [$\boldsymbol{enc_{Ef}}$] as follows.[3] The elements from the non-empty encoding alphabet of enc_{Ef} compose the representation. The encoding structure of enc_{Ef} shall be a digraph

$$g = (N, A),$$

i.e., N is a finite set of so-called nodes, and $A \subset N \times N$.[4] An element of A is known as directed edge or arc, and a digraph may be visualized as circles (nodes) that are possibly connected by arrows (arcs). Note the special case $(\{\}, \{\})$ that we call the trivial graph. Also, with $M \subset N$, a digraph $f = (M, (M \times M) \cap A)$ is known as subgraph of g.

A structural component $m \in N$ is to represent exactly one symbol from the encoding alphabet $\mathbb{E}$, which we model as $n : N \to \mathbb{E}$, the node function of enc_{Ef}. $n(m)$ shall be called the value of m.

[2]The frame is also an entry to mathematical modeling of developmental-search dynamics, which we see outside the scope of the present work.

[3]We may introduce notation ν as $[\boldsymbol{\nu}]$.

[4]For an introduction to graph theory with an emphasis on informatics, see, e.g., (Wood 1987).

Eventually, we can define an encoding of f as

$$enc_{Ef} = (\mathbb{E}, dom(n), A, n),$$

and $|dom(n)|$ shall be called its encoding size.

Ultimately, we may capture a "structural representation" of f as follows. $img(n)$ shall be known as minimal alphabet of enc_{Ef} since it contains only those $a \in \mathbb{E}$ that are values of nodes of the encoding. Thus, for given (A, n) and arbitrary $\mathbb{E}$, the set of all encodings of f is determined as

$$\boldsymbol{enc_{If}} = \{(\mathbb{E}, dom(\mathsf{n}), \mathsf{A}, \mathsf{n}) \mid img(\mathsf{n}) \subset \mathbb{E}\},$$

and we call

$$(img(n), dom(n), A, n)$$

the minimal encoding of f over $t = (A, n)$ that is the structural representation of f $[\boldsymbol{rep_{t_f}}]$. Therefore, t may be envisioned as a network such that each node is marked with exactly one symbol from $img(n)$. As t determines the minimal encoding, we can apply the context of the latter to the former, e.g., t's encoding alphabet shall be the encoding alphabet of the minimal encoding.

With the abstract notion of a structural representation t at hand, an entity can show in different ways. Especially, the concept of the node function of t allows for changing t by defining the function sufficiently flexible. This property of t is relevant to designing a semantic mapping, because the latter represents the change of a structural representation of a decision point into the structural representation of a potential solution.

Next, we express the resource-efficient concept of a "string" in terms of an encoding. Let

$$N = \{n_0, .., n_k\}, k \geq 0,$$

be the node set of a digraph $g = (N, A)$ with

$$A = \{(n_i, n_{i+1}) \mid 0 \leq i < |N|\},$$

then g is called a path. Note $k = 0$ that implies $N = \{n_0\}, A = \emptyset$, which is a path that shows as single circle with no arcs, modeling an atomic component. Let $g = (\{0, .., k\}, A), k \geq 0$, be a path, then we call g a sequence. A sequence $s = (N, A)$ is determined by N because $|N|$ fixes A; thus, s shall be denoted by $\boldsymbol{(N)}$.

(3.1)

Let

$$enc_{Ef} = (\mathbb{E}, dom(n), A, n), N = dom(n)$$

with sequence (N), then we name E a string encoding of f. E is determined by $(\mathbb{E}, n)$ because

i) A is determined by $|N|$, and

ii) $N = dom(n)$.

Thus, we write a string encoding enc_{Ef} as $(\mathbb{E}, \boldsymbol{n})$ which determines the tuple

$$\sigma = (n(j)_i), j \in N$$

that shall be called a string over $\mathbb{E}$ with size $|\boldsymbol{\sigma}| := |N|$. Thus, $|\sigma|$ is the encoding size of enc_{Ef}. With $t = (A, n)$, consider the structural representation rep_{t_f} that we name f's string representation. n determines $dom(n)$ which fixes A; therefore, rep_{t_f} is given by n, and we denote the former by $\boldsymbol{rep_{n_f}}$ which can be identified with σ $[\boldsymbol{rep_{n_f}} = \sigma]$.

Let string encoding $enc_{Ef} = (\mathbb{E}, n)$ with minimal alphabet $\boldsymbol{min_\alpha} = img(n) \subset \mathbb{E}$. If all $a \in min_\alpha$ are from the same class x, e.g., "letter," then we call $\sigma = rep_{n_f}$ an $\boldsymbol{x}$ string,[5] and with $|\sigma| = k$, we get a $\boldsymbol{k}$-$\boldsymbol{x}$ string. For instance, a 3-letter string is given by

$$enc_{Ef} = (\{\text{'a','b','c','d'}\}, \{(0,\text{'b'}),(1,\text{'a'}),(2,\text{'d'})\})^6$$

with $min_\alpha = \{\text{'a','b','d'}\}$, $rep_{n_f} = (\text{'b','a','d'})$. We shall also write σ in its string notation, a corresponding quoted symbol series—e.g., $s =$ "bad"—, and both $\sigma = rep_{n_f}$ and s can be identified $[\boldsymbol{rep_{n_f}} = s]$. σ's string value $[\boldsymbol{\$\sigma}]$—its "contents"—can be expressed by s, e.g., $\$s =$ "foobar".

For given $\sigma = rep_{n_f}$, we call $i \in dom(n)$ a position in σ, and i is also a position of $a = n(i) \in \mathbb{E}$, a building σ.[7] Regarding σ, $dom(n)$ shall be known as its position set $[\boldsymbol{pos_\sigma}]$. We define $\boldsymbol{\sigma(i)} = n(i)$, while $\boldsymbol{\sigma_i}$ shall denote σ's i-th structural component.[8]

A node function n determines string representation $\sigma = rep_{n_f}$. Thus, an appropriate data type element representing σ can be synthesized—e.g., sequentially—from n: a composer *for* n and *of* σ computes $n(i)$ for all $i \in dom(n) = pos_\sigma$ and stores the values in the corresponding element components.

- Thus, n represents σ's sequence information $[\boldsymbol{seq_\sigma}]$.

For node set $N = \emptyset$ of a string encoding, the corresponding trivial graph

$$\sigma = \text{“”}, |\sigma| = 0,$$

results that we call the trivial string, known as the empty word and often denoted by $\boldsymbol{\epsilon}$.

A string τ shall be known as a substring of σ if and only if

$$(\tau = \epsilon) \vee \exists q : \tau_0 = \sigma_q \wedge q + |\tau| \leq |\sigma|.$$

[5] While the letter class allows for a simple writing of subsequent examples, recall that the introduced notion of an alphabet is abstract: it may contain arbitrary entities, supporting our strategic perspective.

[6] For ease of notation, note that $\mathbb{N} \ni i = n_i \in N$, because, for a string encoding, N is the node set of a sequence.

[7] Example: 2 is a position in $\sigma =$"aba", and 0 and 2 are positions of 'a'.

[8] Example: for $\sigma =$ "test", $\sigma(1) =$ 'e' and $\sigma(0) = \sigma(3) =$ 't', but $\sigma_0 \neq \sigma_3$.

q shall be named starting position of τ, and the latter shall be identified with $\boldsymbol{\sigma_{q,|\tau|}}$.[9]

Summarizing, the usual, informal notion of a string—a "finite sequence of symbols from an alphabet", e.g., (Hopcroft and Ullman 1979; Wood 1987)—shows as special case of the presented encoding concept given by directed graphs (digraph)[10] and functions. This view of encoding allows flexible descriptions of producing and transforming general structural representations, which is of interest regarding semantic mappings which operate between such representations.

The concept of a *set* of structures matters regarding an EA population. Thus, for the design of the projected search algorithm, an encoding of such a set is of interest next.

3.2.2 Encoding of a set and space

A digraph $g = (\{n_0, n_1, n_2, n_3\}, \{(n_0, n_1), (n_1, n_2), (n_1, n_3)\})$ shall be called a fork, and we name $g = (\{n_0, .., n_k\}, \{(n_0, n_1), (n_1, n_2), .., (n_{k-1}, n_k), (n_k, n_0)\})$, $k \geq 0$, a cycle. Note a cycle $c = (\{n\}, \{(n, n)\})$, i.e., c's single node connects to itself, that is known as a loop graph. We call a graph containing such structures forked and cyclic, and, regarding a path, path-containing. These subgraphs give rise to six disjoint classes whose union equals the set of all graphs:

- The first class contains only the trivial graph which is neither forked nor cyclic nor path-containing.
- Second class: graphs that are path-containing and neither cyclic nor forked, and we call it path-containing, denoted by ¶ (pronounce "path").
- Third class: graphs that are cyclic and neither path-containing nor forked, with the loop graph as only representative.
- Fourth class: graphs that are cyclic and path-containing and not forked, and it shall be called circular.
- Fifth class: graphs that are forked and not cyclic, denoted by FNC.
- Eventually, the sixth class: graphs that are forked and cyclic.

Let the class of a graph g be denoted by $\boldsymbol{clss(g)}$, so that, e.g., for a sequence g, $clss(g) = ¶$. We consider two graphs identical if they are in the same class. Two encodings shall be known as identical if their encoding structures—which are graphs—are identical and if their alphabets are equal.

[9] Example: For $\sigma =$ "abcd", $\sigma_{1,2} =$ "bc".
To illustrate different strings having the same string value, let $\sigma =$ "hownow", $\rho =$ "browncow". $\alpha = \sigma_{1,2}$, $\beta = \sigma_{4,2}$, and $\gamma = \rho_{2,2}$ have identical values $\$\alpha = \$\beta = \$\gamma =$ "ow".

[10] We shall also use "graph" in place of "digraph."

Let E denote a finite set, and, for each $e \in E$, let there be one individual encoding

$$enc_{ie} = (\mathbb{E}_e, N_e, A_e, n_e).$$

The resulting set S of individual encodings shall be called an encoding set of E, and we can identify an $e \in E$ with its $enc_{ie} \in S$. If all encodings in S are mutually identical, then $(\mathbb{E}_e, clss(N_e, A_e))$ are equal for all $e \in E$. Thus, for $\mathbb{E} := \mathbb{E}_e$ and $c := clss(N_e, A_e)$, $(\mathbb{E}, c)$ determines the encoding alphabet and the graph class of the encoding structure of each element in S. Then, we name S a homogeneous encoding of E [$\boldsymbol{enc_{SE}}$] with encoding structure c and encoding alphabet $\mathbb{E}$.

- If the structure is the path-containing or circular class, then the encoding shall be called linear. If all $s \in enc_{SE}$ have equal encoding size m, then we call m the encoding size of enc_{SE}, and the latter shall be named fixed-size.

Regarding a given problem, we assume that a homogeneous encoding enc_{SE} is supplied for each set E of interest, such as the search space of an employed solver. While this restriction simplifies design of the projected search algorithm, it still allows for dealing with real-world problems. For instance, for a problem p, consider given enc_{Spot_p}, an encoding of its potential-solution space. Then, $enc_{Tsol_p} = \{enc_{iq} \in S \mid q \in sol_p\}$ is an encoding of $sol_p \subset pot_p$. (3.2)

For a set E, consider a homogeneous encoding

$$enc_{SE} = \{(\mathbb{E}, N_e, A_e, n_e) \mid e \in E\}$$

which determines the structural representations $rep_{(A_e,n_e)_e}$ of each $e \in E$. We call $\{rep_{(A_e,n_e)_e} \mid e \in E\}$ the representing set of E over S.

- Especially, dec_p and pot_p shall be identified with respective representing sets that matter to the projected search algorithm during its computing of a semantic mapping.

In summary, we have given an encoding concept for a set. For a problem, properties of involved spaces and their implications for semantic mappings can be discussed on this background next.

Some characteristics of an encoding reflect properties of a material, digital computing environment. For instance, finite memory is honored by finite size and alphabet of an encoding. Thus, a representing set of decision space dec_p and of potential-solution space pot_p, respectively, is also finite, while, theoretically, the spaces may contain infinitely many points.

- Thus, there is a finite number of corresponding semantic mappings, and, as $sol_p \subset pot_p$, a solution space is finite.

Depending on the representing sets of dec_p and pot_p, $dec_p \cap pot_p = \emptyset$ does or does not hold, so that we discuss both cases next.

3.2.3 Intersecting decision and potential-solution space

Let $dec_p \cap pot_p \neq \emptyset$, then there are three mutually exclusive cases.

Firstly, let $dec_p \subset pot_p$. In this case only, $\boldsymbol{s_p} \in S_p$ can be trivial, while, for a real-world problem, $n = |dec_p| \ggg 1$ and $|pot_p| \ggg 1$ imply a large number of semantic mappings, including $n! - 1$ non-trivial bijections alone from the decision space into itself.

Secondly, let $pot_p \subsetneq dec_p$ hold, then s_p is non-bijective, which implies that it is not trivial.

Thirdly, $pot_p \neq dec_p \wedge pot_p \not\subseteq dec_p$ remains, so that there is a non-empty set

$$S = \{d \in dec_p = dom(s_p) \mid d \notin pot_p = rng(s_p)\},$$

which implies $s_p(d) \neq d$ for all $d \in S$, so that s_p is not trivial.

In summary, given intersecting dec_p and pot_p, for each real-world problem and "almost all" problems, there exist many non-trivial semantic mappings, respectively. This situation matters to the design of the projected search algorithm, since we must decide on a mapping and suggest a subalgorithm that computes it.

Next, the discussion continues for $dec_p \cap pot_p = \emptyset$.

3.2.4 Disjoint decision and potential-solution space

Regarding dec_p and pot_p, respective disjoint encoding alphabets or different graph classes render these spaces disjoint. Also, different encoding sizes may have this effect.

- Given disjoint spaces, an $m \in S_p$ is always non-trivial. We focus on this case, at least for the immediate reason that it gives independence from any problem-specific phenotypic representation of pot_p.

Some $m \in S_p$ may mutually differ in how far they are beneficial, i.e., supporting the performance of a search algorithm alg_{a_m}. For an extreme instance, consider $m(dec_p) = o$ with global optimum o of p, while $n \in S_p$, $n \neq m$, maps the space onto a global "pessimum."

In summary, for problem p with large pot_p and $dec_p \cap pot_p = \emptyset$, $id_{dec_p} \notin S_p$ and $|S_p|$ is big. This situation poses a task for the design of the projected search algorithm:

- Identify a subalgorithm that, given p, computes a mapping $m \in S_p$ that is beneficial to the resulting search algorithm alg_{a_m}.

We approach this object next by fusing principles of EA design with the current discussion.

3.3 A genotype-phenotype mapping: an algorithmic metaphor of development

We consider non-trivial $m \in S_p$ for an algorithm alg_{a_m}. Thus, $r \in rep_a$ represents information to a subalgorithm that computes $s = m(r) \in sol_p$. Such a developmental algorithm $\mathbf{dev}_{\rightarrow}$ is a part of the projected search algorithm $\mathbf{p}_{\rightarrow}$.

(3.3)

$r \in fea_m \subset dec_p$—a decision with feasible semantics in terms of the mapping—shall be called a genotype of its phenotype s under m, which reflects that a biological phenotype is a living structure that interacts with its spatio-temporal context. Likewise, under $m \in S_p$, $s \in sol_p$ represents interaction with its problem environment.

- Note that, while we are focusing on EAs, the concept of a developmental algorithm is independent from this class of search procedures. Hence, on the one hand, an arbitrary search algorithm that uses this concept shall also be called developmental. Thus, $m \neq id_{dec_p}$ holds for such alg_{a_m}, because the corresponding developmental algorithm $\text{dev}_{\rightarrow}$ represents m, a non-trivial semantic mapping (cf. 3.3, p.54).

- On the other hand, a search algorithm alg_{a_m} for problem p shall be called common if and only if $m = id_{dec_p}$ and $rep_a = sea_a$. From 2.4, p.31, the following statements and their implications result for this algorithm.

(3.4)

- $dec_p \subset pot_p$
- $sol_p \subset pot_p$
- $inf_p \subset pot_p$
- $inf_p \cap sol_p = \emptyset$
- $inf_p \cup sol_p = pot_p$
- $rep_a = sea_a \subset fea_m \subset (dec_p \cap sol_p) \subset pot_p$, which implies
- $rep_a \subset sol_p$, which implies
- $rep_a \cap inf_p = \emptyset$

Thus, for a common search algorithm, the search space is a subset of the solution space, and, especially, a genotype and its phenotype are identical. This scenario is the simplest conceivable setup of a search algorithm: each search point is feasible. Especially, the concept of a semantic representation of an entity is insignificant to the process, because $m \in S_p$ is trivial. Therefore, the algorithm has no freedom of interpreting—i.e., assigning meaning to—a located point.

- Thus, a common approach cannot adapt its understanding of a search point to the current properties of the underlying problem.

Almost all GP algorithms belong to this class for some reasons to follow. Firstly, this field has been strongly influenced by (Koza 1992) which propagates common approaches. This publication made the name of the field known and led to a massive rise of research on artificial evolution of algorithms. Secondly, as most GP-related publications show, common search usually shows the desired behavior: for a given problem, it often delivers a rising progression of the average fitness of the population. Therefore, no immediate need for conceptually different search algorithms is felt. Thirdly, designing and implementing a developmental algorithm represents a substantial additional effort.

Fourthly, from a perspective that ignores ontogeny, it seems, for each problem, blatantly detrimental to have a search algorithm waste expensive resources on locating and transforming possibly infeasible search points while the user is only interested in feasible solutions. Indeed, a developmental search algorithm alg_{a_m} is to return a feasible structure of interest, and, to this end, a must obey

$$S := m(sea_a) \cap sol_p \neq \emptyset.$$

$$m^{-1}(S) = sea_a \cap fea_m = rep_a,$$

the representative space of the algorithm, follows. Restricting m to this space results in a function $f := m|_{rep_a}$ with $dom(f) = rep_a$ (cf. A.1, p.205),

$$rng(f) = sol_p \subset rng(m) = pot_p,$$

$$img(f) \subset sol_p \text{ (cf. 2.4, p.31).}$$

Thus, $f : rep_a \to sol_p$ maps genotypes under m onto phenotypes for p. Therefore, f shall be called the genotype-phenotype mapping of a which determines m and rep_a which determine f [$\boldsymbol{gp_a}$].

In summary, we have formally introduced a general concept of the genotype-phenotype mapping (GPM) of a developmental search algorithm for arbitrary structures.[11] (3.5)

Next, a specific, *empirical GPM* must be identified for the ontogenic version of the projected search algorithm $p_{\to}$. As entry point, we consider the first-ever instance of a developmental GP algorithm, suggested in (Banzhaf 1993). The feasibility and effectiveness of the proposed search procedure is demonstrated on a problem, and it is hypothesized that a developmental stage can enhance search performance of EAs.

[11] Here, a GPM is a crude metaphor of biological ontogeny of an individual. Development is a continuous phenomenon during organismic existence, while a GPM is a singular, initial event that yields an artificial phenotype. Furthermore, identical instances of a natural genotype do not yield identical phenotypes, because their ontogeny also depends on their habitats. Thus, biological development represents a mapping that is not a function. Such differences between the smooth network of natural structure-building processes and the comparatively bumpy concepts of a GPM and other artificial mechanisms will ultimately only dissolve in autopoietic programming.

An empirical investigation of this hypothesis shall carry the design of $p_{\rightarrow}$ from here, and, to this end, a comparison of certain search procedures for a real-world problem will serve. (3.6)

These *empirical search algorithms* shall be named

- common and
- developmental, respectively,

and they are instances of $p_{\rightarrow}$ (cf. 3.1, p.47).[12] A motivation for the *manual* design of $dev_{\rightarrow}$, the developmental algorithm that gives rise to the developmental instance of $p_{\rightarrow}$, follows.

For problem p with large potential-solution space, $|S_p|$ is big (cf. 3.2.4, p.53). For p, let there be a search algorithm alg_{a_m}, an acceptance value q, and an evaluation function o. For $t = (p, o, q)$, we define problem d that asks for an $m \in dec_d := S_p$ with $m(sea_a) \subset acc_t$. Thus, facing d, the user is interested in a semantic mapping of p that projects a's entire search space onto at least one acceptable solution. In this ideal situation, the creation of the first individual of the initial population represents such a solution, yielding great search performance. However, this scenario is unrealistic for a real-world problem p for that, by its definition, $s \in acc_t$ is unknown, so that a solution to d is unknown. Thus, solving d implies solving p.

This complex is an instance of the conflict of recursion (cf. 1.2.8, p.26). Its resolution requires a complete algorithmic metaphor of organic evolution which "simultaneously adapts everything." An according EA would adapt all of its components, not merely its population containing structures of interest. Here, the component in question would be $dev_{\rightarrow}$ which computes the semantic mapping—and, therefore, the GPM—of $p_{\rightarrow}$. Thus, $dev_{\rightarrow}$, the empirical developmental algorithm, must be designed manually next.

3.4 The empirical developmental algorithm: an instance of a genotype-phenotype mapping

A starting point toward a developmental algorithm is Kimura's "neutral theory[13] of molecular evolution" (Kimura 1968; Kimura 1983) which is supported by empirical results (Mukai 1985). The theory postulates that organic evolution on the molecular level is essentially driven by mutations that are almost or utterly neutral, i.e., insignificant to natural selection, which effects random genetic drift. One can see "neutral evolution" and the classic, Darwinian flavor which is selection-centered as superimposed phenomena that add up to the evolution of species. This framework

[12] Therefore, here, the description of SOLUTION (cf. 1.2.7, p.24)—which implements $p_{\rightarrow}$—commences. As our focus is on investigating behavior of empirical search algorithms, optimizing a special test problem only matters as means to this end, so that no acceptance value is needed for such a problem. Thus, SOLUTION will, in principle, represent an open-ended process.

[13] We shall prefer "neutrality theory."

includes the concept of neutral variants, i.e., different genotypes that yield similar phenotypes,[14] that one can readily express as algorithmic metaphor. While such artificial neutrality only recently gained some attention of the GP community, it has been suggested at an early stage (Banzhaf 1994). Although neutral variation involves detrimental potential, such as supporting introns[15] (e.g., (Brameier and Banzhaf 2003)), it can foster performance (e.g., (Miller and Thomson 2000; Yu and Miller 2002)). Regarding organic evolution, a neutral mutation derives one of a given phenotype's neutral variants from another such one, opposed to an adaptive mutation that results in a fitness change. The former is, according to the neutrality theory, a major reason for the high genetic diversity observed in natural populations.

To the end of assimilating neutrality into our framework, we catch the previously introduced notion of an EA individual in formal terms next.

3.4.1 An individual: a genotype and a phenotype

Let alg_{a_m} be a search algorithm for problem p with $m \in S_p$ (standard assumption). We denote $\{\ (d, m(d)) \mid d \in fea_m\}$ by $\boldsymbol{ind_m}$, so that an individual $i \in ind_m$ represents its genotype d and phenotype $m(d)$. Let r be a run of a, and let t denote a point in run time. By definition, r's population is the tuple of $n > 0$ individuals that are available to r at t. Thus,

$$pop_{r,t} = (j_0, .., j_k) \text{ with } j_0, .., j_k \in ind_m.$$

The identity of $j = (d, m(d))_i \in pop_{r,t}$ over time is given by genotype d and $0 \leq i \leq k$, because $|m(d)| = 1$. Thus, if $j' = (e, m(d))_i \in pop_{r,t+1}$ with $d = e$, then j and j' are identical [$\boldsymbol{j = j'}$].

$j \in ind_m$ shall be known as one of r's individuals that is real at time t if and only if $j \in pop_{r,t}$, else it is unreal at t.

For r, we name a genotype d real at t if and only if there is an individual $j = (d, m(d))$ that is real at t, else d is unreal. Let this terminology hold ditto for phenotype $m(d)$.

If $i = (d, m(d)) \in ind_m$, then i and its genotype d shall be called singular at t if and only if i is the only real individual of r with genotype d at t.

Let $i_k = (d, m(d)) \in pop_{r,t}$, $j_l = (e, m(d)) \in pop_{r,s}$. If and only if $d = e$, we call i_k a double of j_l. In particular, $(d, m(d))_i, (d, m(d))_j \in pop_{r,t}$ with $i \neq j$ may hold, i.e., doubles may represent the same genotype—and phenotype—at the same time. As an extreme, at t, only one genotype g may be real, i.e., each current individual of the population represents g. As opposite scenario, there are $|pop_{r,t}|$ singular genotypes.

For a run r and time t, let $s = (r, t)$, and consider

$$\boldsymbol{sng_s} := \{\ i \in pop_s \mid i \text{ singular for } r \text{ at } t\}.$$

[14] As selection is fitness-based, it behaves in an unbiased manner regarding "neutral" variants, hence the term.

[15] Genetic information that does not influence phenotypic quality.

We define the genetic diversity of pop_s as

$$\boldsymbol{gd_s} := \frac{|sng_s|}{|pop_s|}.$$

Usually, $|pop_s| > 1$, so that, for the aforementioned two extremes, $gd_s = 0$ and $gd_s = 1$ result, respectively, with other gd_s values in $[0, 1]$.

- For the remainder of the present work, $|pop_{r,t}| > 1$ shall be fixed for all t, which is advantageous to the later synthesis of the projected search algorithm, as we will detail then.

The presented concepts of an individual and its encompassing population gives rise to a discussion of neutral mutations of the former and genetic diversity of the latter next.

3.4.2 Neutral genotypic mutation and genetic diversity

Consider alg_{a_m} with $m \in S_p$ for problem p. Mutation shall be represented by a function

$$\boldsymbol{mut} : dec_p \to dec_p$$

with $mut(d) \neq d$ for all $d \in dec_p$ (cf. 2.6, p.39).[16]

For $m(d) = m(mut(d))$, d and $mut(d)$ shall be known as neutral variants of $m(d)$. We call the mutation of d to $mut(d)$ neutral under m.

Let r be a run of a. A neutral mutation regarding individual

$$\dot{g} := (d, m(d))_\iota \in P := pop_{r,t}$$

may result in $gd_{r,t+1} > gd_{r,t}$.

- **Proof** For instance, let

  $$\dot{k} := (d, m(d))_\kappa \in P, \iota \neq \kappa,$$

  which implies that $\dot{k}$ is a double of $\dot{g}$. Let a neutral mutation regarding $\dot{g}$ yield

  $$\dot{l} := (e, m(d))_\kappa \in Q := pop_{r,t+1}, e \neq d,$$

  and let $(e, m(d)) \notin P$. Thus, $\dot{l}$ replaces $\dot{k}$ that is a double of $\dot{g}$, which implies that $\dot{l}$ does not replace an individual which is singular for r at t, while $\dot{l}$ is singular for r at $t + 1$. Therefore, $|sng_{r,t}| < |sng_{r,t+1}|$, and $gd_{r,t} < gd_{r,t+1}$ follows, because $|P| = |Q|$. □

(3.7)

In summary, a neutral mutation—that can only occur for a developmental search algorithm alg_{a_m}—may increase genetic diversity of a's population over run-time.

(3.8)

[16] Thus, mut is merely undefined for the pathological case $|dec_p| = 1$ which will not be considered as it is irrelevant to problem solving.

3.4.3 Genetic diversity and performance

Let the standard assumption hold and consider run r of given alg_{a_m}. As an extreme, each real individual at time t represents the same phenotype q, which we name maximal phenotypic convergence in q. With both a developmental and common run, this scenario may occur once r finds a very good phenotype $\mathsf{q} := m(d)$. Then, for instance, selection favors d for copying, so that the number of real d-representing individuals rises. d and q are said to be taking over the population which is getting trapped in q. (3.9)

Thus, without a countering process, at time $s > t$, maximal phenotypic convergence in q rules. Otherwise, the trapped population may escape, e.g., because r finds a better q'. If d is not acceptable, the trapping process is known as premature convergence of the population. (3.10)

For a common search algorithm, this phenomenon is necessarily reflected by a sinking genetic diversity, because the number of singular genotypes decreases. For a developmental search algorithm, however, convergence—premature or not—does not necessarily mean sinking genetic diversity, because $m(d)$ may be the phenotype of real neutral variants. (3.11)

Especially, in presence of development, neutral mutations may increase and possibly maximize genetic diversity (cf. 3.7, p.58) during run-time interval $T = [t, u]$ with maximal phenotypic convergence in some $m(d)$. However:

For a common run r, such convergence in $m(d)$ during T is equivalent to minimal genetic diversity—

- **Proof** Fixed population size $|pop_{r,t}| > 1$ holds for t in T (cf. 3.4.1, p.58);
  if each individual that is real at t represents $m(d)$,
  then d is the only real genotype at t, since $m = id_{dec_p}$.
  Thus, $|sng_{r,t}| = 0$, equivalent to $gd_{r,t} = 0$, for all t in T. □

—while a permanently high genetic diversity is desirable: it supports exploration as it confronts a search process with potentially different subenvironments. (3.12)

This property is usually critical to performance on a real-world problem as its acceptable search points are mostly scattered over space. Therefore, there is a number of contributions, e.g., (Brameier and Banzhaf 2002; McPhee and Hopper 1999; Rosca 1995; Keller and Banzhaf 1994), attempting to avoid diversity loss—well-known for common approaches—by use of additional *explicit*, manually designed procedures. However, a neutral mutation increases and maintains diversity *implicitly*,[17] supporting our strategic objective (cf. 1.5, p.14). (3.13)

[17] Neutrality emerges from properties of both the underlying genotypic representation and GPM.

For a phenotype q, one can imagine such mutations having search drift through space along lines of q's so-called neutral network that contains q's neutral variants. In GP, this concept has received attention on a practical and theoretical level (Yu and Miller 2001; Miller and Thomson 2000; Langdon and Poli 1998; Keller and Banzhaf 1996). As an EA variation operator contributes to reproduction (cf. 2.6, p.39), it introduces a new, real individual to the underlying search process. Thus, neutral mutations of a real genotype d build the real neutral network of $m(d)$.

(3.14)

As an extreme, the neutral networks of all phenotypes percolate through search space in the following way inspired by (Eigen 1992). Let a distance measure be given on the search space. Then, for each pair of different phenotypes $(\mathtt{q}, \mathtt{r})$, a pair of genotypes (q, r) exists with $m(q) = \mathtt{q}$ and $m(r) = \mathtt{r}$ such that q is closest possible to r.

(3.15)

This situation enables search to have its trapped population escape from a premature phenotype[18] r by building its real neutral network R: convergence implies that ever more individuals in the population represent r, while their genotypes—r's neutral variants—form R. Especially, for a given $r' \in R$, ever more real individuals may represent r' due to selection-guided copying, thus "reinforcing" R.

(3.16)

Therefore, for $r \in R$, with rising probability, an r-instance carried by an individual is subjected to a neutral mutation which extends or reinforces R, implying that R's build-up is autocatalytic.[19] Thus, either, with sufficient population size given, R grows into the neutral network N of r, or R remains a proper subset of N where one may imagine the trapped population leaping and walking. In this manner, remarkable search progress can come as transition of individuals between neutral networks, as follows.

(3.17)

Consider a measure that represents distance as minimal number of point mutations leading from a genotype r to q. A real neutral network R of a premature phenotype r may be close (cf. 3.15, p.60) to a network Q of a better yet unreal phenotype. Therefore, with many neutral variants $r \in R$ (cf. 3.16, p.60), one of them is likely to be selected for a non-neutral mutation into a $q \in Q$, resulting in a "dam breach" through which the population escapes.[20]

- In summary, for a developmental search algorithm, the very process that is detrimental to successful exploration, i.e., premature convergence, dissolves itself as result of its effect.[21]

(3.18)

[18] Example: A local unacceptable optimum.

[19] This self-enforcing growth is an emergent phenomenon, as it is not established by a dedicated method, so that, to the naive observer, the network is growing itself. Processes of this type are typical of the morphogenesis of natural organisms.

[20] Due to the locally high fitness of r, a mutation of $r \in R$ may also likely lead to a genotype q' with *lower* fitness. However, fitness-based selection gives q' little chance for long-term proliferation.

[21] This is a further cybernetic example of implicit self-regulation through coupled exciting and inhibiting processes (cf. 2.2, p.34).

In this perspective, we see a developmental algorithm giving rise to a genotypic population divided into disjoint subpopulations that represent real neutral networks. During premature convergence, search freezes in the single, phenotypic "population," while it continues in the genotypic one.

More neutrality-related arguments for using developmental algorithms in EAs follow. A search algorithm often faces a constrained optimization problem p, i.e., a potential solution $\mathsf{q} \in pot_p$ must satisfy one or more restrictions in order to be a feasible $\mathsf{q} \in sol_p \subset pot_p$. Such a constraint (s. def.) is the more rigid, the more a user requires its obedience from q, and if there is no compromise, the constraint is named hard. Then, two extreme flavors of a search algorithm alg_{a_m} locate feasible solutions:

- On the one hand, alg_{a_m} may only use safe search operators that obey all constraints, so that each operator, when given a feasible point $q \in rep_a$, returns $q' \in rep_a$. Thus, alg_{a_m}'s search space is closed within all feasible decision points, i.e., $sea_a \subset fea_m$, which, due to 2.4, p.31, implies $rep_a = sea_a$,[22] from which it follows that

  – a common search algorithm exclusively uses safe search operators.

- On the other hand, alg_{a_m} may use search operators that effect full search, i.e., there is an iteration of their applications that locates each decision point. Thus, $dec_p = sea_a$ follows, that is, maximal structural diversity is available. However, $rep_a = sea_a$ must hold, implying $dec_p = rep_a$. Therefore, via semantic mapping m, each $d \in dec_p$ is to represent a feasible solution.

  – Given such m, one has the freedom to design unsafe search operators.

  Due to the presupposed hard constraint, there is $d \in dec_p$ whose representation under the given encoding is not applicable.[23] Thus, $d \notin sol_p$, but

  $$m(d) \in sol_p \Leftarrow dec_p = rep_a \text{ (above)}.$$

  This gives $d \neq m(d) \Rightarrow m \neq id_{dec_p}$ which implies that

  – alg_{a_m} is developmental (cf. 3.3, p.54).

  In particular, due to $dom(m) = dec_p$, m is a genotype-phenotype mapping

  $$gp_a = m|_{(rep_a = dec_p)} = m. \tag{3.19}$$

  The described extreme developmental search algorithm shall be called full : it can locate each structure, and each structure represents a feasible solution.

[22] This is a minimal condition for a search algorithm (cf. definition of representative space).

[23] For instance, in the robot example, not each setting of motion-control parameters avoids collision.

This property is desirable as it allows for many neutral variants of phenotypes, so that their neutral networks may be close: then, percolation is given.

- We therefore decide that $\mathrm{d}_{\rightarrow}$, the empirical developmental search algorithm, shall be full. This is of particular relevance here, because a GP algorithm must deal with hard constraints given for *each* problem it is applied to, as follows. (3.20)

3.5 Hard generic constraints for a Genetic-Programming algorithm

3.5.1 Preliminaries

We use the term GP problem for a task p to be approached by a GP algorithm.[24] For such a procedure, a structure of interest is represented by an algorithm, so that $s \in sol_p$ is such a feasible algorithm.

Some basic terminology of formal languages which can describe an algorithm is to be assimilated into our framework next. A string over alphabet Δ is also called a word, and $\boldsymbol{\Delta^*}$ denotes the set of all words over Δ. A grammar G defines, among others, a finite set of rules, a terminal alphabet Θ containing terminal symbols, and a start symbol[25] from which one can, using rules, derive a word over Θ that is known as sentence. The set of all derivable sentences is named the language of G [$\boldsymbol{L(G)}$]. G's rules define the syntax of sentences from $L(G)$, i.e., their structure.

In order to structurally represent a feasible solution, a GP algorithm alg_{a_m} employs a given target language $\boldsymbol{L_a}$. A given grammar G with $L_a = L(G)$ shall be called alg_{a_m}'s target grammar. We allow $\mathrm{p}_{\rightarrow}$, the projected search algorithm, to employ a Turing-complete target language, because the technical objective refers to real-world problems. The declaration of an available target language as L_a is necessary, since the underlying computing environment of alg_{a_m} will usually offer several languages, while alg_{a_m} structurally represents all feasible solutions as $w \in L_a$.[26]

- The terminal alphabet of L_a shall be called the target alphabet of alg_{a_m} [$\boldsymbol{\Omega_a}$] with $\omega \in \Omega_a$ as a target symbol. Thus, Ω_a defines the symbol set of alg_{a_m}, i.e., the union of its function and terminal sets.

- We identify $w \in L_a$ with an equivalent program, i.e., a computationally identical sentence from a programming language that is supported by the computing environment.

[24] While we focus on such methods, several of the coming concepts are readily transferable to other types of search processes.

[25] Literature also gives the synonym sentence symbol.

[26] For instance, for its quality evaluation, a feasible solution from target language LISP must be expressed in the machine language of the processor that runs the underlying search process.

3.5.2 Constraints

For given problem p, in terms of target language L_a, a GP algorithm $\mathtt{gp}\boldsymbol{alg}_{\boldsymbol{a_m}}$ structurally represents $s \in sol_p$ as $w \in L_a$. We therefore identify s and w, implying $sol_p \subset L_a$. (3.21)

Each material computing environment offers finite memory as a hard size constraint, so that there is, as we name it, a maximal phenotypic size $l \geq |w|$ for a $w \in L_a$ that can actually be produced by $\mathtt{gp}alg_{a_m}$. (3.22)

Thus, if L_a's syntax allows for the construction of a sentence $w \in L_a : |w| > l$, then w is infeasible, implying $sol_p \subsetneq L_a$. In particular, the syntax of a target language to be employed regarding a real-world problem must allow for indefinite $|w|$, because the size of an acceptable phenotype is unknown *a priori*.[27] (3.23)

A potential solution is an unconstrained structure, so that, for a GP problem p, $pot_p = \Omega_a^*$ (cf. 3.2, p.52). With 3.23, we get $sol_p \subsetneq L_a \subsetneq \Omega_a^* = pot_p \Rightarrow sol_p \subsetneq pot_p$ in accordance with 2.4, p.31. With alphabet Δ, $l \geq 0$, let

$$\boldsymbol{\Delta}^{\boldsymbol{max_l}} := \{s \in \Delta^* \mid |s| \leq l\},$$

the set of strings over Δ with, at most, size l.

For a given computing environment with size constraint $k > 0$, all structures representable there reside in $\Omega_a^{max_k} \subsetneq \Omega_a^* = pot_p$. Thus, eventually,

$$sol_p \subset (\Omega_a^{max_k} \cap L_a) \subsetneq L_a \subsetneq \Omega_a^* = pot_p$$

follows from above. (3.24)

L_a implies the most prominent generic hard restriction: it requires $s \in pot_p$ to be, as we name it, legal, i.e., syntactically correct $\Leftrightarrow s \in L_a$, in order to be feasible (cf. 3.21, p.63). This syntactic constraint, as it shall be known, is special among all generic hard restrictions $\mathtt{gp}alg_{a_m}$ may face for a given problem: s with $s \notin sol_p \wedge s \in L_a$ violates one or more restrictions, but it still carries semantics; however, an $s \notin L_a$ that *only* violates the syntactic constraint is meaningless and thus unavailable to a common approach, even if s, as an extreme, is only one point mutation away from a global optimum.

- Due to its significance, mostly, only the syntactic constraint shall be considered subsequently.[28] Then, while legality of $s \in pot_p$ is always necessary for $s \in sol_p$, from here, this restriction is also sufficient for $s \in sol_p$.

[27] Example: the length of an arithmetic expression that appropriately models a given black box's I/O behavior is initially unknown.

[28] Then, regarding the size constraint, we assume, for given $\mathtt{gp}alg_{a_m}$, that there is sufficient memory available during a run.

3.5.3 Syntactic constraint and projected search algorithm

The constraint usually strongly reduces the size of the search space of a common GP algorithm, compared to the encompassing potential-solution space.

For instance, regarding some problem p, consider a fixed-size linear encoding of pot_p over some encoding set with encoding size three, and let the constraints on size and syntax be the only hard restrictions. Let $\mathtt{gp}alg_{a_m}$ be common, implying $m = id_{dec_p}$ and $rep_a = sea_a$, with grammar G describing arithmetic infix expressions over terminal alphabet $\Omega_a = \{a, b, +, *\}$.

From this scenario, to be known as infix example, it follows that $\mathtt{gp}alg_{a_m}$ must initialize three decision variables with one target symbol each such that $s \in sol_p$ results. For instance, while "$a + b$" $\in sol_p$, "$aa-$" $\in inf_p$ due to the syntactic constraint. With $|\Omega_a| = 4$ and three positions in $s \in pot_p$, $|pot_p| = 4^3$ results. However, all two operands from Ω_a may only occur in positions zero and two of $s \in sol_p$, while all two operators may only show in position one, giving

$$|sol_p| = 2^3 = 2^{-3}|pot_p| \Rightarrow$$

$$|sol_p| < |inf_p| < |pot_p| = |sol_p| + |inf_p|.$$

Usually, due to combinatorial explosion, a structural constraint on a composing process such as a GP algorithm results in $\frac{|inf_p|}{|sol_p|}$ rising exponentially over the size of the composition. (3.25)

From a typical maximal linear-expression size $n \geq 10$ for a real-world GP problem, we get, for common $\mathtt{gp}alg_{a_m}$,

$$|pot_p| > |inf_p| >> |sol_p|. \tag{3.26}$$

Thus, we get, as special case of 3.4, p.54,

$$rep_a = sea_a \subset sol_p \subsetneq pot_p.$$

- Therefore, almost all of pot_p is inaccessible to common search, which represents a disadvantage, as follows.

Although, for common $\mathtt{gp}alg_{a_m}$, $d \in inf_m$, by definition, does not semantically represent $r \in sol_p$, d may contain one or more substructures s that stand for an acceptable solution or a—possibly composite—part thereof.

For instance, consider a modified infix example with maximal phenotypic size four. Then, while "$ab * b$" $\in inf_p$, its substructures[29] a, b, "$b * b$" $\in sol_p$, and, especially, it features $b \in sol_p$ twice. Let "$b * b$" be a substructure of an acceptable $s \in sea_a$. Then, the existence of several $s' \in sea_a$ that contain "$b * b$" increases the probability of locating s. However, the larger inf_p is, the fewer such $s' \in sea_a = rep_a$

[29] We drop string notation for one-symbol strings.

exist that are points in *short* walks from real points to s, because $rep_a \cap inf_p = \emptyset$ (cf. 3.4, p.54). For example, given
i) real individual "$a + b$"$\in sea_a$ and
ii) unreal, acceptable "$a + b * b$"$\in sea_a$,
the shortest walk from i) to ii) contains unreal $\sigma =$"$a + b*$"$\in \inf_p \Rightarrow \sigma \notin sea_a$.

Put vividly: common search tries to migrate from good to better representatives, but infeasible space regions block fast transition.

(3.27)

- Summarizing, a structural constraint usually yields $inf_p \neq \emptyset$ and especially $|inf_p| >> |sol_p|$, which is detrimental to common search where it shows as remarkably poor structural diversity in sea_a (cf. 3.26, p.64).

$inf_p \neq \emptyset$ is a property of the underlying problem p and, thus, cannot be eliminated by design of a search algorithm. However, some implications do depend on design, as follows. Unavailability of elements in inf_p may also be detrimental to *non*-common search progress:

- For alg_{a_m} with given $m \neq id_{dec_p}$,

$$inf_p \neq \emptyset \implies inf_m \neq \emptyset$$

may hold, so that an m-redesign with $inf_m = \emptyset$ is a desirable extreme.

From 2.4, p.31,

$$inf_m = \emptyset \implies fea_m = dec_p$$

results, implying

$$rep_a = sea_a \subset fea_m = dec_p = dom(m) \implies$$
$$m(sea_a) \subset m(dec_p) = m(fea_m) \subset sol_p.$$

(3.28)

- In words, theory does not exclude $rep_a = dec_p$, so that, by designing m accordingly, one might be able to provide the maximally available structural diversity, given by dec_p, to alg_{a_m} such that each structure represents a feasible solution.

This situation may be beneficial to alg_{a_m}, because it is the opposite of 3.27. Notably, given an appropriate m, alg_{a_m} is developmental and full, because i) $m \neq id$ holds by precondition, and ii) $rep_a = dec_p$ (above).

- Thus, pursuing the technical objective, we decide that $d_{\rightarrow}$, the empirical developmental search algorithm, shall be full.

Next, depending on this type, further $d_{\rightarrow}$ components can be determined as parts of the projected search algorithm.

Chapter 4

Algorithmic components

4.1 Basic components of the projected search algorithm

4.1.1 Design principles

Minimalism shall guide design decisions in order to save computing resources, i.e., memory and CPU cycles, and it supports a search algorithm's autopoiesis, as it requires and endorses implicitness of the algorithm (cf. 1.2.8, p.26), because each explicitly designed entity eventually calls for manual control. A second design objective is efficiency of an approach's use of these resources in obtaining empirical results. Thus, designed structural representations of entities should be similar to those used by the computing environment. If a trade-off between computing time vs. memory is to be made, saving of time, usually the more limited resource, shall have priority.

Following several design objectives usually leads to conflicting decisions, and, as resolving compromise, a design should be as minimal and efficient as possible in order to be as effective as necessary regarding search progress. For instance, given a problem, a smallest possible target language should be given that probably allows for the expression of an acceptable solution.[1]

The design of the encoding of a decision space comes first, as it necessarily influences decisions on search operators and GPM.

4.1.2 Encoding of a decision space

Put in formal terms defined so far, (Banzhaf 1994) suggests a linear fixed-size encoding of a decision space. Especially, the encoding alphabet is binary, instanced as $\mathbb{B} := \{0, 1\}$, and the encoding structure is ¶, the path-containing class. As we shall show next, this proposal is most advantageous.

[1] Note that the need to determine such a language, along with numerous other technical decisions, are instances of the conflict of recursion.

$\mathbb{B}$ provides for a practical, simplest, and universal encoding of an entity, e.g., a search point, when given a digital computing environment, as its memory is organized as a binary string σ. Given problem p, both $\mathbb{B}$ and ¶ allow for an encoding of $d \in dec_p$ as a binary string τ. Thus, τ can be a substring of σ, implying that $(\mathbb{B}, ¶)$ is an encoding of a decision space that results in most memory-efficient representations of its elements.[2] Also, as consequence, we can represent a search operator most time-efficiently, because there are elementary operators provided by the computing environment for the modification of memory contents, such as bit-inversion.[3]

- For these reasons, we decide on a binary encoding of a decision space. Furthermore, as each considered decision space, dec_p shall have a fixed-size encoding, because an identical encoding size for each $d \in dec_p$ favors a small system specification, obeying minimalism. Thus, per search run on problem p, a fixed-size binary encoding $\{\mathbb{B}, ¶\}$ of dec_p shall be given. Therefore, for given size n, $dec_p = 2^n$, and we identify $d \in dec_p$, encoded as string s over $\mathbb{B}$, with s (cf. 3.2.2, p.52).

Next, an encoding of the potential-solution space of a problem shall be determined for $\mathrm{p}_{\rightarrow}$, the projected search algorithm.

4.1.3 Encoding of a potential-solution space

For $\mathtt{gp}alg_{a_m}$ for problem p,

$$sol_p \subset (\Omega_a^{max_k} \cap L_a) \subset L_a \subset \Omega_a^*$$

holds, while $pot_p = \Omega_a^{max_k}$ (cf. 3.24, p.63). Thus, the encoding alphabet of a pot_p-encoding is Ω_a as given by a grammar defining L_a. Therefore, in order to define the encoding, its structure and the nature of L_a must be determined.

- ¶ shall be this structure, allowing for a space-efficient representation of $\mathsf{q} \in pot_p$ in the underlying memory of the environment.[4]

With $sol_p \subset pot_p$, the former inherits the latter's encoding, so that a feasible solution is structurally represented as a string. This is advantageous, because a string representation can be readily converted to a structure that the computing environment can directly execute. A corresponding program converter, i.e., an interpreter or compiler, produces a machine program[5].

[2] An encoding of a space S over $\mathbb{B}$ as well as the resulting encoding of an $s \in S$ shall be called binary.

[3] Moreover, as a binary encoding of a decision space is standard in the field of genetic algorithms, the large body of respective literature may support further work.

[4] Note: given an alphabet Δ, a $w \in \Delta^*$ does not necessarily require ¶ for its structural representation. For example, *tree* may be represented as "tree"—i.e., via ¶—or, e.g., as some forked structure, i.e., via FNC.

[5] A sentence of a machine language, i.e., a set of exactly those sequences of processor operations that do not yield an undefined state of the environment.

A string representation is also expected by a processor, so that ¶ allows for the encoding of a feasible solution as a machine program. In this case, p$_{\rightarrow}$ would emulate an instance of machine-language GP, e.g., (Nordin 1994; Crepeau 1995; Nordin 1997).

- However, for the present work, L_a shall be an existing, high-level general-purpose language for that a compiler exists. The use of the latter, as opposed to interpreting, allows for saving computing time if the quality of a feasible solution depends on many fitness cases. Also, L_a must be apt for general purposes so that it is good for arbitrary real-world domains. Eventually, it shall be high-level—in particular, not a machine language—so that a real feasible solution can be more easily interpreted by a human user.

(4.1)

In summary, for problem p and alg_{a_m}:=p$_{\rightarrow}$ (projected search algorithm), a pot_p-encoding equals $(\Omega_a, ¶)$ with given L_a as described. As $\Omega_a^{max_k} = pot_p$ with k as p's size constraint, we obtain

$$|pot_p| = \sum_{i=0}^{k} |\Omega_a^i|.$$

(4.2)

4.1.4 The mutation search operator

A mutator is the simplest form of a search operator, taking one argument only, i.e., $d \in dec_p$, from which it derives a different structure, implementing an underlying mutation function (cf. 3.4.2, p.58). Especially, a point mutator [***pm***], in its elementary instance, only changes the value of a single random-selected decision variable to another random value.

Let $d \in dec_p$ be given to pm with $d' := pm(d)$, $d'' := pm(d')$, and so forth. An indefinite number of repetitions gives dec_p. Any search operator with this property shall be called ergodic, and, under non-elitist selection, using a single ergodic search operator yields $dec_p = sea_a$. This equality is necessary for d$_{\rightarrow}$, the empirical developmental search algorithm, to be full as desired (cf. 3.5.3, p.65).

- Thus, we decide to make an ergodic point mutator a search operator of d$_{\rightarrow}$. Since d$_{\rightarrow}$'s decision-space encoding is binary, the bit data type characterizes an atomic component of the genotype representation. Thus, d$_{\rightarrow}$'s point mutator inverts one random-selected bit of a genotype.

(4.3)

As recombination plays a prominent role in many GP algorithms, we discuss its significance for d$_{\rightarrow}$'s design.

4.1.5 Recombination

Typical recombinators used in GP seem to behave like macro-mutation (Luke and Spector 1997), they may introduce an undesirable bloating of phenotypes, at least for a tree representation, e.g., (Angeline 1998), they may be performing worse than mutation, e.g., (Chellapilla 1997), or only slightly better when operating on a bigger population (Luke and Spector 1998). If one wants clear improvement over mutation, one must invest notable effort into designing complex recombinators, especially regarding GP homology, i.e., interchanged segments are similar regarding their intra-parental position, function, or fitness contribution, e.g., (Hansen 2003; Platel, Clergue, and Collard 2003; Nordin, Banzhaf, and Francone 1999).

With respect to effecting macro-mutations in $d_{\rightarrow}$, a recombinator is obsolete because a series of point mutations can amount to the same result. Also, a point mutator alone is a source of variation which is necessary for evolution. Especially, there are frequent, critical states of a GP run when a recombinator, in contrast with a mutator, cannot introduce a novel genotype.

For instance, consider a population, with genetic diversity at zero, of fixed-length binary strings. Then, no number of two-parent, "same-position" homologous recombinations can increase diversity, because the exchanged parental substrings are identical. However, a single point mutation discovers a new genotype.

- In conclusion, as a point mutator is the leanest kind of search operator, supporting minimal design, recombination shall not be used by $p_{\rightarrow}$, the projected search algorithm.

We have determined basic components of $p_{\rightarrow}$ which implements both the empirical common and developmental search algorithm. Next, the genotype-phenotype mapping of the latter can be designed.

4.2 The empirical genotype-phenotype mapping

4.2.1 A paragon

We do not have to determine biological role models for a genotype-phenotype mapping (GPM), because a purely formal discussion is an option. However, trusting an insight underlying Computational Intelligence that nature has solved critical issues of structure design before it yielded the human brain, we shall consider biological GPM. Nevertheless, our strategic objective will not allow for explicit design of numerous, complex, mimicking machineries, which, due to the conflict of recursion, is an endless dead end.

Development of an organic, multicellular phenotype under influence of its genotype, physically represented by DNA or RNA molecules, yields, on a macroscopic level, a phenotype which interacts with its habitat. Ontogeny of this structure is driven by a most intricate cause-effect network that still lacks a whole, explanatory model.

Our focus shifts to the molecular level of observation with a cell as elementary instance of life: it is i) a structural and functional component of a multicellular organism, or ii) an autonomous life form, e.g., a bacterium. The field of molecular biology (for instance, (Watson, Hopkins, Roberts, Steitz, and Weiner 1992)) is concerned with phenomena that build and maintain cellular structures and processes. On this level, a phenotype consists of proteins (s. def.) which are elementary architectural and functional components. Therefore, their context serves well for designing small algorithmic metaphors. To that end, one may interpret a given protein q as a sequence of its composing molecules that are amino acids and that give rise to q's individual, three-dimensional, folded conformation which determines q's role as tissue part or biochemical agent.

The underlying production process, interpreting a genotype, is often called protein synthesis or, more to the point, polypeptide synthesis.

4.2.2 Polypeptide synthesis and genetic codes

Regarding inspiration for GP, we are merely interested in a most elementary description of the synthesis[6] in question, emphasizing the theme of transforming information to behavioral structure. Due to our underlying objectives, advanced artificial machinery should ideally emerge from metaphors of molecular basics, finding its own design that honors the "physics" of the artificial medium. Especially, structural complexity must emerge from a man-made binary substrate. The former thus increases its evolvability by staying in sync with the medium and additional organizational layers to which it contributes by its development which is guided by evolved information.

As instance of natural, evolved information, a DNA molecule is a sequence of nucleotides, namely, adenine (A), guanine (G), cytosine (C), and thymine (T). RNA is a molecule type structurally similar to DNA, consisting of A, C, G, and uracil (U), a further nucleotide. A DNA or RNA segment is a contiguous nucleotide sequence, and a segment that is a *transcriptional unit* in the following context is called a gene. It contains exons, i.e., coding segments, that may be separated by introns that are sequences which do not contribute to the synthesized product.

A cellular process named transcription reads, in its first phase, a DNA gene, producing a similar RNA sequence, known as a primary transcript or mRNA precursor. In a second step, splicing, a.k.a. splicing out (Fincham 1994), or intron splicing (Watson, Hopkins, Roberts, Steitz, and Weiner 1992), removes introns from the precursor, yielding the final product: *(mature)* messenger RNA (mRNA). Three consecutive nucleotides in DNA or mRNA are known as codon, so that one may view such molecules as codon sequences. For instance, in segment "UGCUACGUAAG", starting at the first 'G,' one identifies GCU, ACG, and UAA as codons.

In a scanning step, a cellular process called translation determines codons of an mRNA segment. Due to phenomena that establish the entire process, a certain codon corresponds to, or codes for, a certain amino acid that is said to be encoded

[6] (Creighton 2002) greatly elaborates on proteins.

by the former. Thus, the segment gives rise to an according amino-acid sequence, and a long[7] instance of such a chain, resulting from a gene, is a polypeptide. **For instance**, CCG codes for amino acid proline, and ditto GCU for alanine. Thus, a segment containing "CCG GCU" results in the synthesis of an amino-acid sequence "proline alanine" as part of the sequence that is encoded by the segment.

With four different nucleotides and three per codon, $4^3 = 64$ different codons exist of which 61 encode an amino acid, while three stop codons cause translation to terminate upon their encounter, *ending synthesis of a polypeptide chain.* The mapping of codons and amino acids is a function, known as the—nearly—universal genetic code,[8] so that, for instance, CCG codes for Proline only. Thus, the code is sometimes called frozen, which hints at the view that it, like all stable, complex biological phenomena, has been adapted by organic evolution. It is non-injective, mapping codons onto 20 amino acids, so that, e.g., CCU, CCC, CCA, and CCG code for proline.

A synthesized polypeptide chain folds into a three-dimensional structure that characterizes the protein. Acid sequence and chain length determine the folding that yields the resulting phenotype which, in turn, determines the protein's behavior, i.e., its role.

- From an abstract point of view, biosynthesis, as the entire process is often called, interprets information to produce a structure that exhibits behavior, thus implementing the grand developmental theme on the elementary level of life (see Figure 4.1). The different structural media involved allow for an interpretation of a genotype g such that a phenotype is produced without changing g. (4.4)

- Therefore, biosynthesis shall inspire algorithmic metaphors of the artificial GPM in discussion.

4.2.3 A genotype-phenotype mapping

As we have determined $alg_{a_m} := \mathrm{d}_{\rightarrow}$, the empirical developmental search algorithm for a given problem p, to be full, $rep_a = dec_p$ holds (cf. 3.19, p.61). Thus, for $\mathrm{d}_{\rightarrow}$'s GPM m, namely, $gp_a : rep_a \rightarrow sol_p$, we get

$$gp_a : dec_p \rightarrow sol_p \subset pot_p$$

$$\Longrightarrow\ gp_a \in S_p,$$

i.e., gp_a is a semantic mapping of p.

- In general, for a full developmental search algorithm $alg_{a_{gp_a}}$, we obtain $gp_a \in S_p$, i.e., a GPM of a is a semantic mapping for a.

[7] By this term, molecular biologists usually mean a few dozen up to a few thousands of acids.

[8] There are, indeed, rare exceptions where an organism uses a slightly different code, e.g., for the synthesis of some mitochondrial proteins.

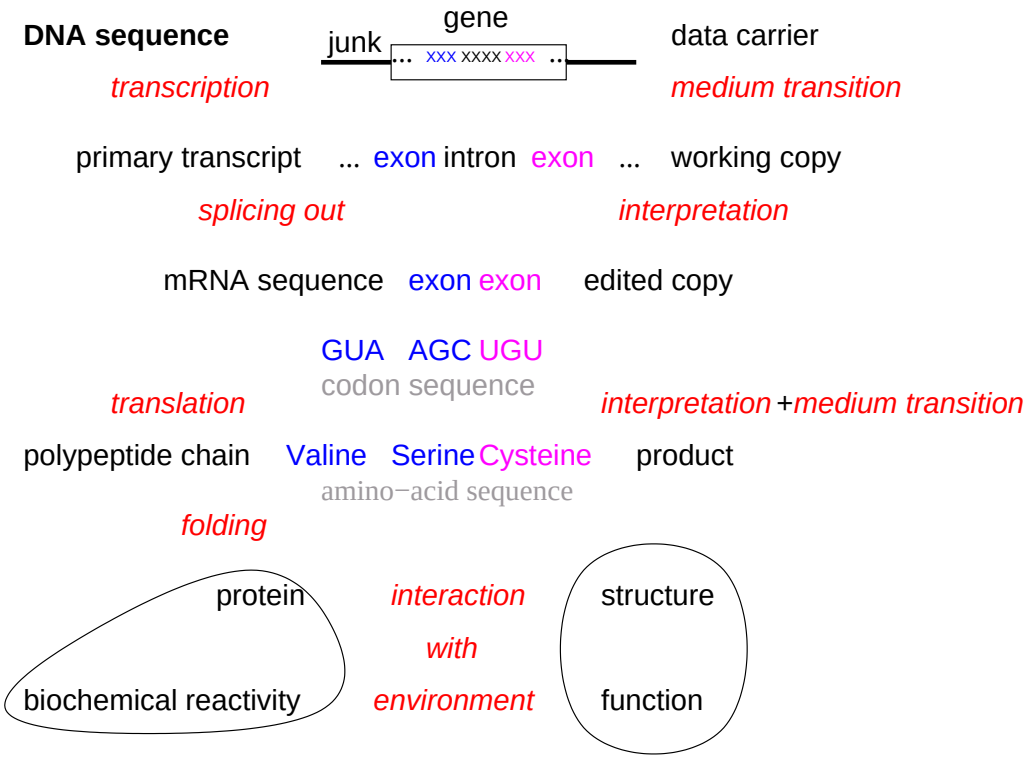

Figure 4.1: Simplified scheme of biosynthesis: we focus on some of its steps that are instances of transitions between media and of data interpretation (for brevity, *processes* are presented mechanistically as agents, rather than as emergent phenomena). A DNA sequence serves as data carrier. *Transcription*, effecting a medium transition to RNA, ignores so-called junk between genes and yields a primary transcript that represents a working copy, so that original data remains untouched. The underlying molecular machinery continues by retaining some subsequences (exons) and *splicing out* others (introns). *Translation* interprets the edited result as codon sequence and represents it in the medium of amino acids. The synthesized polypeptide chain is subject to molecular forces that fold it into a protein. This structure or its biochemical reactivity represent function, i.e., interaction with its environment. The sequence of codons and corresponding acids is information, i.e., an invariant of interest, giving so-called colinearity of gene and encoded protein.

However, for all gp_a, one shows: if $sol_p \subsetneq pot_p$ holds, then

$$\begin{aligned} & inf_p = pot_p \setminus sol_p \neq \emptyset \\ \Longrightarrow\ & \exists\, m \in S_p : img(m) \cap inf_p \neq \emptyset \\ \Longrightarrow\ & img(m) \not\subset sol_p \implies m \neq gp_a. \end{aligned}$$

- Thus, for problem p, not each $m \in S_p$ can be used as GPM of a full, developmental alg_{a_m} to be designed for p.
- Especially, for real-world or GP problem p, $sol_p \subsetneq pot_p$ holds due to 3.26, p.64. Thus, not each $m \in S_p$ may be $gp_{\mathrm{d}_\rightarrow}$.

Therefore, we shall discuss a guideline for designing a GPM of $\mathrm{d}_\rightarrow$, the empirical developmental search algorithm.

4.2.4 An attractive genotype-phenotype mapping

For problem p, $\mathrm{d}_\rightarrow$ being full implies $dec_p = rep_{\mathrm{d}_\rightarrow}$. Thus, we may use each function $m : dec_p \rightarrow sol_p$ as GPM for $\mathrm{d}_\rightarrow$.[9] We therefore define

$$\boldsymbol{G_p} := \{m \in S_p \mid m \neq id_{dec_p},\, img(m) \cap sol_p \neq \emptyset\}.$$

Thus, $m \in G_p$ maps at least one $d \in dec_p$ into sol_p, implying $fea_m \neq \emptyset$.

For a search algorithm alg_{a_m} with a GPM $gp_a : rep_a \rightarrow sol_p$, each $f \in G_p$ qualifies for $gp_a := f$, $m = gp_a$, if alg_{a_m} is full and developmental. Thus, we name f a potential genotype-phenotype mapping of a full developmental search algorithm for p. Therefore, for some p, our task of designing a GPM for $\mathrm{d}_\rightarrow$, the empirical developmental search algorithm, can be restated as identifying an appropriate $f \in G_p$. Usually, $|G_p| > 1$, so that properties of such f are of interest next.

We define an ideal genotype-phenotype mapping $\hat{gp}_{\mathrm{d}_\rightarrow}$ to map $rep_{\mathrm{d}_\rightarrow}$ onto an acceptable $q \in sol_p$, i.e., each point that $\mathrm{d}_\rightarrow$ finds, using $\hat{gp}_{\mathrm{d}_\rightarrow}$, is desirable for the user. Especially, $\mathrm{d}_\rightarrow$ locates q soonest possible, i.e., with the creation of the first individual of the initial population, showing Utopian search performance. Also, with $img(\hat{gp}_{\mathrm{d}_\rightarrow}) = q$, $\hat{gp}_{\mathrm{d}_\rightarrow}$ is usually "maximally non-injective", since, mostly, $|dom(\hat{gp}_{\mathrm{d}_\rightarrow})| = |rep_{\mathrm{d}_\rightarrow}| \gg 1$. For a real-world problem p, one cannot give $\hat{gp}_{\mathrm{d}_\rightarrow}$ *a priori*, at least because q is, by definition of p, unknown. However, a less fantastic, yet beneficial property of a desirable $gp_{\mathrm{d}_\rightarrow} \in G_p$ can be sketched:

- The more genotypes $gp_{\mathrm{d}_\rightarrow}$ maps to acceptable phenotypes, the more likely is $\mathrm{d}_\rightarrow$ to locate such a phenotype. An according $gp_{\mathrm{d}_\rightarrow}$ shall be called attractive, reflecting the notion that the mapping attracts many genotypes to fewer, acceptable phenotypes.

We shall follow this notion as a guideline for designing a practical GPM for the empirical developmental search algorithm.

[9] Note that statements on $\mathrm{d}_\rightarrow$ that are not GP-specific hold for any full, developmental search algorithm.

4.2.5 Necessary and desirable properties of an attractive genotype-phenotype mapping

If an acceptable solution of problem p is unknown, a search algorithm alg_{a_m} with GPM $m := gp_a$ must be able, in principle, to locate each feasible solution $s \in sol_p$. This is equivalent to $img(gp_a) = rng(gp_a) = sol_p$, i.e., gp_a must be surjective, residing in

$$\boldsymbol{SG_p} := \{f \in G_p \mid img(f) = sol_p\},$$

the set of all surjective GPMs for a full developmental search algorithm for p. (4.5)

$|dec_p|$ and $|sol_p|$ are finite (cf. 3.2.1, p.48), so we obtain a simple criterion. In general, for a surjective function f with finite domain,

$$|dom(f)| \geq |rng(f)| \implies (\boldsymbol{f \in SG_p \Leftrightarrow |dec_p| \geq |sol_p|}).$$

(4.6)

We distinguish
i) $|sol_p| = 1 \Rightarrow |SG_p| = 1$, and
ii) $|sol_p| > 1 \Rightarrow |SG_p| > 1$.

i) is trivial, and, for ii), we must identify a beneficial GPM in SG_p through construction. (4.7)

For a function f, an explicit function definition shall be represented by a look-up table that gives $f(d)$ for each $d \in dom(f)$. An algorithm that computes f by use of the table, to be called an explicit mapping algorithm of f, can thus be efficient. Any other function definition and according algorithm shall be named implicit.

For given real-world problem p, an explicit definition of $f \in SG_p$ is impractical: dec_p is large, since sol_p is large (cf. 2.5, p.33) and $|dec_p| \geq |sol_p|$ holds (cf. 4.6). Therefore, we must construct an implicit definition of a surjective GPM for $\mathrm{d}_{\rightarrow}$, the empirical developmental search algorithm. Thus, $\boldsymbol{\mu}$, the according projected mapping algorithm, will interpret a genotype $g \in dec_p = rep_{\mathrm{d}_{\rightarrow}}$, thereby composing $\boldsymbol{p_g} \in sol_p$, the phenotype of g. (4.8)

4.2.6 Structural vs. functional interpretation of a genotype

μ must interpret a genotype as a set of parameters that represents information guiding the composition of the phenotype. One can identify two—possibly combined—types of interpretation:

- functional: a parameter represents information that guides the synthesis of a set of parts of the phenotype.
- structural: a parameter represents this set.

For minimalism, the discussion shall focus on a structural interpretation, because it directly approaches phenotype composition, while a functional interpretation would call for the introduction of additional mechanisms.

- Thus, genotype $g \in dec_p$ represents $\boldsymbol{g_{par}}$, a set of parameters to be identified with structural, genotypic components. Each such component in g_{par} represents P, a subset of parts of p_g which is to be composed. Likewise, $q \in P$ shall be known as phenotypic component of p_g.

(4.9)

μ therefore has to establish a mapping between genotypic and phenotypic components in order to synthesize p_g. Thus, for defining μ, we must determine a representation of a genotypic component.

4.2.7 Structural genotypic representation

To alleviate human interpretation of binary string b, we may indicate a possibly given, logical structure of b by a space that partitions its string notation. For instance, "0110010010" may be rendered as "01 10 01 00 10".

For a search algorithm alg_{a_m}, genotype $g \in fea_m \subset dec_p$ is encoded as a string (cf. 4.1.2, p.67). We thus determine the simplest representation of genotypic component $c \in g_{par}$ as a single substring of g, $c \neq \epsilon$, with component size $|c|$ and component value $\$c$. **For instance**, in $g =$"00**101**01001", one may let c with starting position two and component size three be a component with value $\$c =$"101".

For minimalism, we decide[10] that all components of a given genotype g shall have the same size, called g's component size. Therefore, for all $c, d \in g_{par}$, we get $|c| = |d| \leq |g|$. Note that, here, the component size of a genotype is individual. **For instance**, for genotype "0110 1 0100" with component size four, "0110" and "0100" may be components.

(4.10)

For minimalism, a bit of genotype g shall be a bit of a component of g, so that g only consists of its genotypic components.[11] **For instance**, for the above genotype "0110 1 0100", we have set "0110" and "0100" as components. However, the visually isolated "1", at position four, must be in a component of g, so that, e.g., "0 1 01" must be in g_{par}.

(4.11)

As a further design decision, for all $c, d \in g_{par}$, c and d shall not overlap, since this restriction allows for a simple computation of the starting position of a component.[12] **For instance**, above, "0 1 01" overlaps with the two other components, so that the

[10] A series of design decisions implemented in the present work is to follow. Dropping the implied restrictions hints to future work.

[11] In future work, one may consider frozen genotypic parts, invisible to evolution, that may, e.g., protect absolute, untouchable user knowledge.

[12] In future work, one may relax this constraint, e.g., to implement pleiotropy, i.e., one genotypic feature influencing several phenotypic traits.

given g_{par} is unacceptable here. For the same genotype with component size *three*, however, $g_{par} = \{\text{“011”},\text{“010”},\text{“100”}\}$ passes.

(4.12)

Text unit 4.12, p.77, and 4.11 imply that the size of $g \in fea_m \subset dec_p$ is an integer multiple of its component size, because $|g|$ is fixed for a given problem (cf. 4.1.2, p.68). Thus, for $c \in g_{par}$, $\exists n \in \mathbb{N} : |g| = n|c|$.

(4.13)

As the component size is individual for a genotype (cf. 4.10, p.76), n is individual.[13] Otherwise, however, μ can be defined in a unified way for all genotypes.

- We therefore decide that all components of all genotypes $g \in rep_{\mathrm{d}_\rightarrow}$ shall have the same component size.

Thus, with 4.13, for all $g, h \in rep_{\mathrm{d}_\rightarrow}$, one gets $n = |g_{par}| = |h_{par}|$, i.e., all genotypes consist of the same number of components, called the component number of $rep_{\mathrm{d}_\rightarrow}$.

- In summary, each genotype in $rep_{\mathrm{d}_\rightarrow}$ entirely consists of the same number of non-overlapping equal-sized components.[14]

(4.14)

In particular, the structure of a genotypic component is determined by component size $\boldsymbol{c_s}$ of $rep_{\mathrm{d}_\rightarrow}$. Therefore, the size of a genotype g is implied by c_s and component number $\boldsymbol{c_{\mathsf{n}}}$ as $|g| = c_{\mathsf{n}} \cdot c_s$. Thus, given g and either c_{n} or c_s, μ can compute g_{par} and represent it as a set of c_{n} indexed c_s-bit strings whose ordered concatenation yields g. **For instance**, given $g =$“01101000”, with $c_{\mathsf{n}} = 4$ or $c_s = 2$ known, one gets $g_{par} = \{01_0, 10_1, 10_2, 00_3\}$.

To avoid visual clashes, especially in a mathematical expression, we introduce the bracketed string notations **(..)** and **[..]** as equivalents to the quoted notation, so that, e.g., “001 000”= (001 000) = [001 000].

Next, we can discuss a concrete structural representation of a genotype of $\mathrm{d}_\rightarrow$, the empirical developmental search algorithm.

4.2.8 Binary genotypic representation

With c_s given, the resulting binary representation of a genotypic component allows for 2^{c_s} different component values that μ can interpret. As a genotype g entirely consists of such components (cf. 4.14, p.77), μ can view it as a string over

$$\mathbb{A} := g_{par} = \mathbb{B}^{c_s}$$

[13] In future work, this may support speciation, the emergence of species.

[14] Interestingly, design decisions only resulting from minimalism and efficiency principles often lead to metaphors of biological paragons. This will also be seen regarding the above decisions.

that we name the source alphabet of the representation, with $a \in \mathbb{A}$ as source symbol.[15] **For instance**, for $c_s = 2$, we obtain $\mathbb{A} = \mathbb{B}^{c_s} = \{00, 01, 10, 11\}$, and, with $c_\mathsf{n} = 3$, $g =$"01 11 10" as a genotype over $\mathbb{B}^{c_\mathsf{n} \cdot c_s}$.

With a bit as the atomic component of a genotype (cf. 4.1.4, p.69), $|g| = c_\mathsf{n} c_s$ shall be called g's atomic genotype size, and we name c_n the symbolic genotype size of g [$|\boldsymbol{g}|_{\boldsymbol{s}}$]. **For instance**, with $c_\mathsf{n} = 2$ and $c_s = 3$, $g =$"011 001" has $|g| = 6$ and $|g|_s = 2$.

In summary, for a full developmental search algorithm alg_{a_m} for problem p, we identify a genotype $g \in rep_a = dec_p$ with its encoding, fixed-size binary string (cf. 3.2.2, p.52). Given fixed size $k := c_\mathsf{n} c_s$, we therefore identify dec_p with $\mathbb{A}^{c_\mathsf{n}}$.

(4.15)

$$dec_p = fea_m = rep_a = \mathbb{A}^{c_\mathsf{n}}$$

follows (cf. 3.19, p.61).

(4.16)

- As $\mathrm{d}_{\rightarrow}$, the empirical developmental search algorithm, is full (cf. 3.20, p.62), given a problem p, the user defines the underlying genotypic representation by determining c_n and c_s.

4.2.9 Phenotypic representation

For a given phenotype p_g, $\boldsymbol{p}_{\boldsymbol{g}_{\boldsymbol{par}}}$ shall denote the set of its phenotypic components. For a GP algorithm alg_{a_m} applied to a problem p, $sol_p \subsetneq L_a \subsetneq \Omega_a^*$ for target language L_a (cf. 3.23, p.63). Thus, as $p_g \in sol_p$, it is a string in Ω_a^*.

- In analogy to 4.12, p.77, we decide that a component of phenotype p shall be represented by a set of p-substrings that do not overlap, supporting minimalism, because overlap wastes memory.[16] **For instance**, $\{(a+), (e)\}$ represents a component of "$a + c * e$", while $\{(+c*), (*e)\}$ does not.

- Furthermore, each substring of p must be covered by a $\mathsf{c} \in \mathsf{p}_{par}$. **For instance**, for $\mathsf{p} =$ "$a + b + c$", $\{\{(a)\}, \{(b), (+c)\}\}$ is no potential p_{par}, as $\mathsf{p}_{1,1} =$"+" is not covered.

- As a further decision, a phenotypic component shall contain exactly one substring of p. For instance, for $\mathsf{p} =$ "$a + c * e$", $\{(+c)\}$ is such a simple phenotypic component. Thus, we identify a simple component with its single element.

In summary, for a given phenotype p, p_{par} can be represented as a set of indexed p-substrings whose ordered concatenation yields p. **For instance**, given $\mathsf{p} = (a+b*c)$, $\mathsf{p}_{par} = \{(a+)_0, (b)_1, (*c)_2\}$ or, trivially, $\mathsf{p}_{par} = \{(a + b * c)_0\}$.

[15] The terms reflect the notion that a genotype g is the origin of information on composing p_g.

[16] As with the genotypic discussion, subsequent design decisions are to focus on a minimal design. The implied alternatives and, especially, their hybrids, point to future work.

For $\mathtt{p} : |\mathtt{p}| > 1$, several phenotypic-component sets of $\mathtt{p}$ exist. Thus, for designing μ, we must identify one. Given a $\mathtt{p}_{par}$, if all $c \in \mathtt{p}_{par}$ have the same size, $\mathtt{p}_{par}$ shall be called normal. **For instance**, for $\mathtt{p}$ = "sin(a)", the set $\mathtt{p}_{par} = \{$ "sin(", "a)" $\}$ is normal.

- We decide that μ shall only employ normal sets.

If $|\mathtt{p}|$ is odd, then $\mathtt{p}$ has only two normal component sets:
i) $\mathtt{p}_{par} = \{\mathtt{p}\}$, to be called monolithic, and
ii) the set whose $|\mathtt{p}|$ components all have size one, named atomic.
For instance, for $\mathtt{p}$ = "a + b", $\mathtt{p}_{par} = \{[a]_0, [+]_1, [b]_2\}$ is atomic and $\mathtt{p}_{par} = \{[a+b]_0\}$ is monolithic.

As a current constraint, for genotype g, μ interprets a genotypic component $c \in g_{par}$ as $S \subset p_{g_{par}}, S \neq \emptyset$ (cf. 4.9, p.76). If $p_{g_{par}}$ is monolithic, then $S = p_{g_{par}}$, so that each c semantically represents p_g. **For instance**, let g = 01 10 11 11 00 with p_g = "a + b" with monolithic $p_{g_{par}} = \{[a + b]_0\}$, then each $c \in g_{par} = \{00_0, 10_1, 11_2, 11_3, 00_4\}$ represents $[a + b] = p_g$.

Thus, μ is explicit which contradicts the definition of μ realizing an implicit mapping, and, therefore, a monolithic component set cannot be the only set type employed by μ.

- For that reason and for minimalism, we decide that μ shall only employ the atomic component set of a phenotype.

In conclusion, for the empirical developmental search algorithm $\mathrm{d}_{\rightarrow}$, a phenotypic component is represented by a target symbol $a \in \Omega_{\mathrm{d}_{\rightarrow}}$. Thus, for phenotype

$$\mathtt{p} = \mathtt{c}_0..\mathtt{c}_k, 0 \leq k, \mathtt{c}_i \in \Omega_{\mathrm{d}_{\rightarrow}},$$
$$\text{we obtain } \mathtt{p}_{par} = \{\mathtt{c}_0, .., \mathtt{c}_k\}. \tag{4.17}$$

In extended summary, representations for genotypic and phenotypic components, respectively, have been defined. Next, we continue μ's design by determining, for each component c of a genotype g, the set of p_g's components that c is to represent semantically (cf. 4.9, p.76). (4.18)

4.2.10 A mapping from genotypic to phenotypic structural components

Given genotype g, unit 4.9, p.76, states that each component g_j represents a p_{g_i}. As μ must fully synthesize p_g, each p_{g_i} is to be represented by a g_j. This relation shall be called an individual component relation of g. **For instance**, with $g = (c_0c_1c_2c_3)$ and $p_g = (\mathtt{c}_0\mathtt{c}_1\mathtt{c}_2)$, $\{(c_1, \mathtt{c}0), (c_0, \mathtt{c}_1), (c_3, \mathtt{c}_2), (c_0, \mathtt{c}_2), (c_2, \mathtt{c}_2)\}$ defines such a relation. This concept implies that each genotype is a direct definition of its phenotype, which gives an explicit GPM which has been ruled out for μ. Thus, we must determine a type of component relation that is independent from an individual genotype and

phenotype. To that end, let there be a GP algorithm alg_{a_m} and a source alphabet $\mathbb{A}$. While a given structural component $c = e_{q,l}$ is only associated with its genotype or phenotype e, c's component value $\$c$ from $\mathbb{A}$ or Ω_a is independent from c and e (cf. 9, p.51). Thus, a viable relation of components may only consider their values. (4.19)

Text unit 4.9, p.76, and 4.19 imply that $b \in \mathbb{A}$ is to represent $O \subset \Omega_a$. Thus, an according component-value relation of alg_{a_m} is described by

$$\boldsymbol{cvr_a} \subset \mathbb{A} \times \Omega_a.$$

Therefore, with $c \in g_{par}$ representing $\mathtt{c} \in p_{g_{par}}$, the latter carries a value from $cvr_a(\$c) \subset \Omega_a$.

- Since a component carries exactly one value (cf. def. of the node function), $|cvr_a(\$c)| = 1$ must hold.[17] Eventually, therefore, a component-value relation is a function $cvr_a : \mathbb{A} \to \Omega_a$. (4.20)

  For instance, given $\mathbb{A} := \{00, 01, 10, 11\}$ and $\Omega_a := \{\mathtt{a}, \mathtt{b}, +, *\}$,

  then $cvr_a := \{(00, \mathtt{a}), (01, \mathtt{b}), (10, \mathtt{b}), (11, +)\}$ may be designed.

  We obtain $dom(cvr_a) = \mathbb{A}$, $rng(cvr_a) = \Omega_a$, and, as cvr_a is a function:

  $$|\mathbb{A}| \geq |\Omega_a| \;\Leftrightarrow\; \forall(S \subset \Omega_a : S \neq \emptyset)\; \exists cvr_a : S = img(cvr_a). \tag{4.21}$$

The set of all source symbols of a genotype g shall be called its genotypic alphabet $\mathbb{A}_{\boldsymbol{g}} \subset \mathbb{A}$.[18] Accordingly, one may speak of the phenotypic alphabet $\boldsymbol{\Omega_{p_g}}$ of p_g and g. Given g and cvr_a, μ can compute

$$\Omega_{p_g} = \{\mathtt{o} \in \Omega_a \;\big|\; \exists\, b \in \mathbb{A}_g : cvr_a(b) = \mathtt{o}\} = cvr_a(\mathbb{A}_g).$$

Thus, g and cvr_a determine Ω_{p_g} which μ can use for synthesizing p_g. As a composer of p_g (cf. def.), μ requires seq_{p_g}, the phenotype's sequence information (cf. 3.2.1, p.50) that orders the $o \in \Omega_{p_g}$, thus yielding p_g.

4.2.11 Phenotypic sequence information

The meaning of a genotype g manifests itself in g's contents sequence, that is, the combination of a content and its position is significant. Thus, expressing this combination, given by g, in p_g is the essence of genotype-phenotype mapping. To that end, μ is to determine p_g's sequence information. For a real-world problem, usually, $|\Omega_{p_g}| > 1$ holds. Thus, there may be more than one potential p_g, i.e., a legal string over Ω_{p_g}, while, however, only one p_g is to be composed, implying that μ must

[17] Relaxing this constraint calls for future work on an additional mechanism that determines one of several values, possibly depending on phenotypic status and environment.

[18] Example: $g = (00\ 11\ 11\ 00) \Rightarrow \mathbb{A}_g = \{00, 11\}$.

determine exactly one sequence information.[19] To that end, μ could generate p_g as, e.g., a legal random string over Ω_{p_g}, which would, however, violate the GPM essence. Therefore, artificial evolution must learn the structure of a solution as well as its components, the latter represented here as elements from Ω_{p_g}. To that end, one must link genotypic and phenotypic structure, so that the former assumes meaning via fitness feedback. Here, this approach requires connecting seq_g, the genotypic sequence information, with seq_{p_g} which μ must derive from the former.

Consider $k : k+1 = c_{\mathsf{n}}$, the given component number of $rep_{\mathrm{d}_{\rightarrow}}$. Thus, genotype $g = (g_i) \in \mathbb{A}^{k+1}$ for $0 \leq i \leq k$, representing seq_g. A minimal approach to linking genotypic and phenotypic sequence information identifies g's position sequence 0,...,k with the position sequence of a string $\mathtt{p} \in {\Omega_a}^{k+1}$, thus aligning g_i with $\mathtt{p}_i$. Eventually, to complete $\mathtt{p}$, each $\mathtt{p}_i$ must receive an $\mathtt{o} \in \Omega_{p_g}$. Honoring the GPM essence, one efficiently derives, from $g = (g_i)$,

$$\mathtt{p} := (cvr_a(g_i)) \in {\Omega_a}^{k+1}.$$

This composition of $\mathtt{p}$ from g realizes the latter's transcription under cvr_a,

$$\boldsymbol{tra}_{\boldsymbol{fea_m,cvr_a}} : fea_m \rightarrow {\Omega_a}^{k+1},\ tra_{fea_m,cvr_a}(g_0...g_k) := cvr_a(g_0)...cvr_a(g_k)$$

($\boldsymbol{tra}$ for short), yielding the primary transcript of genotype g (under tra). (4.22)

Note that tra is well-defined, because $dom(cvr_a) = \mathbb{A}$ and $rng(cvr_a) = \Omega_a$ (cf. 4.2.10, p.80) and since cvr_a is a function (cf. 4.20, p.80).[20] We obtain, due to 4.16, p.78,

$$dom(tra) = fea_m = \mathbb{A}^{c_{\mathsf{n}}},\ rng(tra) = \Omega_a^{c_{\mathsf{n}}}. \tag{4.23}$$

With $img(cvr_a) \subset \Omega_a$ (cf. 4.21, p.80),

$$img(tra) = (img(cvr_a))^{c_{\mathsf{n}}} \subset \Omega_a^{c_{\mathsf{n}}},$$

so that, here, one does not get a narrower description of $img(tra)$ than the trivial statement that, for a function f, $img(f) \subset rng(f)$ (cf. 2.1, p.29). An algorithm computing tra shall be called a transcriptor $\boldsymbol{\tau}$ which, especially, computes the sequence information of the primary transcript of g. **For the infix example,** consider

$$\mathbb{A} := \{00, 01, 10, 11\},\ \Omega_a := \{\mathtt{a}, \mathtt{b}, +, *\},\ cvr_a := \{(00, \mathtt{a}), (01, \mathtt{b}), (10, \mathtt{b}), (11, +)\},$$

and $c_{\mathsf{n}} := 3$. This implies $img(tra) = \{\mathtt{a}, \mathtt{b}, +\}^3$, and $g := 01\ 10\ 00$ and $h := 01\ 11\ 00$ yield $tra(g) =$ "$\mathtt{bba}$" and $tra(h) =$ "$\mathtt{b} + \mathtt{a}$" as primary transcripts, implying $tra(g) \notin sol_p$, $tra(h) \in sol_p$.

- *In general*, if a primary transcript of a genotype h is legal, it shall be the phenotype, i.e.,

$$tra(h) \in sol_p \ \Leftrightarrow\ \mu(h) := \tau(h) = p_h. \tag{4.24}$$

The other case requires discussion, because μ must deliver a phenotype for an arbitrary genotype (cf. 3.19, p.61).

[19] Future work may look into relaxing this constraint, having g yield several permuted, feasible strings over Ω_{p_g}, thus exploring a phenotypic space of g.

[20] Interestingly, the formal discussion, unrelated to natural phenomena, has lead to metaphors of a cellular process (cf. 4.2.2, p.71).

4.2.12 Repairing a phenotypic component sequence

For a genotype $g : tra(g) \notin sol_p$, to maintain the GPM essence, μ must map $tra(g)$ to a $\mathsf{g} \in sol_p$ that is to be the phenotype. This repairing has to be deterministic due to 2.1, p.30. (4.25)

First, we analyze whether μ can determine for given $tra(g)$ whether its repairing is to commence, at all. To that end, an efficient algorithm for deciding $tra(g) \in sol_p \subset L_{\mathrm{d}_\rightarrow}$ must exist. Therefore, a target language of $\mathrm{d}_\rightarrow$ shall be defined by an LALR(1)[21] grammar G (Aho, Sethi, and Ullman 1986), because, for $L(G)$, the word problem is in O(n). (4.26)

Especially, during LR parsing of a string w, it is always efficiently possible to compute the legal-symbol set of the current—i.e., actually scanned—w_i. The set contains those terminal symbols of G that are syntactically correct for w_i regarding $w_0..w_{i-1}$. **For the infix example**—which shall be the default for subsequent illustrations—, in $w =$ "abc", current $w_0 =$ a is legal, and current $w_1 =$ b is not. Regarding our practical background, we note: a language that is essentially LALR(1) is valid for real-world problems, because it can be Turing-complete.[22]

- As a further design decision, we set $\epsilon \notin L_{\mathrm{d}_\rightarrow}$ for the empty word, because representing behavior by ϵ is counter-intuitive. (4.27)

- In summary, an efficient repairing algorithm, or cleaner, $\boldsymbol{\rho}$ exists that, given $tra(g)$ and $L_{\mathrm{d}_\rightarrow}$, decides $w := tra(g) \in L_{\mathrm{d}_\rightarrow}$. Especially, during parsing, ρ can decide whether the current w_i is legal, and ρ can compute its legal-symbol set. (4.28)

A necessary, rectifying structure modification (s. def.) of $w := tra(g) \notin L_{\mathrm{d}_\rightarrow}$ results from a composition of one or more basic repairing methods which ρ may apply to an illegal, current w_i: deletion eliminates w_i, replacement exchanges it for a legal symbol, and insertion introduces one or more symbols into w such that w_i, possibly getting shifted, is legal at its (new) position j.[23]

- A cleaner ρ, given $w := tra(g) \notin L_{\mathrm{d}_\rightarrow}$ and using these methods, is to deterministically deliver $w' \in L_{\mathrm{d}_\rightarrow}$ (cf. 4.25, p.82), and $p_g := w'$.

For instance, consider primary transcript $w =$ "a + +b" with illegal $w_2 = +$. ρ-instances can transform w into legal "a + b" by deleting w_2, or, by inserting a at w_2, into "a + a + b". Eventually, an instance may change w by replacing w_2 with b, giving $v =$ "a + b b" with illegal $v_3 =$ b, so that, e.g., final deleting of v_3 gives "a + b". Alternatively, it may alter v by inserting + at v_3, giving "a + b + b".

[21] Look-Ahead one symbol during the Left-to-right scanning of the input for producing a Rightmost derivation.

[22] This is exemplified by many general-purpose languages like C and FORTRAN90.

[23] For purists: the second one may not be considered basic if one prefers to view it as a composition of the first and third.

In general, ρ computes the transformation sequence

$$(tra(g) = t_0 \to t_1 \to .. \to t_{n-1} \to t_n = p_g),$$

with $t_i, 0 < i < n$, as intermediate transcripts of genotype g. The examples show that repairing methods may or must be combined in order to have a phenotype result. A pure ρ employs one repairing method only until it uses a different method once, at most, at its end. Accordingly, such ρ shall be known as *deleting*, *replacing*, or *inserting* repairing, respectively. For instance, as seen, repairing may progress, using replacement only, up to the end of the current intermediate transcript from which a phenotype can only emerge by deletion or insertion.

The variance of phenotype size over a population favorably distinguishes Genetic Programming from standard flavors of other Evolutionary-Computation paradigms, e.g., Evolution Strategies or Genetic Algorithms.

We have $|tra(g)| = |g|_s$, the symbolic genotype size (cf. 4.22, p.81). In general, for all repairing types, $|p_g| = |g|_s \Leftarrow tra(g) = p_g \in L_{\mathrm{d}_\to}$, because ρ does not modify a legal $tra(g)$.

- Under deleting repairing for a real-world problem, usually, $|p_g| \leq |g|_s$, because this type may remove symbols from the primary transcript. (4.29)

  There are pathological exceptions that virtually only matter for toy problems. **For the infix example**, consider $tra(g) = +$, so that deleting repairing, first, eliminates the only symbol, yielding intermediate transcript ϵ. For a given target language whose smallest sentences have size three, ρ must thus insert, at least, three symbols into the transcript to generate a phenotype, resulting in $|p_g| > |tra(g)| = |g|_s$.

- Under replacing repairing, usually, $|p_g|$ about equals $|g|_s$, because repairing, reaching the end of the current intermediate transcript, must delete or insert few symbols only, if at all. (4.30)

- Under inserting repairing, usually, $|p_g| \geq |g|_s$, because ρ mostly does not have to delete or replace symbols eventually, since it can grow an illegal intermediate transcript into a phenotype. (4.31)

- In summary, we have introduced repairing methods by which ρ can transform a given $tra(g) \notin L_{\mathrm{d}_\to}$ into a phenotype it returns, while, else, it delivers $tra(g)$. Thus, ρ returns p_g for the primary transcript of an arbitrary genotype g.

Next, we can complete the definition of μ by combining its subalgorithms τ, the transcriptor, and ρ, the cleaner.

4.2.13 Composing a genotype-phenotype mapping

Computing a GPM m of $\mathrm{d}_{\rightarrow}$, the empirical developmental search algorithm, μ interprets a genotype $g \in dec_p = rep_{\mathrm{d}_{\rightarrow}}$ by composing its phenotype $p_g \in sol_p \subset L_{\mathrm{d}_{\rightarrow}}$ (cf. 4.8, p.75). Thus, μ invokes τ for given g and receives primary transcript $tra(g)$. It then calls ρ on $tra(g)$ and receives phenotype p_g. We obtain $\mu(g) := \rho(\tau(g))$.
(4.32)

Thus, μ computes

$$m = tra \circ \boldsymbol{rpr},$$

the concatenation of tra and rpr, the function computed by ρ. In particular, given a problem p, μ is to compute $m \in SG_p$, a surjective GPM (cf. 4.7, p.75).

- As $\mathrm{d}_{\rightarrow}$ is full and developmental, we will construct a surjective μ-instance for the general case of such a GP algorithm alg_{a_m}. We begin with tra, the first component of m.

Let a component-value relation cvr_a be given along with a

$$g \in rep_a = fea_m = dec_p$$

and $c_{\mathsf{n}} := (k+1) \in \mathbb{N}$ as component number of rep_a. We then have

$$tra : dec_p \rightarrow {\Omega_a}^{k+1},\ tra(g_0..g_k) = cvr_a(g_0)..cvr_a(g_k).$$

Next, we identify properties of rpr. As ρ takes an arbitrary primary transcript as input, $dom(rpr) = {\Omega_a}^{k+1}$ follows.
(4.33)

Since ρ computes a phenotype,

$$img(rpr) \subset sol_p \subset ({\Omega_a}^{max_l} \cap L_a) \subsetneq L_a$$

follows with $l > 0$ (cf. 3.24, p.63). $l \leq c_{\mathsf{n}}$ or $l \geq c_{\mathsf{n}}$ follows from the above discussion on the sizes of a genotype and its phenotype. In accordance with $img(rpr)$, we define $rng(rpr) := sol_p$. $\boldsymbol{rpr} : {\Omega_a}^{k+1} \rightarrow sol_p$ results.
(4.34)

We must show that $m = tra \circ rpr$ is well-defined. rpr and tra are functions. For given c_{n}, $dom(m) = dom(tra)$, $img(tra) \subset dom(rpr)$, $img(rpr) \subset rng(m)$ hold, as follows, and give the statement.

$$dom(m) = dec_p = fea_m = dom(tra)$$
$$img(tra) = (img(cvr_a))^{c_{\mathsf{n}}} \subset {\Omega_a}^{c_{\mathsf{n}}} = dom(rpr)$$
$$img(rpr) \subset sol_p = rng(m)$$

(cf. 3.28, p.65)(4.23, p.81; 4.33, p.84). □
(4.35)

As $m : dec_p \to sol_p$ by definition, m surjective $\Leftrightarrow$ $img(m) = sol_p$. For a certain $l > 0$, $sol_p = ({\Omega_a}^{max_l} \cap L_a)$ (cf. 3.24, p.63). Thus, due to 4.35,

$$m \text{ surjective } \Leftrightarrow \ img(m) = ({\Omega_a}^{max_l} \cap L_a).$$

Depending on given parameters like $c_\mathtt{n}$, the *rpr*-instance, cvr_a, and l, the mapping m may or may not be surjective.

For instance, consider a modified infix example with no fixed-size encoding of pot_p, so that one theoretically faces indefinitely long phenotypes. Let $\mathbb{A} := \{00, 01, 10, 11\}$, $\Omega_a := \{\mathtt{a}, \mathtt{b}, +, *\}$, $cvr_a := \{(00, \mathtt{a}), (01, \mathtt{b}), (10, \mathtt{b}), (11, +)\}$ and $c_\mathtt{n} := 3$. Assume that, actually, due to the size constraint, $l = 4$ holds. Thus, $sol_p = ({\Omega_a}^{max_4} \cap L_a)$, so that, e.g., $\mathtt{abb}*$, $\mathtt{a}+$, $\mathtt{a}+\mathtt{b} \in {\Omega_a}^{max_4}$ and "$\mathtt{a}+\mathtt{b}$" $\in L_a$ hold, and $sol_p = \{\mathtt{a}, \mathtt{b}, \mathtt{a}+\mathtt{a}, \mathtt{a}*\mathtt{a}, \mathtt{a}+\mathtt{b}, \mathtt{a}*\mathtt{b}, \mathtt{b}+\mathtt{a}, \mathtt{b}*\mathtt{a}, \mathtt{b}+\mathtt{b}, \mathtt{b}*\mathtt{b}\}$. (4.36)

Let ρ perform deleting repairing, then the resulting, computed m is not surjective, because

i) "$*$" $\notin img(cvr_a)$ and

ii) for the example, deleting repairing does not introduce a symbol to the phenotype it delivers. Thus, $img(m)$ lacks phenotypes that contain "$*$", so that $img(m) \neq sol_p$, i.e., m is not surjective. □

However, for another $cvr_a : img(cvr_a) = \Omega_a$, m is surjective: (4.37)

Following from the assumption, for each $s \in \Omega_a$ contained in a $\mathtt{q} \in sol_p$, there is $b \in \mathbb{A} : cvr_a(b) = s$.

- Thus, for each $\mathtt{q} \in sol_p : |\mathtt{q}| = 3$—e.g., "$\mathtt{a}+\mathtt{b}$"—, there is $g \in dec_p : tra(g) = \mathtt{q}$, e.g., $g := 00\ 11\ 01$.
- The sole case $\mathtt{q} \in sol_p : |\mathtt{q}| = 1$ remains, e.g., $\mathtt{q} =$ "$\mathtt{a}$". Then, there is $g \in dec_p : tra(g) \notin sol_p$ and deleting repairing eliminates symbols from $tra(g)$ such that $\mathtt{q}$ results.[24]

Thus, for each $\mathtt{q} \in sol_p$, there is a $g \in dec_p : m(g) = \mathtt{q}$, implying i) $sol_p \subset img(m)$, while ii) $img(m) \subset sol_p$ by definition of m.

$$i) \wedge ii) \ \Rightarrow \ sol_p = img(m) \ \Leftrightarrow \ m \text{ surjective.}$$

□

The last example inspires a general design of a desired $m : m \in SG_p$.

Let there be a full developmental $\mathtt{gp}alg_{a_m}$ for a problem p, component number $c_\mathtt{n}$, cvr_a, and $sol_p := (\Omega_a^{max_{c_\mathtt{n}}} \cap L_a)$.

[24] For example, $g := 00\ 01\ 01$ gives $v := tra(g) =$ "$\mathtt{abb}$", so that deleting repairing eliminates v_1 and v_2, which results in $p_g =$ "$\mathtt{a}$".

- **If** rpr **is deleting and if** $img(cvr_a) = \Omega_a$**,**
  then m **is surjective.** (4.38)

As cvr_a is surjective by assumption, for each $\mathtt{q} := \mathtt{q}_0..\mathtt{q}_l \in sol_p,\ l < c_\mathtt{n}$, there is a

$$g := g_0..g_{c_\mathtt{n}-1} \in dec_p : tra(g)_{0,l} = cvr_a(g_0)..cvr_a(g_l) = \mathtt{q}_0..\mathtt{q}_l = \mathtt{q},$$

and if $l < c_\mathtt{n} - 1$, there are $cvr_a(g_i)$, $l < i \leq c_\mathtt{n} - 1$, that are illegal in $tra(g)_{l+1}$. ρ successively deletes these, so that $rpr(tra(g)) = m(g) = \mathtt{q}$ results.

If, alternatively, $l = c_\mathtt{n} - 1$, then $tra(g) = q \in sol_p$, so that $\rho = id$ by definition, and, again, $rpr(tra(g)) = tra(g) = m(g) = \mathtt{q}$ follows.

Thus, for each $\mathtt{q} \in sol_p$, there is a $g \in dec_p$ with $m(g) = \mathtt{q}$, so that, as above,

$$sol_p = img(m) \ \Leftrightarrow \ m \text{ surjective.}$$

□

- In summary, we have derived an implicit, surjective GPM. Thus, for the empirical developmental search algorithm $\mathrm{d}_{\rightharpoonup}$ in particular, such $m \in SG_p$ guarantees availability of each optimal phenotype. While 4.38 shows this special relevance of deleting repairing, comparing all three repairing types must continue.

4.3 Repairing types of the empirical genotype-phenotype mapping

4.3.1 Size-related issues

For a real-world problem, an acceptable feasible solution and, thus, its size are unknown. In this perspective, setting a large genotype size for safety is recommendable. Then, however, only deleting repairing can produce a phenotype whose size is significantly smaller than the set genotype size (cf. 4.29, p.83)(4.30, 4.31) and whose quality marks it as a good acceptable solution.

- Thus, for a real-world problem, deleting repairing supports search performance.[25] [26] (4.39)

Access to a short phenotype is indeed relevant, as such a solution may be more general than a long one, due to an argument from Machine Learning (Mitchell

[25] Example: for 4.36, p.85, and an arbitrary genotype, only deleting repairing can produce the shortest phenotypes, `a` and `b`.

[26] Note: while a small genotype size and an arbitrary repairing type can also yield a short phenotype, they do not give all long phenotypes, because a GPM is a function, so that a genotype represents one phenotype only.

1997). A solution delivered by a learning system may be, as an extreme, specialized in that it explicitly represents only the given training data.[27] (4.40)

- A general implicit solution, in contrast, is comparatively short, so that, for a real-world problem, a small phenotype size is a necessary condition for a good solution. (4.41)

GP as an instance of Machine Learning (cf. (Banzhaf, Nordin, Keller, and Francone 1998)) corroborates 4.40: the relation of size and generality of a feasible solution has been studied, e.g., (Rosca 1996; Nordin and Banzhaf 1995), and results corroborate 4.41.

GP algorithms naturally generate phenotypes of variable size, in contrast to other Evolutionary-Computation flavors, because a target language—a Turing-complete one, in particular—usually contains sentences of different sizes. Variability in phenotypic size is essential to several beneficial GP properties. Especially, feasible solutions of varying size can represent different compromises on conflicting quality criteria. On the one hand, a short, general solution, e.g., a recursion, has a high fitness whose computation, however, may be impractical because quality evaluation must make explicit the associated behavior. On the other hand, a long, special solution has a low fitness but may yield better to evaluation. Thus, the option on producing feasible solutions of different sizes allows for trading computation time vs. memory vs. quality.

Size variability also supports GP's practical relevance, because most real-world problems are multi-objective, and distinct phenotypes may represent different trade-offs regarding all objectives. This issue is essential to the production of algorithmic models of material structures, whose different sizes and shapes cope in separate ways with the challenges posed by a physical environment. The evolutionary production of such models matters to our strategic objective. While GP is not autopoietic, its feasibility for model production is evident (Globus, Lawton, and Wipke 1998; Keller, Banzhaf, Mehnen, and Weinert 1999; Comisky, Yu, and Koza 2000; Porter, Willis, and Hiden 1996; Schoenauer, Sebag, Jouve, Lamy, and Maitournam 1996).

- In summary, under fixed-size genotype encoding, deleting repairing is necessary for a surjective GPM which makes available full phenotypic structural diversity, i.e., feasible solutions of all shapes and sizes.

The term bloating (cf. 4.1.5, p.70) describes a phenomenon ubiquitous in GP (e.g., (Banzhaf and Langdon 2002; Podgorelec and Kokol 2000; Angeline 1994; Angeline 1998; Iba 1999; Langdon and Poli 1997)): during a GP run, an increasing fraction of phenotypic code does not contribute to quality.[28] Thus, in terms of the present

[27] Example: a toy function regression of $f(x) = x^2$ on a given training set of I/O value pairs $\{(1,1),(2,4),(3,9)\}$. A highly specialized feasible solution, given as an algorithm:
`if x == 1 print 1; if x == 2 print 4; if x == 3 print 9`. The most general phenotype `print x*x` is shorter, because it represents an implicit function.

[28] While such code is often called bloat, the negative connotation is not always appropriate (cf. 3.4, p.56).

phenotype encoding, a substring is bloat if either i) quality evaluation does not execute it, or ii) its execution does not contribute to the behavior of the underlying individual, or iii) its contribution to behavior does not contribute to quality. For i), the bloat shall be called inactive, and active in the other cases.[29]

Bloat can be beneficial and detrimental, so that a GP algorithm should allow for it in a controlled manner. Therefore, some algorithms call a dedicated procedure explicitly dealing with bloat (e.g., (Blickle 1996a; Bleuler, Brack, Thiele, and Zitzler 2001)), thus, however, reducing their implicitness. As a typical detrimental effect, bloating increases the average phenotype size in a population, which may waste memory and, in particular for active bloat, CPU cycles. Only deleting repairing implicitly keeps the size of a phenotype at *and* under the fixed size of the associated genotype (cf. 4.29, p.83)(4.30, 4.31).

- In summary, Developmental GP, a fixed genotype size, and deleting repairing allow for i) genotype bloat—a source symbol that represents an illegal target symbol—while limiting its amount, and ii) phenotype bloat, i.e., active and inactive bloat, limiting its amount, too, thus contributing to our technical objective.

4.3.2 Information-related issues

A real-world problem calls for a large target language. Thus, both inserting and replacing repairing mostly require a deterministic (cf. 4.25, p.82) decision which symbol to introduce in place of a current illegal symbol, because, usually, its legal-symbol set contains more than one element. To that end, the user must design an additional algorithm which computes the symbol in question.

This kind of information flux into the developing phenotype does therefore not result from the activity of search operators, yielding an undesirable, deterministic bias which is incompatible with the EA paradigm whose power comes from—directed—non-determinism on the genotype level. (4.42)

For the infix example, replacing repairing must exchange an illegal operator by an operand symbol. Thus, given a bias for, e.g., `b`, "`a` + ∗" will always result in "`a` + `b`".

For a real-world problem, it is often unknown whether a given bias is detrimental or beneficial. Even if one introduces a beneficial one by chance, it may turn detrimental over time for a dynamic problem.

- In summary, both inserting and replacing repairing add to design effort and run time, also decreasing implicitness of a GP algorithm, and they raise an undesirable search bias. For deleting repairing, however, all of the above troublesome aspects are almost not given: as discussed, a few symbols at most

[29] Example: "`a` + `a` − `a`" features active bloat, while `if (true) then a=1 else a=0` has inactive bloat, namely, the else-branch.

are inserted to complete a syntactic unit at the end of a phenotype. Thus, especially for a real-world problem, deleting repairing is recommendable.

(4.43)

An illegal primary transcript $tra(g)$ may contain a legal substring $\mathtt{s}$ that, especially after a long run-time period, may represent a composite building block,[30] i.e., a part of an acceptable solution.

- Only inserting repairing always—i.e., independent from the $\mathtt{s}$-context—results in a phenotype that contains $\mathtt{s}$:

sufficient maximal phenotypic size given, this cleaner always grows $tra(g)$ into a phenotype by creating a context that turns $\mathtt{s}$ legal, while deleting and replacing repairing eliminate, on syntax error at $\mathtt{s}_0$, this current symbol. Thus, inserting repairing is preserving in that it guarantees $\mathtt{s}$ in p_g, also after a change of the $\mathtt{s}$-context in $tra(g)$ due to a mutation of g. **For instance**, consider illegal $tra(g) =$ "$+\mathtt{ba}+\mathtt{b}$" with legal $\mathtt{s} :=$ "$\mathtt{a}+\mathtt{b}$". Then, deleting repairing gives transformation sequence

$$\text{“}+\mathtt{ba}+\mathtt{b}\text{”} \rightarrow \text{“}\mathtt{ba}+\mathtt{b}\text{”} \rightarrow \text{“}\mathtt{b}+\mathtt{b}\text{”} = p_g$$

which lacks $\mathtt{s}$. Consider $tra(g) =$ "$\mathtt{b}+\mathtt{ba}+\mathtt{b}$". Then, replacing repairing, biased for $\mathtt{b}$ and $+$, gives

$$\text{“}\mathtt{b}+\mathtt{ba}+\mathtt{b}\text{”} \rightarrow \text{“}\mathtt{b}+\mathtt{b}++\mathtt{b}\text{”} \rightarrow \text{“}\mathtt{b}+\mathtt{b}+\mathtt{bb}\text{”} \rightarrow \text{“}\mathtt{b}+\mathtt{b}+\mathtt{b}\text{”} = p_g$$

which, again, lacks $\mathtt{s}$. Eventually, under inserting repairing, the two primary transcripts yield

$$+\mathtt{ba}+\mathtt{b} \quad \rightarrow \quad \mathtt{b}+\mathtt{ba}+\mathtt{b} \quad \rightarrow \quad \mathtt{b}+\mathtt{b}+\mathtt{a}+\mathtt{b} = p_g \quad \text{and}$$

$$\mathtt{b}+\mathtt{ba}+\mathtt{b} \quad \rightarrow \quad \mathtt{b}+\mathtt{b}+\mathtt{a}+\mathtt{b} = p_g,$$

respectively, so that $\mathtt{s}$ features in the phenotype.

- In summary, only inserting repairing is preserving.

- However, from a broad perspective (4.3.1, p.86; 4.3.2), deleting repairing is most recommendable, especially for a real-world problem.

Note that those statements on repairing that we have merely illustrated, by flavors of the infix example, also hold for a Turing-complete language, because a cleaner operates in a purely syntax-based manner, while Turing-completeness comes from semantics attributed to terminal symbols. We have chosen the non-complete infix language as default only to keep examples short. (4.44)

[30] We use this term from the field of Genetic Algorithms metaphorically.

4.3.3 Redundancy and genetic diversity

Further issues of repairing matter to the genetic diversity of the underlying population.

Given a full developmental GP algorithm alg_{a_m}, let its GPM $gp_a = m$ be surjective. We call a non-injective function f redundant because there are

$$g, h \in dom(f) : g \neq h,\ f(g) = f(h).$$

Thus, for gp_a,

$$|dec_p| > |sol_p| \ \Leftrightarrow\ gp_a \text{ redundant.}$$

Redundancy of gp_a is necessary for neutral mutations, so that appropriate mutators can maintain genetic diversity of alg_{a_m}'s population (cf. 3.8, p.58). (4.45)

- This maintenance is implicit—in contrast with dedicated methods, e.g., (Bersano-Begey 1997; Keller and Banzhaf 1994)—because it results as side effect. Thus, a redundant GPM—an algorithmic metaphor of ontogeny—contributes to a self-maintaining GP algorithm, approaching our technical objective.

- The redundancy of gp_a is itself implicit. A redundant component-value relation cvr_a is sufficient for the redundancy of gp_a, because, for genotype $g := g_0..g_k$, $tra(g_0..g_k) = (cvr_a(g_0)..cvr_a(g_k))$.

  For instance, let $\mathbb{A} := \{00, 01, 10, 11\}$, $\Omega_a := \{\mathtt{a}, \mathtt{b}, +, *\}$, $cvr_a := \{(00, \mathtt{a}), (01, \mathtt{b}), (10, \mathtt{b}), (11, +)\}$. Then, genotypes 00 11 01 and 00 11 10 yield identical phenotypes "$\mathtt{a} + \mathtt{b}$" since cvr_a is redundant in $\mathtt{b}$.

- A repairing type may support the redundancy of gp_a, because deleting, replacing, or inserting symbols of different genotypes can yield identical phenotypes. In particular, this can occur in the presence of a non-redundant cvr_a. (4.46)

  For instance, let $\mathbb{A} := \{00, 01, 10, 11\}$, $\Omega_a := \{\mathtt{a}, \mathtt{b}, +, *\}$ and injective $cvr_a := \{(00, \mathtt{a}), (01, \mathtt{b}), (10, *), (11, +)\}$. Then, 00 11 10 01 and 00 11 11 01 yield illegal primary transcripts "$\mathtt{a} + *\mathtt{b}$" and "$\mathtt{a} + +\mathtt{b}$". Under deleting repairing, identical phenotypes "$\mathtt{a} + \mathtt{b}$" result.

- Summarizing, implicit redundancy of gp_a may result from either a redundant component-value relation or repairing. In particular, a superposition of both sources may strongly boost their effect. (4.47)

Also, under non-elitist selection, a found, singular excellent genotype g may become unreal again. If gp_a is not redundant, there is no real genotype $h : h \neq g$, $gp_a(h) = p_g$. Thus, the excellent phenotype p_g has been lost, as it is commonly phrased. If, however, the GPM is redundant, then such an h may exist, keeping p_g in the population. Thus, the redundancy of a GPM, boosting genetic diversity, implicitly

fights loss of good phenotypes, approaching the technical objective. Therefore, a mechanism dedicated to supporting beneficial or opposing detrimental effects of variation in Evolutionary Algorithms, e.g., (Aguirre, Tanaka, and Sugimura 1999; Nordin, Francone, and Banzhaf 1995), is not the only way to these ends.

To emphasize the implementation of our objective by the designed GPM, we relate the latter to biology next.

4.4 Interjection: Biological phenomena and algorithmic metaphors

Using an irrational approach, the user may randomly identify a natural phenomenon as a paragon for an artificial metaphor. In contrast, one may also perform a rational, possibly formal, discussion, detached from biology, that yields a result which, upon analysis, is recognized as a metaphor. Our design process of a GPM for the empirical developmental search algorithm belongs to the latter class, as follows.

4.4.1 Biological development and artificial genotype-phenotype mapping

While neutral mutations and polypeptide synthesis (cf. 3.4, p.56) (4.2.2, p.71) are natural issues that have inspired the terminology for some of the introduced algorithmic entities, our discussion of an artificial GPM has unfolded only along formal arguments and the occasional *Gedankenexperiment*, guided by design principles (cf. 4.1.1, p.67) which we have established without reference to biology. Following them, we exclusively cared for properties of the digital medium at hand, not imposing one-to-one replicas of complex biochemical mechanisms and structures on it. In particular, the spirit of the discussed GPM is rooted in the concept of a semantic mapping (cf. 2.1, p.30) which is independent from a biological background.

They are linked, however, in that both the GPM as well as natural development with biosynthesis at its heart turn information into structure that carries function (cf. 4.4, p.72).[31] A fixed-size binary-string encoding of genotypes has emerged from the purely technical discussion (cf. 4.1.2, p.68). It appears that this representation is a metaphor of the linear encoding of genetic information as a DNA or RNA sequence. Also, all genotypic components in a source alphabet $\mathbb{A}$ are to have the same size. It thus turns out that they are an analogy of—equal-sized—codons (cf. 4.2.2, p.71). Thus, $c \in \mathbb{A}$ shall also be called a codon, and a codon size means the component size. Furthermore, all artificial genotypes of a GP run resulted as equal-sized items, which is in parallel to natural genotypes of the same gender of the same multicellular fauna species having (almost) identical size.

[31] Eventually, each biological phenomenon is related to computation, since an organism uses genetic and environmental information for self-maintenance. This connection sits at the root of Evolutionary Algorithms.

Also, a given component-value relation $cvr_a : \mathbb{A} \rightarrow \Omega_a$ (cf. 4.20, p.80) corresponds to the universal genetic code, so that the former shall be known as a genetic code. A sequence of artificial codons influences the structure of the string of phenotypic components (cf. 4.2.11, p.81) which are target symbols. Since a natural codon determines an amino acid (cf. 4.2.2, p.72), a target symbol is a metaphor of such an acid.

- Interestingly, deleting repairing, which surfaced as the most recommendable cleaner from the discussion, is a counterpart of intron splicing (s. def.), an essential phase of the transcription of codons.

The redundancy of an artificial GPM is necessary for neutral mutations which support genetic diversity (cf. 4.45, p.90). This situation is a mirror image of the hypothesis that natural neutrality is a major reason for the observed, high diversity in populations (Kimura 1983; Mukai 1985).

- In summary, numerous components of the discussed GPM that have resulted from technical arguments turn out to be algorithmic metaphors that lead to implicit effects in the empirical developmental search algorithm, supporting the technical objective.

Thus, remarkably, organic evolution, a presumably intention-free phenomenon, results in paragons of metaphors that result from rational design.[32] This conclusion is of practical value to EA design, as follows.

4.4.2 Approaches toward algorithmic metaphors

Considering the overwhelming richness of shapes, sizes, structures, and mechanisms generated by organic evolution, probably each technical solution resulting from rational design is a metaphor of a biological phenomenon.[33]

- This working hypothesis on designing a problem-oriented artificial system means for EA design that there always is an appropriate algorithmic metaphor for a solvable problem.

Thus, if a user is faced with a problem for that i) no efficient deterministic solution is known and ii) theory does not deny solubility, the hypothesis suggests to produce an appropriate EA. Its making may proceed in an informal, experience-guided manner, and such intuitive design can save much time otherwise taken by a thorough—especially, formal—analysis. Put vividly, this approach saves "thinking time" that natural evolution has already spent on a problem class. (4.48)

[32] This observation builds the core of bionics, a cousin of Evolutionary Algorithms, that applies biological principles to the construction of technical structures, because natural products mostly lead regarding key criteria, e.g., performance, economy, and durability. A classic example is the tensile strength of spider silk which is several times higher than that of the best current steel types.

[33] A quick journey through, e.g., ecology, entomology, and molecular biology may dampen skepticism regarding this assumption.

The feasibility of intuitive design has been sufficiently demonstrated on real-world problems by the EA community. Furthermore, there is no comprehensive theory that models EA dynamics on practical problems. For instance, superimposed randomized processes often raise behavior that is mathematically intractable or, at least, hard to catch. Idealizing assumptions, simplifying such challenges, result in theorems that are of little or no real-world relevance. Especially, for GP, the variable phenotype size severely obstructs formal modeling. For example, predicting effects of recombination on the temporal progression of frequencies of genome parts in a population is difficult, as shown in (O'Reilly and Oppacher 1995).

Also, extending theory for a slight, fast intuitive adaptation of an EA to a problem can sometimes require a big theoretical effort. For GP, a typical example is a series of contributions leading to (Poli 2001) that models the above-mentioned progression with respect to a basic recombinator for tree representation. A minor, swift change, e.g., of this operator, reflecting a property of a new problem, may well render the slowly gained theoretical statements useless for the resulting EA.

- For the mentioned reasons, theory strongly lags behind state-of-the-art EAs.[34] Thus, the technical objective calls for intuitive design which shall, besides rational arguments, lead hereafter.

- A golden rule of this approach requires transferring the *essence* of a biological phenomenon into the target medium in a manner that respects the *nature* of the latter. (4.49)

For instance, (Holland 1992) proposes a binary genotype encoding which is beneficial (cf. 4.1.2, p.67) as it fits with properties of the underlying digital medium. As an example of a detrimental design, numerous GP contributions emphasize non-homologous variation. Detrimental effects of this operator type or beneficial effects of homologous recombinators feature in several works, e.g., (Langdon 2000; Nordin, Banzhaf, and Francone 1999).

An explanation for the feasibility of intuitive design is that both a natural life-form as well as a hypothetical, artificial autopoietic aggregate are cybernetic systems (cf. 1.2.2, p.5). This is because cybernetics, in a view dual to *Wiener*'s definition, is the art of piloting (Ashby 1956). In our wording, both an organism and a truly life-like artificial system must navigate through a dynamic, abstract space of constraints by adapting their structures such that the resulting behavior supports their existence. As this behavior is independent from the process that generates the underlying structure, different generators, e.g., organic evolution or intuitive design vs. a rational process, may result in similar structures.[35]

[34] Nevertheless, theory is relevant to practical EA work, because both are mutual catalysts: a fundamental understanding of simple metaphors fosters intuitive design of a complex system that, in turn, provokes growth of formal models.

[35] One may hypothesize that natural evolution and thinking are merely different instances of the medium-independent phenomenon of evolution.

- In summary, intuitive design may well result in a metaphor that leaves little or no space for improvement. So far, rational design has identified a repairing GPM for the empirical developmental search algorithm, while intuitive design is admissible, too.

4.5 An instance of replacing repairing

While deleting repairing is the overall most recommendable flavor (cf. 4.3.2, p.89), its replacing sibling is desirable for simple situations where (cf. 4.30, p.83) the user wants to determine the phenotype size through the mandatory parameter of genotype size, e.g., since the structure of a feasible solution is fixed by the nature of the underlying problem. Thus, we shall determine an instance of replacing repairing for coming, initial experiments. To that end, two cases must be addressed next: a primary or intermediate transcript is illegal if it either

i) contains an illegal symbol, or

ii) is irreducible, i.e., it is the first part of a sentence, such as "$\mathtt{a} + \mathtt{b} +$".

4.5.1 An illegal symbol

A deterministic decision for one of usually several symbols in the legal-symbol set of $\mathtt{s} \in \Omega_a$ [$\mathtt{L(s)}$] must be made (cf. 4.25, p.82) by a corresponding algorithm to be designed next.

The Hamming distance of $g, h \in \mathbb{B}^n$ [$\mathbf{hd(}\boldsymbol{g,h}\mathbf{)}$] is the number of bit pairs $(g_i, h_i) : g_i \neq h_i$, so that $0 \leq \mathrm{hd}(g,h) \leq n$. For example, $\mathrm{hd}(11,11) = 0$, $\mathrm{hd}(00,01) = 1$, $\mathrm{hd}(00,11) = 2$.

Let $B \subset \mathbb{B}^n$ and $g \in \mathbb{B}^n$, then we name

$h \in B : h \neq g$ Hamming-closest to g for $B :\Leftrightarrow \nexists\, i \in B : (i \neq g, \mathrm{hd}(i,g) < \mathrm{hd}(h,g))$.

For instance, with $B = \mathbb{B}^2$, 01 and 10 are Hamming-closest to 11 for B.

Given $\mathtt{S} \subset \Omega_a$, the codon set of $\mathtt{S}$ under a genetic code cvr_a shall be the set $\{c \in \mathbb{A} \mid cvr_a(c) \in \mathtt{S}\}$. For illegal $\mathtt{s} \in \Omega_a$, its legal-codon set is the codon set of $\mathtt{L(s)}$.

Given $tra(g)$, the minimal-distance set of an illegal $\mathtt{s}_i := cvr_a(g_i)$ is that $\mathtt{S} \subset \mathtt{L}(\mathtt{s}_i)$ whose codon set contains exactly those codons that are Hamming-closest to g_i for $\mathtt{s}_i$'s legal-codon set. In this sense, $\mathtt{s} \in \mathtt{S}$ is closest to $\mathtt{s}_i$ under cvr_a. **For instance**, for

$$\mathbb{A} := \mathbb{B}^2 = \{00, 01, 10, 11\}, \quad \Omega_a := \{\mathtt{a}, \mathtt{b}, +, *\},$$

and $cvr_a := \{(00, \mathtt{a}), (01, \mathtt{b}), (10, *), (11, +)\}$, genotype $g := 00\ 01\ 01$ gives $tra(g) =$ "$\mathtt{a_0 b_1 b_2}$" with $\mathtt{L}(\mathtt{b}_1) = \{*, +\}$. For understanding, one may index a codon by the symbol it encodes, so that, e.g., $01 = 01_{\mathtt{b}}$. The legal-codon set of $\mathtt{b}_1$ equals $\{10_*, 11_+\}$. As $\mathrm{hd}(10_*, 01_{\mathtt{b}}) = 2 > \mathrm{hd}(11_+, 01_{\mathtt{b}}) = 1$, the minimal-distance set of $\mathtt{b}_1$ is $\{+\}$, so that '+' is closest to $\mathtt{b}$ under cvr_a.

The index of a codon $c \in \mathbb{A}$ shall be its decimal value, so that, e.g., $c = 11$ has index 3. Given $\mathtt{s} \in \Omega_a$, that $c \in {cvr_a}^{-1}(\mathtt{s})$ with the least index is the lowest-index codon of $\mathtt{s}$.

Eventually, given $\mathtt{S} \subset \Omega_a$, that $\mathtt{s} \in \mathtt{S}$ whose lowest-index codon has the least index among all codons in ${cvr_a}^{-1}(\mathtt{S}) \subset \mathbb{A}$ is the lowest-index symbol in $\mathtt{S}$.

With these preliminaries at hand, we define a replacing-symbol selector $[\boldsymbol{\sigma_r}]$, a deterministic algorithm, inspired by (Banzhaf 1993):

- Given an illegal $\mathtt{s} \in \Omega_a$, the selector picks the lowest-index symbol in the minimal-distance set of $\mathtt{s}$.

The algorithm is well defined because there is exactly one such closest symbol, since cvr_a is a function and a codon index identifies one codon only. **For instance**, for minimal-distance set $\{001_\mathtt{b}, 100_\mathtt{a}\}$ of illegal $\mathtt{s}_i = cvr_a(101)$, the algorithm computes $\mathtt{b}$ as replacement for $\mathtt{s}_i$, because the index of $001_\mathtt{b}$ is less than that of $100_\mathtt{a}$.

There is freedom in identifying exactly one element in a legal-codon set, and our decision to take the codon with the lowest index is arbitrary. This illustrates the conflict discussed in 4.3.2, p.88: replacing repairing, among other flavors, introduces a problem-unrelated search bias. Also, the host of preliminaries emphasizes the design effort such repairing types require. Eventually, while we have assimilated the handy, established term "repairing" for historical reasons, its connotation of a "faulty" genotype giving rise to a pre-phenotype that requires correction is misleading. Rather, ontogeny represents interpretation of information that has *exclusively* been created by phylogeny which learns "correct", i.e., adapted, genetic information through selection and variation, and which has provided developmental machinery as a corresponding interpreter.[36]

Next, we can design $\boldsymbol{r\rho_{\rightarrow}}$, the replacing repairing algorithm in question.

- It starts parsing the given primary transcript $tra(g)$ in a loop: if parsing meets an illegal symbol $\mathtt{s}$, $r\rho_{\rightarrow}$ exchanges $\mathtt{s}$ for $\sigma_r(\mathtt{s})$, and parsing continues. Thus, the loop computes a sequence of (intermediate) transcripts, and it halts when parsing meets the end of the current transcript $\mathtt{t}$ whose finite size warrants termination.

  Thus, $\mathtt{t}$ is a clean transcript, i.e., it does not contain an illegal symbol.

- If $\mathtt{t}$ can be reduced to the start symbol of the underlying target grammar, it is a sentence of the target language, so that $\mathtt{t} = p_g$, the required phenotype. Thus, $r\rho_{\rightarrow}$ terminates.

  For instance, given $tra(g) :=$ "$\mathtt{abb}$", $r\rho_{\rightarrow}$ may yield $p_g =$ "$\mathtt{a+b}$" as clean reducible transcript.

- However, if clean $\mathtt{t}$ is irreducible, such as "$\mathtt{a+}$", $r\rho_{\rightarrow}$ continues as follows.

[36] This view of natural evolution recommends deleting repairing.

4.5.2 A clean irreducible transcript

A finalizing algorithm turns a clean irreducible transcript into a phenotype. To that end, as an instance of structure modification, it adds and/or deletes symbols.

For instance, finalizing may turn
"sin(a) ∗ cos(b) ∗ (" into
"sin(a) ∗ cos(b) ∗ (a)" or
"sin(a) ∗ cos(b)".

Corresponding methods shall be known as appending finalizing and deleting finalizing, respectively. (4.50)

Appending finalizing

- An instance of appending finalizing [$\boldsymbol{\phi}_{\mathbf{ap}}$], given clean irreducible transcript $\mathtt{t} = \mathtt{s}_0..\mathtt{s}_n$, can always—efficiently—produce a phenotype unless a limit on phenotype size interferes, because ϕ_{ap} can compute $\mathtt{L}(\mathtt{s}_{n+1})$ efficiently, from which it can pick the actual $\mathtt{s}_{n+1}$ that it appends to $\mathtt{t}$, producing a clean transcript $\mathtt{t}'$.

- If $\mathtt{t}'$ is irreducible, ϕ_{ap} repeats the above step on $\mathtt{t}'$, else it halts, returning the computed phenotype.

Thus, for use by ϕ_{ap}, we must design a deterministic algorithm (cf. 4.25, p.82) that computes $\mathtt{s}_{n+1}$, called the appended-symbol selector [$\boldsymbol{\sigma}_{\mathbf{ap}}$].

An appended-symbol selector

The number of symbols to be appended should be minimal for avoiding the discussed problem-unrelated search bias and for efficiency: an appended symbol does not result from a codon, whose value and position in its genotype represent learned information, and computing and storing the symbol consumes resources.

- Thus, ideally, appending finalizing is to extend the given irreducible transcript minimally.

To approach this goal, we assign a termination number $\mathbf{n}_\mathtt{s}$ to each $\mathtt{s} \in \Omega_a$. The number indicates the potential of $\mathtt{s}$ to support a minimal transcript extension, and we can guess this potential since L_a is context-free. For a simple target language, such as a set of arithmetic infix expressions, we define $\mathtt{n}_\mathtt{s}$ as the minimal number of symbols to be appended after $\mathtt{s}$ in order to yield the phenotype. For languages with a sophisticated syntax, one may use a heuristic measure that, e.g., prefers an operand to an operator to the opening symbol of a loop, and assign termination numbers accordingly.

A language, depending on its richness, offers more or less symbol classes, known as tokens. For instance, b may be characterized by the token "variable", while π and 42 may be marked as "constant". Thus, we can assign $\mathtt{n}_\mathtt{s}$ to all $\mathtt{s} \in \Omega_a$ of the same

token by tagging the latter with $\mathtt{n_s}$. **For instance**, consider transcript $\mathtt{t} :=$ "a+". Appending "variable" symbol b finalizes t, while "operator" symbol sin requires, at least, three more symbols: '(', an argument, and ')'. Thus, we assign termination number 0 to token "variable" and termination number 3 to token "operator."

The number of symbols actually needed for finalizing after appending s can be larger than $\mathtt{n_s}$ that does not reflect the left context of s, e.g., an open parenthesis. That circumstance, however, does not keep ϕ_{ap} from completing the phenotype, since the termination numbers "pull" the algorithm to an early end. This approximate and feasible approach gives independence from an inefficient, exact algorithm from compiler construction that, given a clean irreducible symbol sequence, calculates the minimal number of appended symbols for finalizing (Aho, Sethi, and Ullman 1986).

Given these preliminaries, we can design a σ_{ap} instance:

1. For clean irreducible $\mathtt{s}_0..\mathtt{s}_n$, compute legal-symbol set $\mathtt{L}(\mathtt{s}_{n+1})$.

2. Return that $\mathtt{s} \in \mathtt{L}(\mathtt{s}_{n+1})$ with minimal $\mathtt{n_s}$ and, in an ambiguous case, lowest index as actual $\mathtt{s}_{n+1}$.

- In summary, we have designed an instance of appending finalizing, ϕ_{ap}, that warrants a *small* extension of its argument into the phenotype.

Next, deleting finalizing, the remaining alternative, shall be discussed (cf. 4.50, p.96).

Deleting finalizing

As there is no degree of freedom in removing a symbol, the only instance of deleting finalizing [$\boldsymbol{\phi}_{\mathbf{de}}$] of given $\mathtt{t} := \mathtt{s}_0..\mathtt{s}_n$ is:

1. delete $\mathtt{s}_n$, yielding $\mathtt{t}' := \mathtt{s}_0..\mathtt{s}_{n-1}$

2. if $\mathtt{t}'$ is reducible $\vee\ \mathtt{t}' = \epsilon$, halt, else continue on $\mathtt{t} := \mathtt{t}'$.

ϕ_{de} halts due to the finite size of its argument, computing the phenotype or ϵ. The latter case, rare in a practical situation, requires further action (cf. 4.27, p.82). **For instance**, transcript "sin(a+", given to ϕ_{de}, yields

$$\text{sin(a+} \quad \rightarrow \quad \text{sin(a} \quad \rightarrow \quad \text{sin(} \quad \rightarrow \quad \text{sin} \quad \rightarrow \quad \epsilon.$$

- In summary, while ϕ_{de} is recommendable for avoiding a problem-unrelated search bias, it is not applicable to each target language as only finalizing type. Thus, a repairing algorithm using deleting finalizing must deal individually with the ϵ-case. We shall encounter examples later.

Replacing finalizing

More flavors of finalizing, customized for given target languages and problems, can be combined from principles underlying ϕ_{ap} and ϕ_{de}. These latter, elementary types more or less ignore the size of the given transcript as a guideline to desired phenotype size (cf. 4.5, p.94). Therefore, as an instance with broader applicability, also for later use, we design a basic, "messy" hybrid, named replacing finalizing, that exchanges the *last and only illegal* symbol $\mathtt{s}_n$ of the given transcript[37] such that a phenotype with identical size results.

For instance, the given, illegal transcript $\mathtt{t}$ = "$\mathtt{a}++$" may result from replacing repairing $rp_{\rightarrow}$ of primary transcript $tra(g)$ = "+b+". Note that such $\mathtt{t}$ cannot necessarily be finalized by $rp_{\rightarrow}$ continuing on $\mathtt{s}_n$, because it may open a syntactic unit, instead. For example, above $\mathtt{t}$ may yield $\mathtt{t}'$ = "$\mathtt{a}+($".

- Thus, we design an instance of replacing finalizing as follows:

  return $\mathtt{s} \in \mathtt{L}(\mathtt{s}_n)$ that

  i) gives a phenotype when replacing $\mathtt{s}_n$,

  ii) is closest to $\mathtt{s}_n$, honoring $rp_{\rightarrow}$'s spirit. Resolve a possible ambiguity by selecting the lowest-index symbol.

- As a customized flavor, replacing finalizing is not applicable for each target language,[38] but it has the advantage over appending finalizing that it does not introduce as much problem-unrelated information. We thus recommend to employ replacing finalizing when possible.

- In summary, an instance of replacing repairing (cf. 4.5, p.94) has been determined that can be an algorithmic component of $\mathrm{d}_{\rightarrow}$, the empirical developmental search algorithm.

Next, to continue the design of $\mathrm{d}_{\rightarrow}$, we discuss the transformation of a phenotype into a representation that is executable in the given computing environment.[39]

4.6 An executable repaired transcript

For its quality evaluation, a phenotype p_g from the given target language L_a is interpreted by a corresponding target machine. If L_a is the native language of the underlying computing environment—a physical approximation of a universal Turing machine (UTM)[40]—,

i) this UTM and the target machine are identical, else

ii) the target machine is virtual, emulated by the UTM.

[37] Thus, this argument is necessarily non-clean.

[38] Example: for "sin(+", no symbol can replace + *and* give a phenotype.

[39] While this technicality does not contribute to search behavior, it is essential to the practical applicability of the algorithm.

[40] For brevity, we identify approximation and its abstract model.

We focus on ii) (cf. 4.1, p.69). Thus, a commercial compiler is to transform p_g into an equivalent machine program which is a sentence of the native language. In order to make p_g accessible to the compiler, $\mathrm{d}_{\rightharpoonup}$ must embed the former within further information, e.g., a main-program frame, in a step called editing.[41] This additional code follows from the definition of L_a and from p_g-properties. Thus, from a current, practical point of view, editing greatly saves resources in that it spares a developmental GP algorithm the learning of such code.[42] However, regarding our strategic objective, it will be essential to let a future, artificial organism learn *all* properties of its environment—not just the nature of the given problem—in order to have it become independent from manually prepared information and wrappers that protect it from harsh reality.

Eventually, compiling and linking of the edited phenotype yield a machine program as executable representation.

- In extended summary, transcription and, on demand, repairing with integrated finalizing are the essential steps of the modeled genotype-phenotype mapping. Editing, compiling, and linking result as technical necessity. To the target machine, the phenotype is executable, while, to the underlying computing environment, the machine program is. This distinction is cybernetically irrelevant because phenotype and machine program are equivalent, while the GPM assigns semantics—the phenotypic behavior—to the genotype which influences the mapping, thus representing genetic information. For an overview, see Figure 4.2.

4.7 Example of a genotype-phenotype mapping

Recall 4.44, p.89. Let a modified instance of the infix example be given with replacing repairing and replacing finalizing. Consider genetic code

$$cvr_a := \{(000, \mathtt{a}), (001, \mathtt{b}), (010, +), (011, *), (100, \mathtt{a}), (101, \mathtt{b}), (110, +), (111, *)\}.$$

Thus, $g := 000\ 001\ 011$ yields primary transcript $\mathtt{t} := tra(g) =$ "$\mathtt{a}_0\mathtt{b}_1*_2$", and replacing repairing applies to $\mathtt{b}_1$ as illegal symbol with codon 001 in g. The legal-symbol set $\mathtt{L}(\mathtt{b}_1) = \{+, *\}$ results, and $*$ with codon 011 is closest to $\mathtt{b}_1$ which, thus, gets replaced by $*$. The intermediate transcript $\mathtt{t}' =$ "$\mathtt{a}_0 *_1 *_2$" follows, and repairing continues with replacing finalizing swapping illegal $*_2$ with $\mathtt{b}$, yielding

$$\mathtt{t}'' = \text{“}\mathtt{a}_0 *_1 \mathtt{b}_2\text{”} = p_g.$$

With, for instance, ISO-C given as the user-desired language, editing adds, e.g., a function frame, so that the edited phenotype results as

[41] A corresponding biological process arranges a polypeptide chain such that its folding (cf. 4.2.2, p.72) begins.

[42] One could, e.g., feed back compiler messages on submitted edited phenotypes to the algorithm.

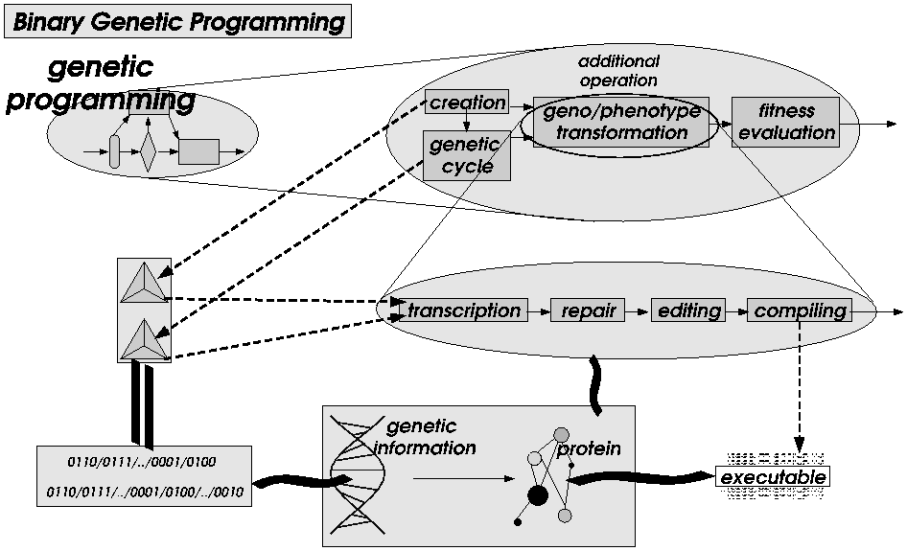

Figure 4.2: Overview of the GP flavor "Binary Genetic Programming," or simple Developmental GP. Creation and the genetic cycle of the underlying Evolutionary Algorithm yield binary genetic information that an extension transforms into phenotypes for subsequent fitness evaluation. This addition transcribes, repairs and edits forms of the genotype such that a compiler can eventually produce the final representation, a phenotypic executable, an analogy of a protein.

```
double fnc(double a, double b) {return  a * b  ;}
```

SOLUTION may then join this representation with others and embed the resulting bundle, which represents the current generation, into a main program that it passes to the given compiler. It forwards the yielded executable file to the underlying operating system for processing and collects subsequent individual output for quality evaluation. For brevity, we skip describing—engineering-intense—technicalities of SOLUTION, e.g., automatic training-data-driven generation of the main program, function-frame generation, and grammar-driven generation of a customized parser for an arbitrary LR(1) target language.

- In extended summary, we have designed essential algorithmic components of $d_{\rightarrow}$, the empirical developmental search algorithm, in a formal context.

Next, the genotypic representation for $c_{\rightarrow}$, the empirical *common* search algorithm and second approach emulated by the projected search algorithm (cf. 3.1, p.47), must be determined.

4.8 A genotypic representation for the empirical common search algorithm

For a common GP algorithm, there is no distinction between a genotype and its phenotype (cf. 3.3, p.54): the structure of interest represents both information and

corresponding function carrier without an internal separation that would allow for interpretation of the former.

- We have determined a phenotype representation for $d_{\rightarrow}$ (cf. 4.1.3, p.69), so that this encoding shall also be used for $c_{\rightarrow}$. Thus, a genotype of $c_{\rightarrow}$ is a sentence of a given target language.

With genotypic representations for both empirical approaches at hand, we can discuss:

4.9 Operators of the projected search algorithm

To keep $c_{\rightarrow}$ and $d_{\rightarrow}$ comparable, they must not differ in principle, with the exception of the existence of a genotype-phenotype mapping. Especially, they have to employ operators with analogous effects on their respective genotypic representations. This must hold due to the principle of mono-causality for comparing entities: ideally, one has them differ in one critical property only, so that an observed behavioral distinction must result from this single issue.

- Therefore, next, we must determine analogous search operators for $c_{\rightarrow}$ and $d_{\rightarrow}$.

4.9.1 Creation and mutation

Random determination

An operator of an Evolutionary Algorithm may have to determine a component of an operand in a randomized manner, e.g., an atomic element of a genotype.

- As default, random determination shall choose each component with equal probability, as this distribution is simplest. **For instance**, for each bit in an n-bit string, its probability of random determination equals $1/n$.

Developmental mutation

- Following from 4.3, p.69, and 4.1.5, p.70, mutation is $d_{\rightarrow}$'s only search operator besides—essential—creation (cf. 2.4.1, p.41).

Later, depending on given codon size and codon number[43] of a particular genotypic representation, a mutator may need specification, since only binary point mutation is standard.

[43] Synonym of component number.

Developmental creation

Minimalism requires a creation operator whose behavior is problem-independent, so that no further entities add to the conflict of recursion.

- Thus, for given codon number n and size s, the operator produces a random-determined $n \cdot s-$bit string. **For instance**, for $n = 3$ and $s = 2$, a created random genotype may equal "01 11 00".

Due to 4.9, p.101, we must design analogous operators for $c_{\rightarrow}$:

Common creation

For a user-given, desired genotype size n, the operator in question is to produce a random sentence $g = p_g \in L_a : |p_g| \approxeq n$. (4.51)

One must allow for $|p_g| \neq n$, because there may be no $\vec{s} \in L_a : |\vec{s}| = n$. **For instance**, for the language of infix expressions over alphabet $\{\mathtt{a}, +\}$, i.e., $\{\vec{s} \mid \vec{s} = \mathtt{a}(+\mathtt{a})^i, i \geq 0\}$, there is no element with even size.

As L_a is LR(1), algorithms we have designed for *developmental* phenotype production can also be used by common creation, e.g., by a minimal instance of a creator of $\vec{s} \in L_a$:

Random-determine $\mathtt{s} \in \mathtt{L}(\vec{s}_0)$ and set $\vec{s}_0 := \mathtt{s}$. Iterate this step for $\vec{s}_i, i \geq 1$, growing an $\vec{s}$ precursor $p\vec{s}$, until $|p\vec{s}| = n$. If—clean—$p\vec{s} \notin L_a$, a ϕ_{ap} flavor[44] yields $\vec{s}$, else $p\vec{s} = \vec{s} = p_g$.

(4.52)

The ϕ_{ap} instance used by the common creator only differs in that no codon order exists which would induce an order on Ω, so that a user-given order is required to resolve ambiguities in case of several symbols with minimal termination number. Note that different creator invocations with same n may result in phenotypes of different size, which may be beneficial (cf. 4.3.1, p.87) and does thus not disadvantage $c_{\rightarrow}$ in a comparison.

For instance, with $n = 4$, the operator may yield a modification sequence

$$\text{“}\mathtt{a}\text{”} \quad \rightarrow \quad \text{“}\mathtt{a}+\text{”} \quad \rightarrow \quad \text{“}\mathtt{a} + \sin\text{”} \quad \rightarrow \quad \text{“}\mathtt{a} + \sin(\text{”}.$$

The ϕ_{ap} flavor may eventually produce “$\mathtt{a} + \sin(\mathtt{a})$” $= \vec{s} \in L_a$ with $|\vec{s}| = 6$. A different invocation may give

$$\text{“}\sin\text{”} \quad \rightarrow \quad \text{“}\sin(\text{”} \quad \rightarrow \quad \text{“}\sin(\mathtt{a}\text{”} \quad \rightarrow \quad \text{“}\sin(\mathtt{a})\text{”} = \vec{s} \in L_a \text{ with } |\vec{s}| = 4.$$

[44] appending finalizing, chosen here for reasons previously explained

Common mutation

Recall footnote 16, p.58, which implies that, for our discussion, mutation is always applicable to a given genotype.

- For minimalism, point mutation shall be the only variator of $c_{\rightarrow}$, the empirical common search algorithm.

Here, a given operand is a $g \in L_a$, so that $\mathtt{s} \in \Omega_a$ is an atomic component to be modified. As a common variator, the operator in question must be safe (cf. 3.4.3, p.61):

---

1. Random-determine $i : 0 \leq i < |g| =: n$.
2. Random-determine $\mathtt{t} \in \mathtt{L}(g_i) : \mathtt{t} \neq g_i$ such that $g' = g_0..g_{i-1}\,\mathtt{t}\,g_{i+1}..g_{n-1} \in L_a$, replace g_i with $\mathtt{t}$, and halt, yielding mutant $g' \neq g$.
3. If such $\mathtt{t}$ does not exist, restart with step 1 to comply with 2.6, p.39.

(4.53)

Thus, the mutator potentially enters a loop that halts with run time going to infinity, because a genotype contains at least one g_i for that $\mathtt{t}$ exists, since, otherwise, L_a would be too poor for the given real-world environment.[45] This common point mutation, called safe single-symbol conversion, changes, for instance, "$\mathtt{a} + \mathtt{b}$" to "$\mathtt{a} * \mathtt{b}$".

- In extended summary, search operators for the projected search algorithm $p_{\rightarrow}$ have been determined.

Thus, we must design a $p_{\rightarrow}$-subroutine for quality evaluation (cf. 2.4.1, p.40) next.

4.10 Quality evaluation and selection for the projected search algorithm

The background of the current issue is given, e.g., by (Blickle and Thiele 1995; Bäck, Fogel, and Michalewicz 1997) that give an overview and theoretical analysis of basic evaluation and selection schemes. A measure of adjusted fitness (Koza 1992) of an individual $\dot{g}$ $[\boldsymbol{a(\dot{g})}]$ has range $]0, 1]$, $a(\dot{g}) = 1$ if and only if $\dot{g}$ is perfect, and, the lower $a(\dot{g})$ is, the lower is $\dot{g}$'s quality. It has the favorable property that its computation only depends on $\dot{g}$, in contrast with, e.g., normalized fitness (Koza 1992). We define our measure $a(\dot{g})$ as $1/(1 + s(\dot{g}))$, where $s(\dot{g})$ is the sum over all fitness cases of the squared error that $\dot{g}$ produces for each case.

[45] For critical applications, one easily safeguards the operator against a pathological, endless loop by adding a counter-based recovery procedure that, e.g., suspends evolution and interviews the user regarding a desirable continuation.

- Thus, the quality measure of adjusted fitness shall be employed by the projected search algorithm.

The principle of selection identifies an element from a population P of individuals based on differences in their quality values (cf. 2.4.1, p.41). A required, minimal selector $\boldsymbol{\sigma}_{\mathbf{p}_\rightarrow}$ therefore picks two individuals for comparison. To that end, the user must provide a quality-based selection rule (s. def.), and we decide on a simple and resource-saving setting of selection probabilities: '1' for the better individual, which implies '0' for the other one. This distribution already determines the candidate to be selected for reproduction, so that $\sigma_{\mathrm{p}_\rightarrow}$ does not have to invoke a pseudo-random-number generator. We obtain as $\sigma_{\mathrm{p}_\rightarrow}$:

---

1. Random-determine $\dot{g}, \dot{h} \in P$.
2. Compute $a(\dot{g}), a(\dot{h})$.
3. If $a(\dot{g}) \geq a(\dot{h})$, return $\dot{g}$, else $\dot{h}$.

   Note that this step does not introduce a bias since both individuals are random-determined.
4. Halt.

$\sigma_{\mathrm{p}_\rightarrow}$ represents an instance of a scheme known as tournament selection with a tournament size of two, or 2-tournament selection.

Reproduction and replacement

During a $\mathrm{p}_\rightarrow$ run, selection for reproduction (cf. 2.4.1, p.41) prepares the creation of offspring through mutation (cf. 2.6, p.39) or copying. On the one hand, a selector for replacement requires computing time. On the other hand, without replacement, an individual has eternal existence, and the resulting growth of the population consumes memory without bounds, if unchecked.

- Heeding the time-first principle (cf. 4.1.1, p.67), we decide against selection for replacement.

- Thus, a simple design of ensuring $\mathrm{p}_\rightarrow$'s limited memory consumption requires a user-given fixed population size (cf. 3.4.1, p.58). (4.54)

  In effect, the population, P, produces an equally big offspring set P'. As a minimal way of determining the reproduced population from P and P', $\mathrm{p}_\rightarrow$ is to discard P, continuing with P' as its population, which implies that the search algorithm is generational (cf. 2.4.1, p.43).

- In summary, we have determined all components of the body of $\mathrm{p}_\rightarrow$'s evolution loop.

4.11 Termination

- Following 2.4.1, p.43, the time-out predicate shall be a termination criterion of the evolution loop.
- As default, the predicate value shall be set to 50, so that, after the completion of that many generations of a $p_{\rightharpoondown}$ run, it terminates. (4.55)

(Koza 1992) states that, after having produced about that number of generations, a run stops showing interesting behavior. This may be but a weak argument for the present work, because our GP algorithms are fundamentally different from the tree- and crossover-based approaches of the cited work. Thus, we may change this default on demand.

Another major termination criterion to be discussed is the success predicate. For our empirical research, it is not as important as in an industrial context, because the focus is not on the production of an individual with a given quality value but on aspects of the behavior of a given search algorithm. In particular, even if a perfect individual is found, we are interested in the continuation of the successful run so that its further behavior can be measured.

- Thus, the termination criterion of $p_{\rightharpoondown}$ shall be identical with the given time-out predicate.
- In summary, we have determined all necessary components of $p_{\rightharpoondown}$, the projected search algorithm, so that it can be synthesized next.

4.12 Projected search algorithm

As $c_{\rightharpoondown}$ and $d_{\rightharpoondown}$, the empirical common and developmental search algorithm, only differ in their genotype-phenotype mapping, we can determine $p_{\rightharpoondown}$ as an instance of the given outline of an Evolutionary Algorithm (cf. 2.4.2, p.43):

- If and only if $p_{\rightharpoondown}$ is to emulate a $d_{\rightharpoondown}$-instance, it invokes one of the designed GPM flavors immediately prior to quality evaluation.

In particular, the flavor that uses the given replacing repairing (cf. 4.5.2, p.98) yields a $d_{\rightharpoondown}$-variant that we call the first developmental search algorithm as it will be an entry point to experiments.

Depending on the common or developmental type of a search algorithm to be emulated, $p_{\rightharpoondown}$ employs the respective representations and operators that we have determined.

- In extended summary, we have designed the projected search algorithm, which concludes the formal part of the initial step toward the technical objective (cf. 3.1, p.47). Next, the empirical part follows, i.e., $p_{\rightharpoondown}$'s behavior shall be observed and discussed. (4.56)

Chapter 5

First empirical problem

5.1 Target language

For experiments, the target language of the projected search algorithm must be decided. Following from 4.27, p.82 (3.5.2, p.63, 4.1, p.69, 4.26, p.82), this language can be a compiled general-purpose language.

- As a design decision, the empirical target language $\boldsymbol{L_{emp}}$ shall be ISO-C (Harbison and Steele 1995) because it is highly portable. (5.1)

  Thus, a final transcript—a phenotype—is a symbol sequence compliant with the ISO-C language definition.

Next, we must determine an empirical problem as an issue of experiments.

5.2 Properties of an empirical problem

5.2.1 Class

There are many contributions applying Genetic-Programming (GP) algorithms to problems of function regression, e.g., (Davidson, Savic, and Walters 1999; Duffy and Engle-Warnick 1999; Guyaguler 2000).[1] This is due to a ubiquitous method of quality evaluation: the input-output relation defining the problem can be conveniently modeled as an isomorphic set of fitness cases. Next, we begin determining a first instance of a regression problem as object of the projected search algorithm (cf. 3.6, p.56).

5.2.2 Type

The previous text unit implies that all problems represent pattern recognition problems. A search algorithm is to find an entity that represents an acceptable solution

[1] Critical remarks to the account that GP can only handle function regressions show the opposite: actually, a general regression problem — given I/O data of a black box, induce the hidden system — is an archetype of problems.

in that it recognizes the pattern in question well. For instance, a located function of a given regression problem recognizes the pattern in the problem representation given as I/O data.[2]

For the intended empirical research, the use of a synthetic problem,—i.e., a designed problem—, is desirable: its properties can be made to support the objective of the research. Opposed to a synthetic problem, there is what we call an emergent problem that arises from an environment as an undesirable phenomenon.

For the hardest pattern recognition problems, a decision maker does not know the type of pattern to be identified. The user of an according search algorithm can, at most, decide whether a found pattern is significant. In terms of Machine Learning, an appropriate search algorithm is an instance of reinforcement learning or even unsupervised learning[3]. (5.2)

A problem occurring in the material world shall be called a physical problem. It is noisy, i.e., i) measured values of a variable are always uncertain because the measuring process, in principle, is imprecise, and possibly ii) a considered variable is irrelevant to the problem, at all. Thus, the resulting problem representation—e.g., fitness cases—is blurry. Therefore, the notion of a "perfect" phenotype is meaningless for a physical problem: a phenotype that satisfies all fitness cases is merely an exact solution to an imprecise problem description.[4] This reasoning is a deep argument for 1, p.47, arguing for focusing the empirical research on the *tendency* of the behavior of a search algorithm.

In summary, we shall determine synthetic regression problems as empirical problems that have some or all of the above properties: aspects of unsupervised learning, and noise. Particularly, we consider above issue ii):

5.2.3 Knowledge

A major task of approaching a problem with emergent properties is determining problem knowledge and representing it to a search algorithm. Especially, determining the set of all problem-relevant variables is a challenge. Ideally, the algorithm determines problem knowledge in parallel to locating an acceptable solution, thus enhancing its performance. (5.3)

We consider the situation of incomplete problem knowledge later. For now, according with 4.9, p.101, we decide that the user shall have full problem knowledge and represent it to the search algorithm. The user is indeed guaranteed to have complete knowledge since the empirical problem is to be synthetic (cf. 5.2.2, p.108).

5.2.4 Dynamics

A static problem problem environment always reacts the same when presented with the same feasible solution. Thus, problem synthesis is simpler for a static than for

[2] A recent buzz word that is synonymous to pattern recognition is data mining.

[3] In professional lingo, the user expects the algorithm to "take me there, I don't know where — get me that, I don't know what."

[4] Thus, an imperfect phenotype may be an even better solution than a perfect phenotype.

a dynamic problem. In order to support a simple execution of empirical research, a considered problem shall be static.[5]

All static real-world problems of interest are variable, i.e., there are different feasible solutions that result in different reactions from the problem environment. Otherwise, there would be only one such reaction, which would imply that each feasible solution would be globally optimal. Also, it is practical to produce a variable problem by use of, e.g., a simple function f with $|img(f)| > 1$. In conclusion, all considered empirical problems shall be variable.

In summary, the first empirical problem shall be a variable synthetic static regression coming with complete knowledge. We can create the determined problem properties with an objective function (s. def.) featuring the same properties. This process results in the simplest problem representation to a GP algorithm, i.e., a set of fitness cases.

5.3 Problem

5.3.1 Representation

The objective function may be arbitrarily chosen from the set of functions with the determined properties (cf. 5.2.4, p.109). It shall be the first empirical function

$$f : \mathbb{R}^4 \to \mathbb{R}; \quad f(m, v, q, a) = \sin(m) \cdot \cos(v) \cdot \frac{1}{\sqrt{e^q}} + \tan(a).$$

As the problem is a symbolic regression, a search algorithm is to determine a symbolic representation of a function that approximates f. Since complete knowledge is given, the component functions of the empirical function are known to a run of the projected search algorithm.

5.3.2 Closure of a run

During a GP run, a component function might take an evolved argument value for that it is not defined. As some component functions of the first empirical function are not defined for each real-valued number, an undesirable behavior of the projected search algorithm may follow. This situation illustrates that, given a grammar G, a sentence $s \in L(G)$ does not necessarily have semantics. Then, the language is "open", i.e., s is open to interpretation. Likewise, we call s an open sentence, else a closed sentence, and defining the semantics of s shall be called closing s.[6] For a GP run r, its property that it produces closed sentences only is called r's closure. In order to support the closure of a run of the projected search algorithm, we use the concept of a protected function as it is standard in GP.

[5] This decision does not render coming empirical results invalid in the context of a dynamic problem: a dynamic problem can be approximated by a sequence of static problems.

[6] For the infix example, the symbol sequence 1/0 is a sentence while the value of the represented mathematical expression is undefined. We close the open sentence by defining that it shall represent the largest positive integer value that is representable in the computing environment.

5.3.3 Closure by protected functions

Division

For protection against division by zero, we define the division function `D(x)` that returns the reciprocal value of its non-zero argument. If $x = 0$, $\mathtt{D}(x) := 1$ while a more appropriate value would be the largest or smallest value representable in the computing environment. The use of such a value, however, is likely to provoke overflow or, respectively, underflow errors during subsequent computation. This situation, in turn, would necessitate the introduction of further protected functions, e.g., a protected multiplication. This again is undesirable as it would violate the principle of minimalism because protection handling consumes computing resources.[7]

A protected function introduces an external bias to a search algorithm.[8] For practical search algorithms, it often cannot be decided theoretically whether an external bias is detrimental or beneficial. This is since, often, the future algorithmic behavior eludes theoretical prediction.[9]

Root, exponential function

We define a protected square root function `sqrt(x)`. It returns the square root of the absolute value of its argument.[10]

Furthermore, we define an overflow-protected exponential function `exp(x)` that returns e^x unless the value of x causes an overflow, in which case the function returns 1.

In extended summary, protected functions have been determined for the first empirical problem. This implies that a produced phenotype is closed, so that a run of the projected search algorithm has closure.

5.3.4 Alphabets

Target alphabet

The empirical problem comes with complete knowledge. Due to the minimalism principle, the target alphabet Ω equals the set of symbols that represent all functions and parameters that are components of the empirical function:

$$\Omega = \{+, *, \mathtt{D}, \sin, \cos, \tan, \mathtt{sqrt}, \exp, (,), m, v, q, a\}$$

[7] This argument also justifies the use of alternative small values besides 1.

[8] This type is opposed to an internal bias that results from evolution alone, and that represents discovered problem knowledge.

[9] The user may try resolving the conflict between the introduction of a potentially detrimental bias and the necessity of a protected function by designing a self-adaptive protected function. This approach, however, results in the conflict of recursion. The situation illustrates a general task for the user of a current practical search algorithm: making an educated guess on the introduction of a bias.

[10] In case of a negative argument value, no imaginary number is produced which would contradict the definition of the empirical function.

For instance, over Ω, the symbol sequence that is structurally identical to the definition of the first empirical function is

$$\sin(m) * \cos(v) * \texttt{D}(\texttt{sqrt}(\exp(q))) + \tan(a)$$

that we call the phenotypic representation of the function.

The more frequently $o \in \Omega$ is contained in an acceptable solution s, the more problem-relevant o shall be called. o is problem-irrelevant if and only if it is not contained in s. (5.4)

The decision on a target alphabet is critical for the performance of a GP algorithm. The size of the solution space of a problem depends on the size of the alphabet (cf. 3.24, p.63). Thus, if the alphabet contains problem-irrelevant symbols, the solution space is unnecessarily large, which is detrimental. It is therefore desirable to determine a small target alphabet as a parameter of a run. However, if the alphabet does not contain all problem-relevant symbols, an acceptable solution may not be located, i.e., the respective run lacks sufficiency (Koza 1992). Thus, Ω must be as small as possible and as large as necessary for sufficiency. We call such Ω an ideal target alphabet. Having complete problem knowledge is sufficient for designing an ideal alphabet. Therefore, for an emergent problem, it is often unknown. Thus, the identification of an ideal alphabet by a self-adaptive GP run shall be investigated later (cf. 5.3, p.108). (5.5)

In summary, we have determined a target alphabet for the first empirical problem. Thus, next, an appropriate source alphabet $\mathbb{A}$ with $|\mathbb{A}| \geq |\Omega|$ (cf. 4.21, p.80) can be defined that is required by the developmental search algorithm. (5.6)

Source alphabet

Due to 5.6, $|\mathbb{A}| \geq 14$ must hold. Following the minimalism principle, the smallest n with $2^n \geq 14$ shall be the size of the source alphabet. $n = 4$ results. $\mathbb{B}^4$ follows as the source alphabet for the first empirical problem:

$$\mathbb{A} = \{0000, 0001, 0010, 0011, .., 1110, 1111\}.$$

Regarding $\mathbb{A}$, a genetic code can be determined (cf. 4.4.1, p.92) as parameter for the developmental search algorithm.

5.3.5 Genetic code

Explicit surjective definition

By definition, a genetic code is a component-value relation cvr_a of a full GP algorithm a, i.e., a mapping from the source into the target alphabet. We explicitly give a genetic code of the first developmental search algorithm in table 5.1, p.112. Following from 4.37, p.85, cvr_a has been designed such that $img(cvr_a) = \Omega$ holds, i.e., cvr_a is surjective, which contributes to the desirable surjectivity of the genotype-phenotype mapping.

Table 5.1: Genetic code of first developmental search algorithm.

0000 +	0001 *	0010 *	0011 D
0100 m	0101 v	0110 q	0111 a
1000 (	1001)	1010 sin	1011 cos
1100 tan	1101 sqrt	1110 exp	1111)

Redundancy

Recall that the redundancy of a genetic code may be beneficial since it supports genetic diversity. The respective discussion has solely focused on the relationship of

i) the redundancy of a genetic code, and

ii) the genetic diversity of a population.

Next, we discuss the relationship of

i) this redundancy, and

ii) phenotypical structure.

Given complete problem knowledge, the structure of the first empirical function is known. We know that, as a function component, multiplication occurs more often than any other binary operator in the target alphabet. Thus, the presented genetic code has been designed for redundancy of the multiplication operator: two different codons represent $*$. This situation increases the probability of producing a phenotype that is structurally close to the empirical function. (5.7)

This is beneficial because the structural identity of a phenotype with the empirical function is sufficient for the phenotype to be perfect. The described situation illustrates that the redundancy of a genetic code influences the performance of the developmental search algorithm, which we follow next. (5.8)

Function redundancy

The redundancy of a function has been defined in a qualitative sense via its injectivity. Let $f : A \to B$ with finite non-empty A and non-empty B be given. f shall be said to have quantitative redundancy r on $b \in B$ with

$$\boldsymbol{red_f(b)} = |f^{-1}(b)||A|^{-1}.$$

$$\sum_{b \in B} red_f(b) = 1$$

follows. Thus, the quantitative redundancy on b is a measure for the fraction of domain elements that f maps onto b. Put vividly, the more domain elements are mapped onto b, the higher is the redundancy on b. (5.9)

If and only if f maps no element onto b, $red_f(b) = 0$. If exactly one $a \in A$ is mapped onto b, $red_f(b) = |A|^{-1}$. If A is mapped onto b, $red_f(b) = 1$. (5.10)

If f is surjective, there is no $b \in B$ with $red_f(b) = 0$. If f is injective, $red_f(b)$ is either $|A|^{-1}$ or 0. Thus, if f is bijective, $red_f(b) = |A|^{-1}$ follows for all $b \in B$.

Next, we employ our concept of the quantitative redundancy of a mathematical function for discussing the relationship between a genetic code cvr_a and the performance of a developmental search algorithm a.

Code redundancy and performance

Let $c = cvr_a$. The higher the quantitative redundancy $r = red_c(o)$ is on $o \in \Omega_a$, the higher is the number of codons encoding o. We call r the code redundancy of c on o. The higher r is, the higher is

i) the probability that a final transcript contains o in n positions
ii) this n.

r strongly influences the probability from i), but it does not solely determine it. **For example**: let $r = 0$ for a given $o \in \Omega$. Then, no codon encodes o (cf. text unit 5.10), so that o is not contained in a primary transcript. However, the employed repairing mode may introduce o into the final transcript. For another example: let $r = 1$. Then, all codons encode o, so that o is contained in a primary transcript. However, the employed repairing mode — e.g., deleting repairing — may reduce the number of instances of o in the final transcript.

If and only if a code has a positive redundancy on a problem-relevant target symbol, it shall be called a beneficial genetic code.

Let a code c be given that maps a source alphabet $\mathbb{A}$. We then call $|\mathbb{A}|^{-1}$ the neutral redundancy of c. If and only if c has its neutral redundancy on a target symbol o, then c shall be called neutral on o. A target symbol with neutral redundancy under c is, regarding the number of its instances in all possible primary transcripts, neither over- nor under-represented. If and only if c is neutral on each $o \in img(c)$, i.e., if and only if c is injective, then we call c a neutral genetic code. Put vividly, c does not favor or neglect any target symbol in its image.

The more the redundancy of a code c on $o \in \Omega$ mirrors o's problem-relevance (s. def.), the likelier a resulting phenotype is structurally close to an acceptable solution. This situation is a beneficial bias (cf. 5.7, p.112). We then call c a mirroring genetic code. Thus, the more frequently a target symbol occurs in an acceptable solution, the more redundant is a mirroring code on this symbol. Especially, given an acceptable solution, a corresponding ideal mirroring code is neutral on those target symbols which are neither rarely nor frequently contained in the solution. The code has a positive redundancy on exactly those symbols whose instances compose the solution. Thus, the redundancy on all other symbols is zero. In particular, in the pathological case that the solution s contains exactly one symbol o—in $|s|$ instances—, the code has redundancy 1 on o.

Eventually, if a code is mirroring, it is beneficial, while a beneficial code is not necessarily mirroring.

In a vivid conclusion, a given code represents a concentration of ingredients in the boiling developing population. This concentration cannot be designed or changed for a *common* search algorithm because it has no genotype-phenotype mapping in the first place. Besides the number of instances of a target symbol in a phenotype, their positions decide on the phenotypic quality, too. Thus, a mirroring code is not sufficient for producing an acceptable phenotype. The distribution of the symbols onto beneficial positions must follow from the dynamics of the developmental search algorithm.[11]

For an emergent problem, one or more beneficial redundancies are usually unknown. Their identification by a GP run is desirable, and we will investigate this issue later (cf. 5.3, p.108). Identifying redundancies represents identifying a relevant target alphabet. This is because a positive redundancy on a symbol s from a large set S of potential target symbols turns s into a target symbol, i.e., there is at least one codon encoding s. Thus, identifying beneficial redundancies on symbols from S supports designing an ideal target alphabet, approaching the conflict discussed at 5.5, p.111.

In summary, we have determined that a genetic code should be mirroring. (5.11)

An unknown beneficial code and target alphabet should be identified by a GP run. (5.12)

Next, the redundancies resulting from the presented genetic code for the first empirical problem shall be discussed.

Empirical-genetic-code redundancy

Since $|\mathbb{A}| = 16$, a redundancy value is a multiple of 16^{-1}, the code's neutral redundancy. For each target symbol, except '$*$' and ')', its redundancy r is neutral because it is encoded by exactly one codon. '$*$' and ')' are encoded by two codons each, resulting in $r = 2 \cdot 16^{-1}$ each. Thus, the redundancy on '$*$' is twice the neutral redundancy, and the reason for this situation has been discussed at 5.7, p.112. The reason for the high redundancy on ')' follows. Seven of the fourteen target symbols, namely

$$\mathtt{D}, (, \sin, \cos, \tan, \mathtt{sqrt}, \exp,$$

require a following-up ')' in order to close an open expression. This problem-unrelated property of the phenotypic representation yields a search bias toward nested phenotypes like

$$\mathtt{D}(m + \sin(v + \mathtt{sqrt}(a * ..)))))))))).$$

[11] This situation reminds us of producing a chemical compound. Redundancies on symbols correspond to concentrations of ingredients, and symbol positions correspond to a critical sequence of adding ingredients to a reactor. This observation indicates combining the fields of Genetic Programming and Artificial Chemistries. For an introduction on the latter subject, see (Dittrich 2000).

The ')'-tail results from the phenotypic size limit that forces a determined repairing mode to finish the open expression (cf. 4.5.2, p.96). Therefore, phenotypes similar to the above structure result from this search bias (cf. 4.42, p.88). Often, the resulting lack of structural diversity is detrimental, unless, by coincidence, the produced structures are similar to an acceptable solution. For the first empirical problem, a perfect solution, e.g., the empirical function, is known. We conclude that i) the produced phenotypes are not similar to a perfect solution and ii) therefore, the described bias is detrimental. Thus, we design the above-neutral redundancy on ')', which raises a counter-bias that supports the production of less nested phenotypes like

$$\mathtt{D}(m) + \sin(v + \mathtt{sqrt}(a)) + .. * q.$$

The described situation is an instance of a generic conflict in cybernetics. A design decision implies biased system behavior. If the bias is detrimental to a designer's objective and if the designer can predict the bias, he or she may design a counter-bias. This bias, however, may be detrimental itself, and so forth. The situation is an instance of the conflict of recursion. Often—e.g., for an environment with unknown dynamics—a detrimental bias of the system behavior is unpredictable. This is a fundamental reason why autopoietic system behavior is necessary, and why an "autopoiesis-giving" subsystem cannot be designed *a priori* and attached to a non-autopoietic system.[12]

In conclusion, problem knowledge can be represented by a genetic code, which may provoke an instance of the conflict of recursion, so that a self-adaptive behavior of a developmental GP run is desirable.

In extended summary, a beneficial genetic code has been determined for the first empirical problem. (5.13)

Determining further parameters of the projected search algorithm continues.

5.3.6 Developmental representation

Genotype size

The size of the phenotypic representation of the first empirical function equals 25. We decide that the genotype size of the empirical developmental search algorithm shall equal this size, that is, a genotype consists of 25 codons. The reason is that, only in pathological cases, the size of a phenotype is very different from the size of its genotype. For instance, a much longer phenotype implies that the developmental mechanism contains problem-specific information that the genotype should represent. Next, the determined genotype size calls for a discussion of the resulting search space.

[12] Rather, an autopoietic system solely creates and maintains itself on the fly when facing an unpredicted situation. Its "service" or "goal-oriented behavior" are merely by-products, or side-effects, while its autopoiesis is its essence that makes it perform "well" in some present situation.

Search space and problem classification

The empirical problem is not a real-world problem because an acceptable solution is known (cf. 2.5, p.33). However, the solution is unknown to instances of the projected search algorithm. In order to complete classifying the empirical problem from the perspective of the developmental instance, we must compute the size of the solution space. The search space of the empirical developmental search algorithm shall also be called the developmental search space.

For the present work, the magnitude's order of space sizes is of interest. We use the $\mathtt{E}\,\boldsymbol{n}$ notation that stands for $*10^n$. There are sixteen different codons in the source alphabet $\mathbb{A}$, and the genotype size g equals 25. Thus, the developmental search space contains

$$|\mathbb{A}^g| \approx 1.3\,\mathtt{E}\,30$$

genotypes (cf. 4.15, p.78). This space represents the solution space of the problem because the genotype-phenotype mapping is a semantic mapping (cf. 3.5, p.55).

In order to perform a conservative problem classification, let a fast implementation of a search algorithm a be given for the problem: let a locate and evaluate a solution in, e.g., $1\,\mathtt{E}{-}12$ seconds.[13] Let a generous run-time period of one year be given, which in itself is already impractical for the majority of problems, anyway. During this period, a run of a locates and evaluates at most about $3.2\,\mathtt{E}\,19$ search points, which is an insignificant fraction of the search space. Thus, from the perspective of the developmental search algorithm, the first empirical problem is a real-world problem.

In conclusion, regarding the technical objective, the problem is valid for empirical research. (5.14)

As we have determined a genotype representation of $\mathrm{d}_{\rightarrow}$, the empirical developmental search algorithm, its search operators can be discussed next (cf. 2.4.1, p.41).

5.3.7 Search operators

Following from text units 4.3, p.69, and 4.1.5, p.70, point mutation is a search operator of $\mathrm{d}_{\rightarrow}$. (5.15)

The objective of the empirical research is the observation the behavior of the projected search algorithm (cf. 4.56, p.105). The behavior strongly depends on the properties of the employed search operators. Thus, observing the influence of different operators is of interest. To that end, we design such operators next. (5.16)

From an atomic perspective, single-bit inversion has been decided on as an operator of the empirical developmental search algorithm (cf. 4.1.4, p.69). However, a genotype

[13] This is especially unrealistic for a GP algorithm since quality evaluation is most time-consuming with these Evolutionary Algorithms. A general, longer argument, making use of the Bremermann limit instead of a concrete time interval, as chosen here, would not yield another classification, so we omit it here.

also represents a codon sequence, so that, from a codon perspective, a codon in a genotype is a "point".[14] Thus, we require point mutators for both perspectives.

Coupled mutator

The coupled mutation operator, applied to a genotype $g = c_0..c_n$, random-determines $c_i = b_0..b_m$ and then inverts random-determined bits $b_j, b_k, j \neq k$. Thus, this operator is a point mutator from the codon perspective, and we call it the coupled mutator. The reason for the chosen design is that a single-bit inversion is equivalent to the point mutator for the atomic perspective. In order to design different operators (cf. 5.16, p.116) and following from the minimalism principle, exactly two bits must be inverted.[15]

The coupled mutator effects the smallest non-atomic modification of a codon. For a codon with size n, there are $n - 1$ non-atomic mutators. For instance, for a codon with size three, there is the coupled mutator as well as a mutator that inverts all three bits of the codon. All $n - 1$ mutators qualify as point mutators for the codon perspective. Next, we design an operator that emulates these mutators for the first empirical problem.

Unrestricted mutator

The unrestricted mutator u, applied to a genotype $g = c_0..c_n$, random-picks $c_i = b_0..b_3$. Four alternative cases occur, and we set the probability p for each case.
i) $p = 0.5$: u random-picks exactly one bit.
ii) $p = 0.35$: u random-picks exactly two different bits.
iii) $p = 0.1$: u random-picks exactly three different bits.
iv) $p = 0.05$: u picks all four bits.

u then inverts the picked bit(s).

The probability distribution is sound as the sum over the probabilities equals one.[16] Following from the distribution, the unrestricted mutator emulates the point mutation for the atomic perspective because it can perform a single-bit inversion. The mutator also emulates all point mutators for the the codon perspective regarding the first empirical problem.

- In extended summary, we have determined search operators of the empirical developmental search algorithm. Properties of a search operator co-determine the dimensionality of a genotype representation, which shall be discussed next.

[14] With the pathological exception of a single-bit codon, these two perspectives are different.

[15] The coupled mutator assumes a codon size of two or more, which is met in the context of the first empirical problem. This mutator is an algorithmic metaphor of the biological phenomenon that some mutations tend to change nucleic acids in a coupled manner.

[16] The distribution reflects the principle of variation that has been identified in the context of natural evolution: a small hereditary genetic change occurs more often than a big change. This is because a big change is more likely to be disruptive — especially, lethal — to offspring, so that the modified genome is unlikely to be reproduced.

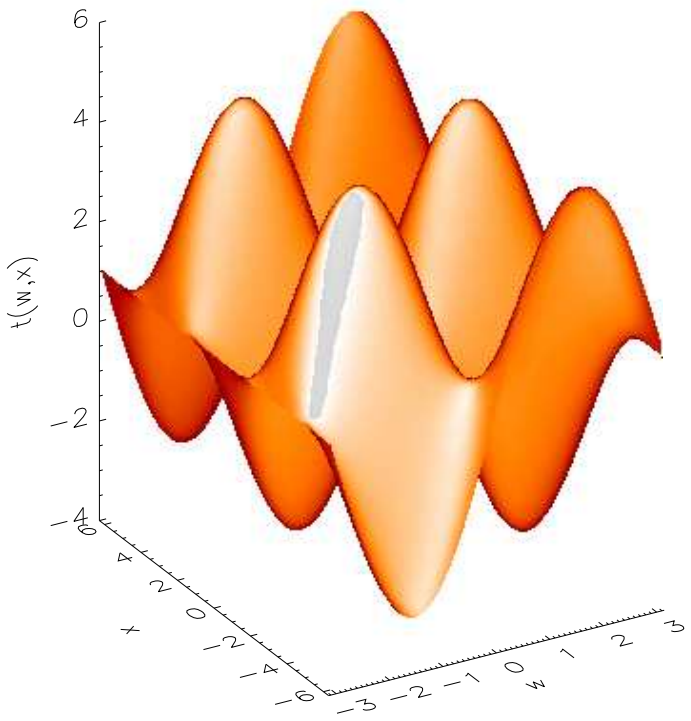

Figure 5.1: An example of a synthetic fitness landscape over a planar solution space generated by dimensions x and w. Quality function t maps the space into fitness values, generating the landscape. Profiles for real-world problems usually do not offer nice properties such as continuity.

5.3.8 Dimensionality of a representation

For a genotype representation and a respective point mutator, a mutable genotype component shall be called a dimension of the representation.[17] For a moving population (cf. 2.4.1, p.41), a dimension corresponds to a direction of a move. (5.17)

The number of dimensions is called the dimensionality of the representation. For instance, for a 3-bit representation and a single-bit point mutator, each bit is a dimension, which gives a dimensionality of three. (Keller and Banzhaf 1996) indicates a relationship between the performance of a search algorithm and the dimensionality of the employed representation. We detail and extend this issue next.

Fitness landscape and performance

A semantic mapping s maps the search space of a search algorithm a onto the solution space. $f \circ s(sea_a)$, the concatenation of a's fitness function f after s, yields a so-called fitness landscape (see Figure 5.1). Thus, there is a bijection between the solution space and the points of the resulting fitness landscape that may be likened

[17] Thus, a dimension represents a behavioral freedom degree of an underlying search algorithm.

to a 3-D profile of a hilly or mountainous real landscape. Following this notion, a higher point in the landscape corresponds to a higher genotypic fitness, representing a higher solution quality. Especially, a peak in the landscape represents a local optimum. Accordingly, one speaks of an appropriate landscape part between peaks as a valley.

For Genetic Programming, a solution space is discrete since an alphabet is discrete (cf. 3.24, p.63). A fitness landscape is discrete because it results from applying a fitness function to the solution space. A landscape is often very rugged, as several contributions emphasize, e.g., (Albuquerque, Chopard, Mazza, and Tomassini 2000; Kinnear, Jr. 1994). This is because a small difference d between two genotypes may result in a large difference between the respective phenotypic behavior and fitness values. As a small d corresponds to a small move in the search space, a large ruggedness results.

A landscape may be visualized in a more abstract manner, e.g., as a 3-D histogram. A column of the histogram represents a genotype, and its height illustrates the respective fitness. (5.18)

Mostly, search spaces of emergent problems are high-dimensional. Thus, the visualization results from an n-to-three-dimensional mapping with a loss of information. In particular, a visualization may appear smooth while the landscape is rugged. Therefore, while a visualization supports human problem understanding, it must be interpreted regarding the problem's actual dimensionality, especially, since a strongly rugged landscape reflects many local optima, and their existence is detrimental to performance. This is because an individual may become trapped on a relatively bad peak (cf. 3.9, p.59), which supports premature convergence (cf. 3.10, p.59). (5.19)

A population moving through a search space (cf. 2.4.1, p.41) gives the notion of this population moving on the corresponding fitness landscape. A subpopulation is a subset of the population of a run of a search algorithm. Depending on the underlying selection algorithm and the neighborhood of a peak P, the following situation may arise. A subpopulation $\mathfrak{S}$ located on or close to P is unable to leave its vicinity by walking (cf. 2.7, p.42). This is because P is circled by a valley, so that an individual walking from P would probably not be selected for reproduction. Thus, the more distant P is from another, potentially better, peak Q, the more unlikely it becomes that $\mathfrak{S}$ locates Q by walking (cf. (Eigen 1992)). (5.20)

Alternatively, an individual from $\mathfrak{S}$ may leave P by leaping (cf. 2.8, p.42). The end point of a leap may have a quality roughly equal to or better than that of P, the start point. In a beneficial case, the end point E equals Q or is close to Q without a dividing valley between E and Q. However, a leap — corresponding to a macro-mutation — from P may be detrimental because it may miss a better point *close* to P that might have been reached by walking. (5.21)

In summary, a rugged fitness landscape is detrimental to performance. This is partially due to its potential for trapping a subpopulation in a local and unacceptable

optimum. Walking and jumping—the only forms of search—in order to escape the optimum may both be detrimental. Thus, next, we consider a beneficial transformation of a landscape.

Distance measure

The notion of distance is at the root of the current discussion (cf. 5.20, p.119). We use the given definition (cf. 2.4.1, p.42) for defining "distance" on a fitness landscape. As a point mutation results in the smallest structure variation, the distance between two solutions p, q shall be the minimal number of point mutations required to transform p into q. This notion is a special case of the edit distance of structures[18] s and t that is defined as the minimal number of point mutations required to transform s into t.

For two binary sequences of identical size, the Hamming distance (s. def.) is an edit distance. For another example, let the symbol sequences $a+b$ and $a*c$ be given. The transformation of a symbol at position i into an arbitrary different symbol at position i is a point mutator that we call single-symbol converter. Then, the edit distance of the given sequences equals two.

Let a distance measure d be given for A, a non-empty set of structures. Then,

$$\max(\{\mathrm{d}(a,b)|a,b \in A\})$$

shall be called the diameter of A, and we denote it by $\emptyset_{\boldsymbol{A}}$. The diameter of a space can be used for assessing whether a given distance $\mathrm{d}(a,b)$ is "small" or "big", i.e., whether a and b are "close" or "far apart". To that end,

$$\mathrm{d}(a,b)\emptyset_A^{-1}$$

is considered, and, thus, the larger the resulting value is, the bigger is the distance, while 0 is the minimum and 1 is the maximum. We call this value the relative distance of a and b. It is undefined for $\emptyset_A = 0$, which is equivalent to A containing exactly one element.

In summary, the edit distance, a formal distance measure for structures, has been identified. Thus, the intended transformation of a landscape can be considered next.

Fitness-landscape transformation

Two different peaks A, B may be adjacent, i.e., $\mathrm{d}(A,B) = 1$. Put differently, there is a connecting trail (cf. 2.4.1, p.42) whose start point is, e.g., A and whose end point is B. Thus, this trail is shortest possible, which is almost ideal since it implies that there is no separating valley. Thus, exactly one appropriate point mutation regarding A locates B. A beneficial transformation of a fitness landscape follows:

[18] This distance measure is discussed and applied in (Keller and Banzhaf 1994) in order to explicitly maintain genetic diversity.

- for each pair of non-adjacent peaks, a shortest connecting trail is introduced, i.e., the peaks are made adjacent. We call this transformation the complete connecting of the landscape.

Topologically, the connecting trail is a saddle between the peaks. Therefore, the transformation turns a peak into a saddle point that we call a transformed peak. Thus, all local optima become saddle points, and a *sole* global optimum, if existing, becomes the only remaining local optimum. Especially, in the transformed landscape, a subpopulation cannot get trapped prematurely in a non-global optimum, which is highly beneficial.

Complete connecting implies that, from each point p in the landscape, there is a short monotonously rising trail leading to a global optimum, as follows:
i) If p is a global optimum, the trail size is zero; else
ii) if p is a transformed peak, p is adjacent to a global optimum; else
iii) p is not a transformed peak:
there is a monotonously rising trail to the closest transformed peak q
because q was a local optimum. From q, ii) applies.

On a completely connected landscape, a hill climber

- that knows[19] among all *rising* trails starting at its position the best one to take

locates a perfect individual with fewest steps, exhibiting best performance. We call this search procedure a directed hill climber. The underlying complete connecting is almost ideal and suggests an ideal transformation:

for each point—not merely for each local optimum—that is not a global optimum, a shortest possible connecting trail to a global optimum is introduced. We call this transformation absolute connecting. On the resulting landscape, a directed hill climber locates a perfect individual in, at most, one step, exhibiting overall best possible performance. (5.22)

This extreme space restructuring emphasizes that some beneficial transformations, in their pure form, are infeasible for real-world problems due to lack of problem knowledge. We therefore focus on complete connecting as a—still unreachable—role model.

Opposed to absolute connecting, complete connecting results in a landscape that is more amenable to visualization. Compared to the original landscape, the completely connected landscape is much smoother because valleys between local optima have been eliminated.[20] Next, we consider the practicality of beneficial transformations.

[19] The knowledge may come from an oracle as known in theoretical informatics.

[20] One may visualize complete connecting as covering the original landscape with a think blanket. A fold of the blanket connecting two peaks represents a saddle.

Increasing dimensionality

Complete connecting links two arbitrary peaks of a fitness landscape by a shortest trail. For a real-world problem, an acceptable local optimum, corresponding to a peak, is unknown. Therefore, complete connecting is infeasible, so that we discuss an approximation. (5.23)

First, let the search space be a subset of the solution space, i.e., we consider a common search algorithm (cf. 3.4, p.54). (5.24)

The core operation of complete connecting is linking two *non-adjacent* peaks by a *two-point* trail. Thus, the trail is no subset of the search space. **For instance**, consider a search algorithm a with a fitness function f. Let there be a symbol set $\{+, -, a, b\}$, a language $L = \{+a, -a, +b, -b\}$, and the single-symbol converter as a point mutator. The resulting two-dimensional search space is $sea_a = L$. We call this scenario the sign example. Let the points $+a$ and $-b$ be peaks with

$$f(-b) > f(+a). \tag{5.25}$$

The peaks are not adjacent. A shortest modification sequence equals

$$[+a] \to [-a] \to [-b],$$

and the only other such sequence is

$$[+a] \to [+b] \to [-b].$$

Thus, there exists no two-point trail in sea_a.

In general, for connecting two non-adjacent peaks by a two-point trail, a transformation of the search space must reduce the peak distance to one. As the distance measure depends on the encoding e of the search space, we discuss changing e into an encoding g. For a set S, let the representation of $i \in S$ under an encoding c be denoted by $\boldsymbol{i_c}$. Let there be non-adjacent representations p_e, q_e of peaks p and q. There are to be adjacent representations p_g, q_g. Thus, a point mutation is to be able to change p_g to q_g. To that end, we add a component to a point representation under e such that mutation can change the component value to a value that results in the representation of q_g. Thus, this procedure creates g by adding one dimension to e, which is the desired change of e (cf. 5.17, p.118). (5.26)

In summary, we have determined a principle of a beneficial fitness-landscape transformation: increasing the dimensionality of the encoding of the search space S. This embeds S into S', called the hyperspace of S that, in turn, is known as a subspace of S'. One may visualize the effect of embedding as bending the original landscape within its hyperspace. Ideally, the bending results in a transformed landscape in that points of interests, e.g., peaks, are closer. Next, we apply the identified principle.

Increasing the dimensionality of the search-space encoding

For the sign example, increasing the dimensionality of the search-space encoding may mean adding one or more symbol positions to a phenotype representation (cf. 5.26, p.122). However, this option is not generally available for a common Genetic-Programming algorithm, due to the hard syntactic constraint (cf. 3.5.2, p.63).[21] Thus, one wants to increase the dimensionality of the encoding e of the search space sea_a while respecting the syntactic constraint. To that end, one may introduce an encoding

$$f : f \neq e$$

of sea_a which is a subset of the solution space sol_p (cf. 5.24, p.122) which is encoded by e. Therefore, under f, sea_a is *not* a subset of sol_p. It follows that, for $s \in sea_a$, a function m must map s_f onto s_e that can be executed for quality evolution. Note

$$f \neq e \Rightarrow m \neq id.$$

- Thus, the described situation is equivalent to a genotype being mapped onto its phenotype, and the employed search algorithm is *developmental*. We can therefore use results on genotype-phenotype mapping (cf. 3.3, p.54). Especially, the required new search-space encoding f shall be binary (cf. 4.1.2, p.68) and have a higher dimensionality than the encoding e of the solution space. f represents the search space of an underlying developmental search algorithm.

In summary, we have determined a principle of increasing the search-space dimensionality in compliance with the syntactic constraint: one supplies a higher-dimensional encoding of the solution space. The according representation of the solution space is a developmental search space. Thus, a developmental search algorithm implies the potential of a beneficial landscape transformation. (5.27)

- This is a primary reason for using artificial ontogeny as component of a search algorithm. In particular, the mentioned principle detaches the landscape from the solution space whose fitness topology is frozen since the semantics of the employed target language is static. Instead, the principle attaches the landscape to a developmental search space which is only indirectly linked to the target language by means of a semantic mapping. Thus, the semantics of a point in this space may vary, which implies that the landscape may change.

Example

We use the sign example. The dimensionality of the given solution-space encoding equals two, and a binary search-space encoding of higher dimensionality is wanted. The target alphabet contains four symbols. Thus, the source alphabet must contain, at least, four codons, so that, e.g., the genetic code in table 5.2, p.124, may be given. (5.28)

Table 5.2: Genetic code of sign example.

00	+
01	a
10	-
11	b

Following 5.3.6, p.115, the symbolic genotype size shall equal two. Thus, the atomic genotype size equals four because each codon consists of exactly two bits. Therefore, the search-space dimensionality[22] equals four since single-bit inversion is the point mutator of a binary representation. It follows, as intended, that the search-space dimensionality is higher than the solution-space dimensionality.

The search space represents the solution space through a genotype-phenotype mapping. In order to fully determine such a mapping for the sign example, we must give a repairing mode (cf. 4.2.12, p.82). Deleting repairing shall be used, following the recommendation at 4.43, p.89. Replacing finalizing shall be used, following the recommendation at 4.5.2, p.98. Appending finalizing shall be used in situations where replacing finalizing is not applicable.[23] The determined scenario shall be called the four-dimensional sign example.

Consider the genotype 1111 that gives the primary transcript bb. Deleting repairing yields the transformation sequence

$$\mathsf{bb} \rightarrow \mathsf{b} \rightarrow \epsilon,$$

yielding ϵ as an irreducible transcript which requires finalizing. As replacing finalizing is not applicable since there is no symbol to be replaced, appending finalizing continues the transformation sequence and delivers

$$\epsilon \rightarrow + \rightarrow +\mathsf{a}.$$

Thus, 1111 represents the phenotype +a, i.e., one of the given peaks. Let a point mutator change the considered genotype 1111 into 1011 that gets transcribed into the primary transcript, phenotype and peak −b. Thus, the peaks are adjacent in the search space while they are not in the solution space. This illustrates the effectiveness of increasing the search-space dimensionality (cf. 5.3.8, p.123). A discussion of its feasibility (cf. 5.23, p.122) follows.

[21] For instance, for the sign example, appending a position results in potential solutions of size three, e.g., "$-ba$". As such a solution is infeasible, it cannot be evaluated.

[22] This expression shall be equivalent to the exact but cumbersome term "dimensionality of the search-space encoding."

[23] This is because the use of deleting finalizing is not an option: the symbolic genotype size is two and the size of a sentence of the target language is also two. Thus, the size of the primary transcript is two and the size of the final transcript, i.e., a sentence, must also be two, so that deleting a trailing symbol is no option.

5.3.9 Feasibility of increasing dimensionality

The sign example is simple, open to manually conducting experiments and thought experiments. Especially, one can enumerate and visualize the search space and solution space.

Binary graph and search space

The four-dimensional binary search space $\mathbb{B}^4$ is amenable to visualization. A four-dimensional cube, i.e., a tesseract, can be visualized by a three-dimensional projection: a cube within a cube, the cubes aligned, and corresponding cube corners connected by edges.

The number of corners of an n-dimensional cube, $0 \leq n$, equals 2^n. The 2^4 corners of a tesseract correspond to the 16 elements of $\mathbb{B}^4$. We give a corner-labeling algorithm c that defines a bijection between the corners of an n-dimensional cube and the search space $\mathbb{B}^n$, as follows for the example of a tesseract.

---

c labels an arbitrary origin corner with 0000. This corner has edges connecting it to other unlabeled corners. c labels these corners, in arbitrary order, with unused labels

$$b \in \{x \in \mathbb{B}^4 | hd(x, 0000) = 1\} = \{1000, 0100, 0010, 0001\}.$$

This labeling step is iterated recursively for each newly labeled corner as an origin corner until all cube corners are labeled.

---

c implies that two corners that are connected by an edge have binary labels whose Hamming distance equals one. Thus, a point mutation of a corner label l to a label m corresponds to a walk from a space element to an adjacent element. Therefore, the search space $\mathbb{B}^n$ can be modeled as a graph whose nodes represent the corners of an n-dimensional cube and whose edges represent the cube edges. We call this graph an n-dimensional binary graph. The four-dimensional binary graph can be visualized as a wire-frame model of the mentioned three-dimensional projection of a tesseract.

- In summary, an n-dimensional binary graph models the search space $\mathbb{B}^n$. Next, we use this model for discussing the relationship of i) dimensionality and ii) genetic diversity and neutral networks (cf. text units 3.11, p.59, 4.45, p.90, 3.8, p.58, 3.13, p.59).

Redundancy, diversity, and neutrality in search space

The four-dimensional sign example results in the following list of genotypes, primary transcripts, and phenotypes. Thus, the list represents an explicit genotype-phenotype mapping. (5.29)

```
0000 ++ +a    0001 +a +a    0010 +- +b    0011 +b +b
0100 a+ +a    0101 aa +a    0110 a- -a    0111 ab +a
1000 -+ -a    1001 -a -a    1010 -- -b    1011 -b -b
1100 b+ +a    1101 ba +a    1110 b- -a    1111 bb +a
```

While the solution space and target language contains four elements, namely,

$$sol_p = \{+a, -a, +b, -b\},$$

$$|sea_a| = |\mathbb{B}^4| = 16$$

holds for the search space.

In general, a difference in space sizes depends on the dimensionality of sea_a: the size of a structurally unconstrained space depends exponentially on its dimensionality (cf. text units 4.1.2, p.68, 3.24, p.63). When $|sea_a| > |sol_p|$, the genotype-phenotype mapping must be redundant which is beneficial (cf. 4.3.3, p.91). In the four-dimensional sign example, the mapping is highly redundant on $+a$, for instance. This redundancy solely results from the employed repairing type, since the given genetic code (cf. 5.28, p.123) is non-redundant (cf. 4.46, p.90). (5.30)

Even if a search space is not larger than the accompanying solution space, a genotype-phenotype mapping can still be redundant on a subspace of the solution space. For instance, for another variant of the sign example, the above sol_p could be a subset of a larger target language. This is an argument for the evolution of genetic codes, because it necessitates the evolution of redundancies (cf. 5.12, p.114) regarding different sublanguages, as follows. (5.31)

A population, moving through search space, represents, by use of a genotype-phenotype mapping m, a dynamic subset of the solution space. Thus, in order to stay beneficially redundant, m must be dynamic, too. As a genetic code co-determines m, the co-evolution of codes and genotypes can imply such an adaptation of m.[24] A corresponding developmental search algorithm would experiment with different landscapes, molding them into beneficial shapes. We shall provoke and investigate this situation later.

The sign example also illustrates the beneficial phenomenon of percolating real neutral networks (cf. 3.18, p.60) and its connection to dimensionality and diversity. To that end, we model a neutral network of a phenotype p as a subgraph of the binary graph $\mathbb{B}^n$:

$$S = \{x \in \mathbb{B}^n | x \in m^{-1}(p)\},$$

i.e., the nodes of $\mathbb{B}^n$ that represent p's genotypes.

[24] One may imagine a population moving through search space and watching the solution space through a collection of differently warped windows, or through a single warping window. The window represents an adapting mapping, giving a changing fitness landscape (cf. 5.27, p.123).

We call S the binary neutral network of p. For instances of such networks, consider the genotype-phenotype mapping m of the four-dimensional sign example (cf. 5.29, p.125). For example, the binary neutral network of $+a$ equals

$$N = \{0000, 0001, 0100, 0101, 0111, 1100, 1101, 1111\}.$$

N is an instance of a connected graph G, i.e., for nodes a, b of G, there is a path connecting a, b that is a subgraph of G. Thus, starting at a node of N, each other node of N can be reached by a sequence of point mutations that do not leave N. Especially, N is a cycle percolating through the search space, given in a short notation by

$$0000, 0001, 0101, 0111, 1111, 1101, 1100, 0100, 0000.$$

When visualizing the search space as a tesseract, N is a closed sequence of cube edges.

- In general, connectivity, as we call the property of being connected, of a binary neutral network always allows for finding a previously unreal node of the network without the need for introducing a macro-mutation.

A network that contains more than one element represents genetic diversity. There is a beneficial duality of such diversity (cf. text units 3.12, p.59, 3.15, p.60). On the one hand, high genetic diversity fosters exploration of the search space sea_a. On the other hand, it is necessary for neutral networks percolating through sea_a, and we illustrate this second aspect. Let t be a trail in sea_a that connects the genotypes g, h. Let t be shorter than the shortest trail in sol_p that connects the phenotypes p_g, p_h. Then, we call t a tunnel. A mutation from a genotype in t to another genotype in t shall be called a tunnel effect.

For the sign example, the binary neutral network of peak $-\mathtt{b}$ is $N = \{1010, 1011\}$ (cf. $+\mathtt{a}$'s network above). The genotypic distance

$$\mathrm{hd}(1011_{-\mathtt{b}}, 1111_{+\mathtt{a}}) = 1$$

is minimal. However, the respective minimal phenotypic distance equals two.[25] Thus, the trail $t = (1111, 1011)$ is a tunnel, and a point mutation changing one genotype in t into the other one is a tunnel effect. (5.32)

This is an extreme instance of text units 3.15, p.60, and 3.17, p.60: neutral networks of phenotypes may percolate through search space such that a small mutation may free a trapped subpopulation from premature convergence.

For the sign example, a beneficial point mutation from $1111_{+\mathtt{a}}$ to $1011_{-\mathtt{b}}$ locates a better peak in sea_a. In sol_p, however, either i) a series of two beneficial point mutations or ii) one equivalent macro-mutation would have to locate $-\mathtt{b}$.

In sea_a, the probability of the above beneficial point mutation equals 4^{-1}. We calculate the corresponding probability for sol_p under the point-mutation series from i). Thus, the structure to be changed is $+\mathtt{a}$, the worse phenotype.

[25] Two is even the maximum of all phenotypic distances, because it is the diameter of sol_p.

First, point mutation changes one of the two positions, e.g., position zero that contains target symbol '+', with probability 2^{-1}. Second, selecting the resulting offspring $-\mathtt{a}$ for a mutation occurs with a probability we denote by $\boldsymbol{p_{sel}}$. $p_{sel} < 1$ holds for a population size greater than one because $-\mathtt{a}$ is no peak. Third, in $-\mathtt{a}$, point mutation changes the position that contains 'a' with probability 2^{-1}. Thus, for sol_p, mutation from $+\mathtt{a}$ to $-\mathtt{b}$ has probability

$$2^{-1} \cdot p_{sel} \cdot 2^{-1} = 4^{-1} p_{sel}$$

which is less than the corresponding probability for sea_a. Therefore, locating the better peak is more likely in sea_a, so that the situation is more beneficial for a developmental than a common search algorithm. Regarding the peaks, this is reflected by their relative distance of 4^{-1} in sea_a vs. 1 in sol_p.[26]

If the dimensionality of a search space is higher than that of the solution space, we call the former high-dimensional. The four-dimensional sign example illustrates the principle behind high-dimensional neutral networks: points of interest, e.g., a good genotype g and a better genotype h, may be significantly closer than p_g and p_h. In that case, a search algorithm may locate a trail from g to h with higher probability. We call this beneficial principle the tunnel principle. It is most effective when the situation described at 3.15, p.60, holds: then, short monotonously rising trails go from a space point to a better one.

- Summarizing, the tunnel principle is an argument in favor of the use of a developmental search algorithm. This is because, for a common search algorithm, a genotype equals its phenotype (cf. 3.3, p.54), so that the principle is inoperational.

The illustrated beneficial phenomena resulting from a high-dimensional search space are independent from the four-dimensional sign example. Thus, they occur in large high-dimensional search spaces and the associated large solution spaces of emergent problems. It is there where they are most significant, due to the large diameters of the spaces: phenotypes may have a large relative distance in the solution space while their genotypes may have a tiny relative distance in the search space.[27]

- In summary, a high-dimensional search space supports the *redundancy of a genotype-phenotype mapping*. This redundancy is sufficient and necessary for a neutral network which fosters genetic diversity and may implement the tunnel principle that approximates beneficial landscape transformations. (5.33)

[26] In sea_a, the peaks are closest possible, while, in sol_p, they are located at opposite ends of the space.

[27] In the extreme case, phenotypes may have a distance equal to the space diameter while their genotypes have a Hamming distance of one. This translates into a relative distance of one and virtually zero, respectively.

Redundancy of the genotype-phenotype mapping

A redundant genetic code, an appropriate repairing type, or their combination may result in a redundant genotype-phenotype mapping (cf. 4.46, p.90). For the four-dimensional sign example, redundancy solely results from the employed repairing type, since the genetic code is non-redundant (cf. 5.30, p.126). In general, redundancy-based beneficial phenomena (cf. 5.33, p.128) can be amplified by employing or evolving a redundant code (cf. 5.31, p.126).

- We focus on the evolution of codes, while the evolution of repairing types is further work. (5.34)

Authors discuss high-dimensional phenomena for different applications of computer science, e.g., (Gordon 1994), emphasizing positive implications. The involved principles are medium-independent and therefore of interest to many fields, in particular physics, chemistry, and biology. A lesson learned there is that beneficial results do not come without an investment, e.g., energy, space, or time. Here, one must take care for obtaining a beneficial redundant code for a high-dimensional search space, as follows.

The more redundant a code is to be on a given target alphabet Ω, the more codons there must be for each $s \in \Omega$ (cf. 5.9, p.112). As $|\Omega|$ is fixed, the size of the code domain—the source alphabet $\mathbb{A}$—must be increased. Thus, the codon size c must be increased, since $|\mathbb{A}| = 2^c$ holds. A greater codon size results in a greater atomic genotype size, because a genotype is a sequence of a fixed number of codons. This is equivalent to an increase of the dimensionality of the search space sea_a.

For a space S, we denote its dimensionality by $\mathbf{dim}(\boldsymbol{S})$. The increase of $\mathsf{dim}(sea_a)$ amplifies sea_a's property of being *high-dimensional* (cf. def.), which may be beneficial. However, a larger genotype size consumes more memory, which is detrimental but unavoidable. Also, an increase of $\mathsf{dim}(sea_a)$ amplifies the space's potentially detrimental properties, as follows.

Adding a dimension is equivalent to adding a direction to sea_a. Thus, on the resulting landscape, a knowledgeable search algorithm can create a tunnel effect (cf. 5.22, p.121). However, as knowledge of an emergent problem is incomplete, adding a dimension may decrease the probability of selecting a beneficial direction. As another facet of this conflict, increasing $\mathsf{dim}(sea_a)$ increases

$$|sea_a| = 2^{\mathsf{dim}(sea_a)}.$$

The increase is beneficial or detrimental depending on the ratio r of the number of acceptable to all genotypes in the enlarged sea_a. We call r the acceptability ratio of sea_a. For a small problem p, computing r is feasible, since the fitness evaluation of *each* genotype $g \in sea_a$ is feasible. We also define the term for the solution space sol_p.

Increasing $\mathsf{dim}(sol_p)$, i.e., increasing the maximal size of a phenotype, further increases the large subset of unacceptable phenotypes. Due to combinatorial explosion, this increase is exponential over $\mathsf{dim}(sol_p)$ (cf. 3.25, p.64). A corresponding

increase of computing resources as a counter-measure is infeasible. In the field of optimization, this situation is well known as the curse of dimensionality.[28]

- Summarizing, one can state that increasing the dimensionality of the solution space is always detrimental once the maximal phenotype size allows for representing an acceptable solution.

However, for a search space sea_a, it depends on the employed genotype-phenotype mapping whether increasing $\mathsf{dim}(sea_a)$ is detrimental or a boon of dimensionality, as follows. (5.35)

The redundancy of the used code is critical: i) positive redundancy on a problem-relevant symbol s is beneficial, while ii) it is detrimental on a problem-irrelevant s (cf. 5.4, p.111). This holds since i) supports the acceptability ratio, while ii) weakens it. (5.36)

In the ideal case, a code c has a positive redundancy on exactly all problem-relevant symbols. We call such c a beneficially redundant genetic code, and it is beneficial (s. def.).[29] The defined property is absolute in that a code does or does not have it. However, an absolute notion might be incompatible with the philosophy of Evolutionary Computation that is not based on crisp views. Thus, gradations of beneficial redundancy of c shall be permissible. Intuitively, c is more or less beneficially redundant depending on whether it maps the source alphabet more or less on problem-relevant symbols. Therefore, c may become more beneficially redundant as follows. (5.37)

i) If c has code redundancy $r = red_c(s) = 0$ on a problem-relevant $s \in \Omega$, one increases r, which introduces s to the underlying search process.

ii) If c has redundancy $r > 0$ on a problem-irrelevant s, one decreases r, which assigns a lesser importance to s during search. (5.38)

We implement i). $\mathbb{A}$ is a codon space with $\mathsf{dim}(\mathbb{A}) = |a|, a \in \mathbb{A}$, since the codon size $|a|$ is identical over $\mathbb{A}$. A genotype $g \in sea_a$ is a fixed-size sequence of n codons, so that

$$\mathsf{dim}(sea_a) = n \cdot \mathsf{dim}(\mathbb{A}).$$

Therefore, one can increase $\mathsf{dim}(sea_a)$ by increasing $\mathsf{dim}(\mathbb{A})$. This enlarges $\mathbb{A}$, due to

$$|\mathbb{A}| = 2^{\mathsf{dim}(\mathbb{A})}.$$

Thus, for a code c and a problem-relevant $s \in \Omega$,

$$c^{-1}(s) \subset \mathbb{A}$$

[28] For instance, a search algorithm that exhibits an acceptable performance for a given problem size may not do so for a slightly larger instance of the same problem.

[29] The inversion does not always hold, since a beneficial code may also have a positive redundancy on a problem-irrelevant symbol. Furthermore, a mirroring code is beneficially redundant, while the inversion does not always hold.

may be larger with c being more redundant on s, which especially supports the above option i). In particular, with more codons at hand, finer redundancy gradations can exist because $|\Omega|$ is fixed. Especially, the mirroring quality (s. def.) of a code can be tuned.

For instance, given a surjective code that maps $\mathbb{A} = \mathbb{B}^2$ onto $\Omega, |\Omega| = 4$, the only code-redundancy value occurring is 4^{-1}. However, for $\mathbb{A} = \mathbb{B}^3$, more values occur, e.g., 1/2, 1/4, 1/8, when mapping

— four codons onto the first target symbol,
— two further codons on the second symbol, and
— one further codon each on the remaining two symbols.

- Summarizing, increasing search-space dimensionality may result in a genetic code that is more beneficially redundant or even more mirroring. Such a code results in an increased acceptability ratio, which may supply more beneficial space directions to the next mutation event. In particular, the larger the code redundancies are on all problem-relevant symbols, the higher is the acceptability ratio, which is beneficial. For an extreme toy example, a code results in a genotype-phenotype mapping whose image only contains a global optimum. (5.39)

However, a symmetrical argument holds for the detrimental case that a code has large redundancies on problem-irrelevant target symbols, due to incomplete problem knowledge. In conclusion, increasing search-space dimensionality may result in beneficial or detrimental phenomena, depending on the code, which calls for code evolution that could realize the potential of a high-dimensional search space of supporting a beneficially redundant genotype-phenotype mapping. (5.40)

In extended summary, such a mapping may support—preferably connected—binary neutral networks along which a population explores the landscape using tunnels. The tunnel principle approximates ideal fitness-landscape transformations. Also, neutral networks implicitly support genetic diversity. Thus, unlike a common search algorithm, the developmental population can have diversity in search space *and* convergence in solution space simultaneously.[30] We have demonstrated such beneficial developmental phenomena for the four-dimensional sign example, a synthetic quasi-minimal problem. As their occurrence is independent from space sizes, they can show in the large spaces of emergent problems. (5.41)

However, due to the curse of dimensionality, the critical increase of $\mathsf{dim}(sea_a)$ may also result in detrimental phenomena, depending on the employed genetic code. Therefore, a developmental search algorithm should co-evolve genotypes and genetic codes to the end of a beneficial fitness-landscape transformation. (5.42)

Eventually, a problem-relevant target symbol s is an atomic phenotypic building block. A code that is redundant on s supports the proliferation of s-instances in

[30] This situation is no free lunch — especially, eating *and* keeping the cake — since one invests an ontogenetic apparatus.

a population. Thus, an appropriately adapted code together with point mutation and a preserving repairing type (s. def.) could foster the combination of desirable building blocks.

- In conclusion, we shall investigate code evolution later.

We have identified two performance-critical phenomena for a developmental search algorithm: i) the acceptability ratio of the search space, and ii) the distribution of acceptable and unacceptable points in a fitness landscape, determining its topology. We discuss the interdependence of both issues next.

Genotype-phenotype mapping and developmental causality

The strong causality of a run of a search algorithm with fitness function f is the property that a small change from a genotype g into h results in a small $|f(g)-f(h)|$. For brevity, we shall call this property causality. One has argued that causality is advantageous for the convergence properties of an Evolutionary Algorithm (Rechenberg 1994).

For a small area of a fitness landscape, strong causality is equivalent to a smooth area. (5.43)

This holds because strong causality is equivalent to a small gradient γ of the area, with

$$\gamma = |f(g) - f(h)|\, \mathrm{d}(g,h)^{-1}.$$

Thus, strong causality is desirable because a rugged landscape is detrimental (cf. 5.19, p.119). Especially, strong causality and non-elitist selection support a subpopulation $\mathfrak{S}$ in locating an optimum through point mutation. This holds because point mutation yields a small $\mathrm{d}(g,h)$, so that $|f(g) - f(h)|$ is small.

Then, through point mutations for the better case, $\mathfrak{S}$ approaches a close optimum. In the worse case, $\mathfrak{S}$ may walk into a close valley. However, due to the small quality loss, the decrease in the reproduction probability (cf. 5.20, p.119) may be small. Thus, $\mathfrak{S}$ may cross the valley and reach another peak. Therefore, causality is beneficial because it increases the probability of *walking* to a better peak. While leaping is an alternative to walking, it is potentially detrimental (cf. 5.21, p.119).

- Summarizing, strong causality is beneficial because it fosters search progress by point mutation. It thereby also supports the independence of the search algorithm: point mutation does not need external control, while a macro-mutation does, e.g., a step-size information. Thus, strong causality contributes to the strategic objective. Therefore, next, we discuss increasing causality for a run of a search algorithm, and, equivalently, smoothing the fitness landscape (cf. 5.43, p.132).

Given a fixed acceptance value, increasing the acceptability ratio implies transforming the landscape. An extreme genotype-phenotype mapping of a developmental search algorithm a projects all genotypes onto a perfect phenotype $\mathtt{p}$, i.e.,

$$gp_a : sea_a \to \{\mathtt{p}\}.$$

The resulting landscape is absolutely smooth. We call the transformation landscape leveling. In order to approximate it, a must evolve beneficial codes, since a code co-determines gp_a. (5.44)

We suggest an alternative smoothing principle that does not change the acceptability ratio. Hill building rearranges the columns (cf. 5.18, p.119) that represent genotypes such that the landscape turns into a single hill. We call the result an ideal fitness landscape, because it corresponds to a problem whose perfect solution is located by a hill climber which is a simple search algorithm. In particular, the landscape is smoother than the original.[31]

As the location of a genotype in search space is fixed, hill building implies reassigning quality values to genotypes. This means that genotypes are re-mapped onto other phenotypes such that the ideal landscape results. Thus, for a developmental search algorithm $d_{\rightarrow}$, hill building requires a change of the genotype-phenotype mapping $gp_{d_{\rightarrow}}$, which *again calls for evolving codes.*

We give an **example for the effect of different codes** on landscapes. The four-dimensional sign example with the given code results in the beneficial situation that the peaks $+a$ and $-b$ are connected by a tunnel (cf. 5.32, p.127). A different code may result in an even higher redundancy of $gp_{d_{\rightarrow}}$ on the peak $+a$ while, however, there may be no tunnel leading to a better peak. For an extreme instance, a code c maps all source symbols onto the same target symbol '+', i.e.,

$$img(c) = \{+\}.$$

$$img(gp_{d_{\rightarrow}}) = \{+a\}$$

follows. A smooth and level landscape results, but no genotype represents the better peak $-b$ (cf. 5.25, p.122), so that $d_{\rightarrow}$ cannot locate it. If evolving a code is feasible, it may produce another code that results in a more beneficial genotype-phenotype mapping. (5.45)

- Summarizing, the ideal methods of landscape leveling and hill building increase strong causality. Approximating them for an emergent problem requires evolving a genetic code.

We call a process that increases causality smoothing. Due to the often non-linear nature of an emergent problem, a given genetic code may smooth a local landscape area and roughen another one. The manual design of a code that yields smoothing of the entire fitness landscape is infeasible due to the large size of the search space and incomplete problem knowledge. For instance, one cannot build a hill because, by the nature of the problem, an acceptable peak is unknown.

Opposite to such ideal and global smoothing, local smoothing at a genotype $g \in pop_{r,t}$ of a run r at time t is feasible. In particular, the smoothing of a landscape area

[31] The tunnel principle may be involved, since hill building may move formerly distant landscape points close together.

remote from g is irrelevant if point mutation is the only variation source: then, a remote genotype h cannot become g's recombination partner and influence performance.

However, at a point $g \in sea_a$, a single code may result in some smoothly ascending directions, while others are rugged. The strength of causality at g depends on dimensionality $\mathsf{dim}(sea_a)$: more directions available at g increase the probability that one leads to a much better or worse neighbor.

- Therefore, the notion of local smoothing argues for evolving *individual* genetic codes that co-determine individual genotype-phenotype mappings.[32] We shall therefore discuss this issue later in the present work. (5.46)

Here, we continue with the issue of dimensionality for the first empirical problem.

5.3.10 Dimensionality of the developmental representation

The unrestricted mutation operator changes a single bit of a genotype independently from another bit. Thus, given the determined codon size of four bits and genotype size of 25 codons, the empirical developmental search algorithm $\mathbf{d}_{\rightarrow}$ faces 100 dimensions under this mutator type. The resulting search space of the first empirical problem p has a size of $|sea_{\mathrm{d}_{\rightarrow}}| = 2^{100} \approx 1.3\,\mathtt{E}\,30$ genotypes. (5.47)

In extended summary, this concludes the definition and analysis of the instance of $\mathrm{d}_{\rightarrow}$ that is to run on p. Next, we must determine the instance of the empirical *common* search algorithm $\mathbf{c}_{\rightarrow}$ that is to run on p.

5.3.11 Common representation

Common genotype size

- For $\mathrm{c}_{\rightarrow}$, we set desired genotype size: 25.

This is because the genotype size of the empirical developmental search algorithm equals 25 and both search algorithms are to be compared (cf. 3.6, p.56). Thus, each genotype—which also is a phenotype—of $\mathrm{c}_{\rightarrow}$ is a target-symbol sequence of, *at least*, 25 symbols: if creation produces an irreducible symbol sequence s, the size of a genotype derived from s can be greater than the desired size (cf. 4.51, p.102). As an extreme example, consider the genotype

```
D(D(D(D(D(D(D(D(D(D(D(D(D  (a))))))))))))).
```

Prior to the blank space in the representation, there are 25 symbols. The determined finalizing (cf. 4.52, p.102) of this irreducible subsequence requires appending 15 symbols in order to yield a genotype.

With the genotype size given, we can discuss the resulting search space of $\mathrm{c}_{\rightarrow}$.

[32]Local smoothing for an individual is like a vehicle in rugged terrain that builds a road by moving on it, similar to a state-of-the-art automatic railroad construction unit.

Common search space

We call $sea_{c_\rightarrow}$ the common search space. For the target alphabet, $|\Omega| = 14$ holds (cf. 5.3.4, p.110). For Ω^{25}, i.e., $\{w \in \Omega^* \mid |w| = 25\}$, we find

$$s = |\Omega^{25}| = 14^{25} \approx 4.5\,\mathrm{E}\,28.$$

However, $|sea_{c_\rightarrow}| \ll s$ holds for the following reason. Many $w \in \Omega^{25}$ are not the beginning of a sentence, i.e., $w \notin sea_{c_\rightarrow}$ (cf. 3.5.3, p.64). Each other $w \in \Omega^{25}$ yields exactly one sentence, since finalizing is deterministic.

As the safe single-symbol conversion is the determined point mutator of $c_\rightarrow$, we get $\mathsf{dim}(sea_{c_\rightarrow}) = 25$ (cf. 4.53, p.103).

5.3.12 Common v developmental representation

The developmental search space $\mathbf{sea_{d_\rightarrow}}$ (cf. 5.47, p.134) is high-dimensional compared to the common search space $\mathrm{sea}_{c_\rightarrow}$. Also,

$$|\,\mathrm{sea}_{d_\rightarrow}\,| \gg |\,\mathrm{sea}_{c_\rightarrow}\,|$$

holds by at least two magnitude orders. (5.48)

A high-dimensional search space can support a beneficially redundant genotype-phenotype mapping (cf. 5.40, p.131). Thus, for the first empirical problem, such a mapping may exist for $d_\rightarrow$. Comparing both empirical search algorithms, a better performance is indeed to be expected for $d_\rightarrow$, since we have determined a genetic code with beneficial properties (cf. 5.13, p.115). (5.49)

However, this hand-designed code is only slightly redundant, and $\mathrm{sea}_{d_\rightarrow}$ will not adapt it. In conclusion, we predict a $d_\rightarrow$ performance that is only slightly higher than the $c_\rightarrow$ performance. (5.50)

This prediction follows from the general analysis of the beneficial vs. detrimental potential of a high-dimensional search space (cf. 5.35, p.130). Put vividly, high-dimensional effects may be a curse and a boon for the same run of a search algorithm. Biasing this situation favorably is often hard for emergent problems due to incomplete knowledge. Thus, manually designing a desirable code requires educated guessing. If a mostly detrimental code results, the potential boon turns into an actual curse of dimensionality: the large space size, coming from high dimensionality, does not represent numerous smooth saddles but rugged peak-valley constellations. Regarding the resulting genotype-phenotype mapping, a designed code may lose beneficial potential and may introduce a detrimental situation. This risk is an essential reason for evolving genetic codes. (5.51)

We have given an **example of a genotype-phenotype mapping with both nice and undesirable properties**: the manually designed code (cf. 5.3.5, p.111) is redundant on the multiplication and the closed-paranthesis operator, which may be

beneficial and detrimental. On the one hand, redundancy on multiplication results in more such operators as components of evolved phenotypes. This is desirable since the problem function features two instances of the operator. On the other hand, redundancy on the closed parenthesis is detrimental if many good subexpressions are long: their synthesis becomes unlikely.

- In extended summary, representations and search operators of both empirical search algorithms have been determined. Next, we fix representation-*independent* experimental parameters of the projected search algorithm which emulates both algorithms.

5.3.13 Experiment parameters

Problem instance

Following from 5.2.4, p.109, we represent the first empirical problem $\mathbf{P_1}$ as a set of fitness cases. Due to the real-valued four-dimensional parameter space of the empirical function (cf. 5.3.1, p.109), a fitness case consists of

- four real input values and
- one real output value.

To the empirical developmental search algorithm $d_{\rightarrow}$, P_1 is a real-world problem (cf. 5.14, p.116). Also, as many real-world problems are emergent, incomplete knowledge is essential for experiments regarding the technical objective. In order to give little knowledge about P_1 to an empirical search algorithm, we only offer it

- 10 fitness cases.

The situation is similar to optimizing an emergent problem in an industrial context: often, merely scarce problem data is available due to technical and psychological reasons. Prominent examples are measuring-difficulties and paranoid customer behavior because of a competitive business environment.

In this context, for further reference, we mention the relevance of a small difference in search performance. For i) mass production, ii) an expensive manufacturing process, or iii) a non-linear phenomenon during production, a small improvement in solution quality implies a significant economic advantage. (5.52)

Size parameters

For an experimental run of the projected search algorithm $\mathbf{p}_{\rightarrow}$, we set a

- fixed population size: *500* individuals (cf. 4.54, p.104).

This size results from the experience in Evolutionary Algorithms that a large search space speaks against using a population with few individuals. Facing a large space, even a population with a size several magnitude orders greater than here represents but a negligible space fraction. However, "critical mass" seems to exist in a population whose size ranges in the hundreds and greater. Here, to emphasize the real-world character of P_1, we have chosen a population size at the low end of this range.

If there is a fixed number n of generations that a run r of a generational EA must compute, then we call n the run size of r. For a $p_{\rightarrow}$ run, we set

- run size: *50* generations (cf. 4.55, p.105).

A set E of equally sized runs of an EA $a_{\rightarrow}$, with the runs applied to the same problem, shall be called an experiment with $|E|$ as experiment size. E can be viewed as a virtual run of $a_{\rightarrow}$, since all $|E|$ runs have the same size. Thus, given E and a run size of g generations, we call the set of all $|E|$ generations with identical index i the generation i of the virtual run. Accordingly, we call the set of the $|E|$ populations of E the virtual population of the experiment.

In order to compare the performance of the empirical search algorithms $c_{\rightarrow}$ and $d_{\rightarrow}$ (cf. 3.6, p.56), three experiments must be conducted, and each one shall have

- experiment size: *19* runs.

We index the runs of an experiment with $i \in [0, 18] \cap \mathbb{N}_0$. Each run of the same experiment gets a different seed value, i.e., a value that initializes the underlying pseudo-random number generator. All runs that have the same index—i.e., they come from different experiments—get the same seed value. This supports comparability of experiments.

- The common experiment consists of runs of $c_{\rightarrow}$.

  The coupled experiment consists of runs of $d_{\rightarrow}$ using the coupled mutator.

  The unrestricted experiment consists of runs of $d_{\rightarrow}$ using the unrestricted mutator.

Apart from these differences, the experiments use identical algorithmic components and parameters (cf. 4.9, p.101).

5.3.14 Empirical results and discussion

Since the projected search algorithm $p_{\rightarrow}$ is generational (cf. 4.10, p.104), we shall measure run time as the number of completed generations.

Let r denote a $p_{\rightarrow}$ run using fitness function fit. We call

$$\frac{\sum_{i \in pop_{\mathsf{r},\mathsf{t}}} \mathsf{fit}(i)}{|pop_{\mathsf{r},\mathsf{t}}|}$$

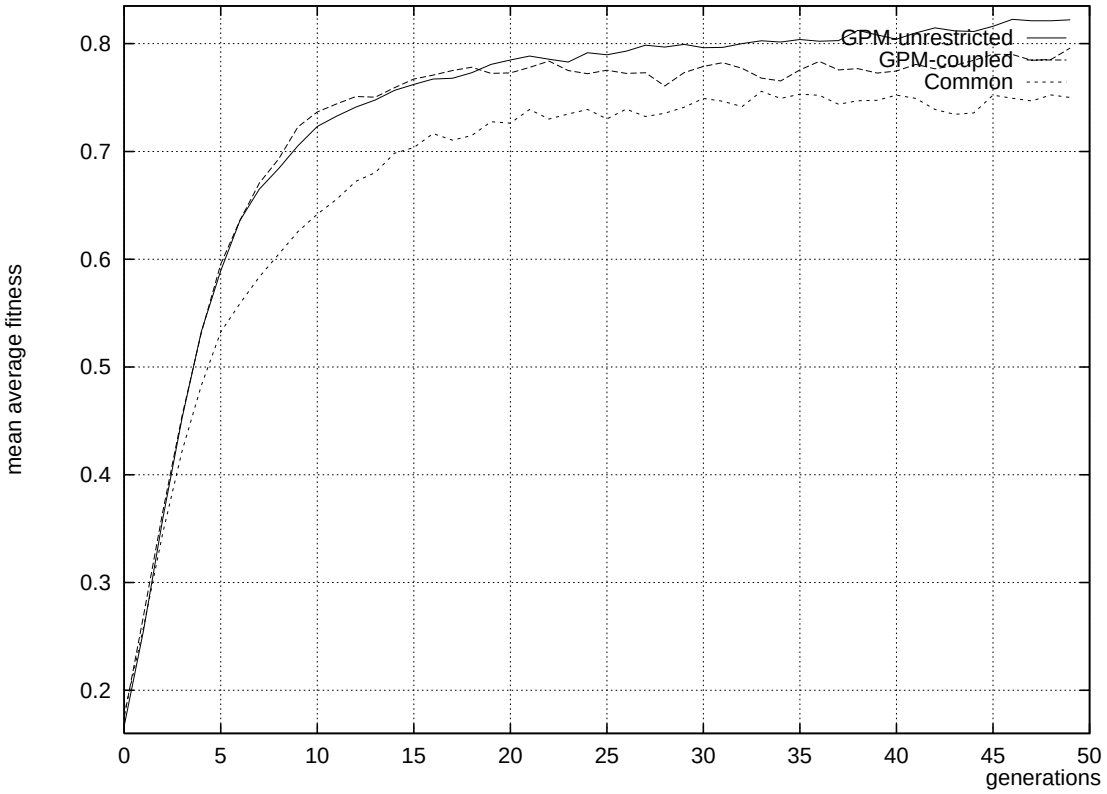

Figure 5.2: Progression of the mean average quality.

the average quality of generation t of r, denoted by $\overline{\mathsf{fit}}_{\mathsf{r}}(\mathsf{t})$.

Let E denote an experiment.

$$\frac{\sum_{r \in \mathsf{E}} \overline{\mathsf{fit}}_r(\mathsf{t})}{|\mathsf{E}|}$$

shall be called the mean average quality of generation t of E, or, synonymously, mean average fitness, denoted by $\overline{\overline{\mathsf{fit}}}_{\mathsf{E}}(\mathsf{t})$.

Figure 5.2 shows graphs that visualize the progression of the mean average fitness for the common, coupled, and unrestricted experiment, respectively. These progressions reflect the performance of the respective empirical search algorithms. The acronym GPM in a graph label expands to "genotype-phenotype mapping" and signals the developmental type of the corresponding search algorithm.

Let $q(t)$ denote a quality measure for a run r or an experiment E, i.e., $\overline{\mathsf{fit}}_r(t)$ or $\overline{\overline{\mathsf{fit}}}_E(t)$. For generations $t, u : t < u$, we call

$$v = \frac{q(u) - q(t)}{u - t}$$

the progress of r or E over time span $[t, u]$. Thus, $v \in \mathbb{R}$, and a greater v is more desirable.

An entity associated with the common, coupled, or unrestricted experiment shall be qualified accordingly, e.g.: "a coupled graph." Both the coupled and the common graph start at the initial `generation 0` at about identical quality values. This indicates the sound situation that there is no artificial bias in the initial experimental conditions. Both graphs show strong progress until `generation 5`, with the coupled progress being greater. This early strong progress is typical for an Evolutionary Algorithm: initial genotypes, being random structures, usually have a low quality. Thus, finding better solutions is more likely early during a run.

At `generation 5`, however, the common progress drops while the coupled progress continues only somewhat weaker. The decrease in progress at the end of an early run stage is EA-typical: subpopulations start getting trapped in local optima. Here, one may explain the observed stronger coupled progress as follows.

Tunnel hypothesis

For P_1, the developmental search space $\mathrm{sea}_{\mathrm{d}_\rightarrow}$ is high-dimensional compared to the common search and solution space $\mathrm{sea}_{\mathrm{c}_\rightarrow}$ (cf. 5.48, p.135). Thus, a beneficially redundant genotype-phenotype mapping $\mathsf{gp}_{\mathrm{d}_\rightarrow}$ may exist (cf. text units 5.40, p.131, 5.3.12, p.135). It depends on $\mathsf{gp}_{\mathrm{d}_\rightarrow}$ whether an increase in $\mathsf{dim}(\mathrm{sea}_{\mathrm{d}_\rightarrow})$ is beneficial or detrimental: the redundancy of the genetic code used by $\mathsf{gp}_{\mathrm{d}_\rightarrow}$ is critical (cf. 5.36, p.130).

As we have given a code with desirable properties for P_1 (cf. 5.13, p.115), the resulting $\mathsf{gp}_{\mathrm{d}_\rightarrow}$ is expected to be beneficially redundant. Thus, we state the tunnel hypothesis: discussed, nice high-dimensional phenomena—especially, the tunnel principle—take effect and result in the observed stronger coupled progress (cf. 5.33, p.128).

Conflict of hypothesis recursion

We have derived the tunnel hypothesis from theoretical discussions. In principle, the ratio of the number of tunnels vs. $|\,\mathrm{sea}_{\mathrm{d}_\rightarrow}\,|$ can be computed using enumeration. However, the large size of $\mathrm{sea}_{\mathrm{d}_\rightarrow}$ (cf. 5.47, p.134) forbids this approach. This situation is an instance of a generic conflict of empirical investigation, as follows. If observed system behavior cannot be explained by current theory or complete observation, one produces a hypothesis on its explanation. One must then test the hypothesis against an appropriate additional observation to be made. In case the test supports the hypothesis, one must yield a next hypothesis on explaining the additional observation, and so on.[33] We call this situation the conflict of hypothesis recursion. One must terminate it by a plausibility argument for the last produced hypothesis.[34] (5.53)

Here, we are satisfied to take the theoretical discussions that lead to the tunnel hypothesis as such an argument.

[33] It is a philosophical issue whether this recursion perhaps continues *ad infinitum*.

[34] The recursion depth depends on the quality of the plausibility argument and available experimental resources.

Coupled and common experiment

At `generation 9`, the coupled progress begins falling stronger, and over the final ten generations, it remains slightly positive on average. The common progress continues falling, and over the final ten generations, it is virtually zero.

The common quality value is always less than the coupled one in the same `generation` $i > 2$, due to the described different coupled and common progress profile. Especially, at the

- `final generation`, the coupled maximal mean average quality is about 6 percent higher relative to
- the common maximal quality, reached at `generation 33`.

Summarizing, the coupled experiment performs better than the common experiment.

The difference between coupled and common progress is best visible between `generations 5` and `20`. Most probably, during this time and later, low-quality local optima (cf. 3.10, p.59) reduce the common progress strongly, while coupled progress may be more stable due to tunnel effects that support escapes of trapped subpopulations.

During the following period, common quality rises to its maximum at `generation 33` and then, on average, slightly falls until the final generation, i.e., `generation 49`.

- Thus, common progress is negative for the time span `[33,49]`.

This indicates that $c_{\rightharpoonup}$'s potential for quality improvement is exhausted. During the same time, coupled quality keeps rising on average, although its progress is not as great as during `generation 0` to `20`.

- Thus, coupled progress is positive for the time span `[33,49]`.

This shows potential for further quality improvement by $\mathbf{d_{c\rightharpoonup}}$, i.e, the coupled instance of $d_{\rightharpoonup}$.

Escapes are performance-critical during a later stage of an EA run r with a fixed-population size: an emergent problem often yields a landscape with numerous local optima. Thus, exploration may turn parts of $\boldsymbol{pop_r}$, i.e., r's population, into trapped subpopulations.

Local optima exist in both a high- and a low-dimensional search space. Thus, both coupled and common progress suffer from trapped subpopulations. An effect of this situation shows in the coupled and the common progression: beginning in `[15,20]` and continuing to the final generation, both progressions oscillate.

Such quality oscillation is typical for a non-elitist selection method and a multimodal problem, as follows. Let $\mathfrak{S} \subset pop_r$ sit at a local optimum q. $\mathfrak{S}$ may migrate from q to a worse or better point, decreasing or increasing $\mathfrak{S}$'s mean average quality, respectively. On the one hand, the search dynamics fosters phenomena that increase

quality. On the other hand, it may lead to situations — e.g., trapped subpopulations — that obstruct a quality increase. In particular, the older an experiment is in terms of run-time, the more and better local optima it has found, so that improving quality becomes less probable. Therefore, a dynamic equilibrium between beneficial and detrimental pressure emerges that yields quality oscillation. (5.54)

This explains the shared issue of decreasing average common and coupled progress.

As for the discussed difference in the progress profiles: a small trapped coupled subpopulation $\mathfrak{S}$ may walk a large neutral network N of the trapping local optimum n. However, due to the high dimensionality of $\text{sea}_{\text{d}\rightarrow}$,[35] N might contain genotypes that are entry points of tunnels to networks of optima that are better than n. Especially, several such genotypes might be entry points to the same network O of a better local optimum. Such situations increase the probability that $\mathfrak{S}$ still locates a better genotype. This might explain why

- the coupled progress is positive for every time span `[t,49]`, $0 \leq \texttt{t} < 49$, as opposed to the common one.

The only difference between $\text{c}_{\rightarrow}$ and $\text{d}_{\text{c}\rightarrow}$ is that the latter is developmental. Thus, this difference causes the observed difference in performance. Therefore, adding algorithmic metaphors of development to a Genetic-Programming algorithm may enhance its performance. This partially realizes the technical objective of the present work (cf. 1.9, p.25).

- In summary, the coupled performance is greater than the common one. This is explained by the developmental type of the coupled experiment. In particular, we hypothesize that tunnel effects counter premature convergence. Thus, the technical objective has been approached. (5.55)

Unrestricted, coupled, and common experiment

The coupled and unrestricted quality progressions are almost equal in period `[0,18]`. Especially, the initial unrestricted quality is virtually that of the two other experiments.

In `[18,49]`, the unrestricted progress is greater than the coupled one:

- At `generation 46`, the unrestricted experiment reaches its maximal mean average quality.

- This quality is about 10 percent higher relative to the common maximal quality, and

[35] $\text{d}_{\text{c}\rightarrow}$ has the same search space $\text{sea}_{\text{d}\rightarrow}$ as $\mathbf{d_{u\rightarrow}}$, the unrestricted $\text{d}_{\rightarrow}$ instance. This holds since: i) $\text{d}_{\text{c}\rightarrow}$ and $\text{d}_{\text{u}\rightarrow}$ use an identical genotypic representation; ii) their mutators manipulate atomic components of the representation, so that both algorithms face the same dimensionality.

- it is about 4 percent higher relative to the coupled maximal quality which is reached at the final generation. (5.56)

- Summarizing, the unrestricted experiment performs better than the coupled and the common one.

The only difference between $\mathrm{c}_{\rightarrow}$ and $\mathrm{d}_{\mathrm{u}\rightarrow}$ is that the latter is developmental. Thus, this difference causes the observed difference in performance. Therefore, adding algorithmic metaphors of ontogeny to a Genetic-Programming algorithm may enhance its performance. This partially realizes the technical objective of the present work (cf. 1.9, p.25).

Both $\mathrm{d}_{\mathrm{u}\rightarrow}$ and $\mathrm{d}_{\mathrm{c}\rightarrow}$ are developmental. Thus, the explanation for the higher coupled performance relative to the common one (cf. 5.55, p.141) also applies to the higher unrestricted performance relative to the common one: the only difference between $\mathrm{d}_{\mathrm{u}\rightarrow}$ and $\mathrm{d}_{\mathrm{c}\rightarrow}$ is the respective mutator, and the given explanation does not consider this difference. Next, we discuss this difference for comparing the unrestricted and the coupled experiment.

Let $\mathbb{A}_{\mathbf{P_1}}$ denote the source alphabet of P_1. Consider the codons

$$c, d \in \mathbb{A}_{\mathrm{P}_1} : \mathrm{hd}(c, d) = 1 \vee \mathrm{hd}(c, d) = 3.$$

Under the coupled mutator $\mathrm{\Rsh}\mathbf{mut}_{\mathbf{cpl}}$, there is no route (s. def.) from c to d because

$$\mathrm{hd}(c, \mathrm{\Rsh mut}_{\mathrm{cpl}}(c)) = 2.$$

For instance, under $\mathrm{\Rsh mut}_{\mathrm{cpl}}$, there is no route from 0011 to 0001. Thus, for a genotype $g = (\mathrm{c}_i), \mathrm{c}_i \in \mathbb{A}_{\mathrm{P}_1}$, repeated $\mathrm{\Rsh mut}_{\mathrm{cpl}}$ application can reach exactly all genotypes whose Hamming distance to g is even. Therefore, a search route starting at g can only explore a proper subspace of $\mathrm{sea}_{\mathrm{d}\rightarrow}$. However, repeated application of the unrestricted mutator $\mathrm{\Rsh}\mathbf{mut}_{\mathbf{unr}}$ reaches all genotypes. Therefore, a search route starting at g can explore $\mathrm{sea}_{\mathrm{d}\rightarrow}$.

- Summarizing, $\mathrm{d}_{\mathrm{u}\rightarrow}$, in contrast to $\mathrm{d}_{\mathrm{c}\rightarrow}$, can access the complete neighborhood of a genotype. (5.57)

As mentioned, the unrestricted quality progression almost equals the coupled progression up to `generation 18`. From there to the final generation, the unrestricted progress is greater than the coupled one.

For a multi-modal problem and non-elitist selection, quality oscillation is likely to occur during a later stage of an experiment (cf. 5.54, p.141). However, the quality oscillation of the unrestricted experiment is dampened compared to the coupled one. We suggest this dampening occurs because an unrestricted trapped subpopulation has more escape routes available due to the larger accessible neighborhood (cf. 5.57, p.142). Thus, it easier finds a better genotype even during a later stage, so that quality is less likely to drop.

- In summary, for $d_{u\rightarrow}$ vs. $d_{c\rightarrow}$, the unrestricted mutator dampens quality oscillation, supporting positive progress.

The described higher performance of $d_{u\rightarrow}$ compared to the common search algorithm $c_{\rightarrow}$ contradicts experience on Evolutionary Computation. This is because the search space size and dimensionality of $d_{u\rightarrow}$ are 2 and 1 orders (cf. 5.48, p.135) of magnitude higher, respectively, than those of $c_{\rightarrow}$. As the population size is identical for both algorithms, a higher performance of the common algorithm should be observed due to the curse of dimensionality.

Thus, the actual observation corroborates the conclusion that a redundant genotype-phenotype mapping may be sufficient for beneficial high-dimensional phenomena (cf. 5.33, p.128): it confirms the redundancy-based predictions that

- better performance was to be expected for instances of $d_{\rightarrow}$ due to the given genetic code (cf. 5.49, p.135), and

- that a $d_{\rightarrow}$ instance should perform only slightly better than $c_{\rightarrow}$ (cf. 5.56, p.142)(cf. 5.50, p.135).

In particular, the phenomena demonstrated by use of small synthetic examples, e.g., the four-dimensional sign example, might be effective for the — large — first empirical problem. This supports the conjecture that these phenomena are independent from the size of a search space (cf. 5.41, p.131).

- In summary, the unrestricted developmental experiment has a higher performance than the common one, which partially realizes the technical objective of the present work. A redundant genotype-phenotype mapping may be sufficient for beneficial high-dimensional phenomena which might scale up in the context of large emergent problems. The unrestricted experiment also has a higher performance than the coupled one. This argues for using point mutation during a developmental search algorithm.

Next, we draw conclusions for further approaching the technical objective.

5.3.15 Conclusion and further research

We have compared the empirical common search algorithm and two instances of the empirical developmental search algorithm on a real-world problem. The results argue for the developmental flavor that we shall further investigate.

An employed genetic code is critical for search progress. An unfortunate manual code design simultaneously loses beneficial and introduces detrimental potential regarding the resulting genotype-phenotype mapping (cf. 5.51, p.135). We have discussed different aspects of this situation (cf. text units 5.46, p.134, 5.45, p.133, 5.44, p.133, 5.42, p.131, 5.12, p.114).

- Summarizing, the concept of a genotype-phenotype mapping makes sense if and only if it employs an adapted genetic code.

- In conclusion, for problems that do not allow for an *a priori* code adaptation, the evolution of a genetic code is necessary. This shall be the next issue of investigation.

All arguments for evolving a structure originate from the subject of incomplete problem knowledge. For evolving codes, the unknown local landscape topology of an individual matters. The hypothesized phenomenon of local smoothing argues for evolving individual codes that then co-determine individual genotype-phenotype mappings (cf. 5.46, p.134).

The observations corroborate experience from Evolution Strategies and Evolutionary Programming that mutation as sole source of variation can imply progress. In particular, mutation appears to be beneficial for both the common and developmental search algorithm.[36]

- In conclusion, we focus on evolving individual genetic codes with mutation as only variation source. Thus, the feasibility of code evolution must be demonstrated. To that end, next, we shall determine a second empirical problem.

(5.58)

[36]We emphasize this issue because recombination has been the primary variation operator of initial contributions to Genetic Programming. This situation has strongly influenced the field.

Chapter 6

Second empirical problem

It is of interest whether an instance of the empirical developmental search algorithm $\mathrm{d}_{\rightarrow}$ can *co-evolve* genetic codes and genotypes (cf. 5.58, p.144)(cf. 1.2.4, p.17). To investigate, we will first propose a principle for evolving genetic codes. To the end of deciding the principle's feasibility, we will design an appropriate $\mathrm{d}_{\rightarrow}$ instance that shall be called the second developmental search algorithm, denoted by $\mathbf{d}_{\mathbf{x}\rightarrow}$. The *first* developmental search algorithm (cf. 4.12, p.105) has resulted from a general discussion. Thus, $\mathrm{d}_{\mathrm{x}\rightarrow}$ shall build on this first instance.

6.1 Genotype evolution

6.1.1 Repairing

An individual $\dot{\boldsymbol{g}}$ with a genotype g and phenotype p_g shall be denoted by $(\boldsymbol{g}, \boldsymbol{p_g})$. Given an illegal primary transcript $tra(g)$, repairing yields p_g (cf. 4.2.12, p.82). A certain instance of replacing repairing (cf. 4.5.2, p.98) is a component of the first developmental search algorithm. A simpler algorithm, namely, deleting repairing (cf. 4.2.12, p.82), has been discussed.

- Due to the minimalism principle, we decide on $\mathrm{d}_{\mathrm{x}\rightarrow}$ to employ deleting repairing, denoted by $\not{\boldsymbol{p}}_{\rightarrow}$.

Therefore, a $\not{p}_{\rightarrow}$ instance must be determined next by identifying one of several finalizing methods (cf. 4.5.2, p.96). First, $\not{p}_{\rightarrow}$ may produce a transformation sequence that ends with an irreducible transcript, so that, second, $\not{p}_{\rightarrow}$ must apply a finalizing algorithm.

- Due to the minimalism principle, $\not{p}_{\rightarrow}$ shall employ *deleting* finalizing (cf. 4.5.2, p.97).

For the infix example, a transformation sequence

$$**\mathtt{a}/ \;\rightarrow\; *\mathtt{a}/ \;\rightarrow\; \mathtt{a}/ \;\rightarrow\; \mathtt{a}$$

results from a $\not{p}_{\rightarrow}$ instance using deleting finalizing.

Deleting finalizing may produce the empty transcript ϵ (cf. 4.5.2, p.97).[1] For instance, for the infix example,

$$++++ \quad \rightarrow^* \quad \epsilon$$

results. For the determined empirical target language (cf. 5.1, p.107), ϵ is irreducible. Thus, if a genotype g_ϵ gets transformed into ϵ, p_{g_ϵ} does not exist, so that quality evaluation of g_ϵ cannot take place. However, a quality value of g_ϵ is essential to $\mathrm{d_{x\rightarrow}}$'s continuation.

- To that end, $\mathrm{d_{x\rightarrow}}$ should consider g_ϵ to have worst quality.

This requirement is plausible from a biological point of view: ϵ corresponds to the lack of an organism. Thus, the natural genotype in question will not reproduce. As Evolutionary Computation crudely models reproductive success by the concept of quality, worst quality is appropriate for g_ϵ. We achieve this as follows. Let $\mathbf{P_2}$ denote the second empirical problem. From a technical point of view, g_ϵ has no semantics because it lacks p_{g_ϵ}. However, as $\mathrm{d_{x\rightarrow}}$ is full (cf. 3.20, p.62), its genotype-phenotype mapping

$$\mathsf{gp}_{\mathrm{d_{x\rightarrow}}} : rep_{\mathrm{d_{x\rightarrow}}} \rightarrow sol_{\mathrm{P_2}} \subset L_{emp}$$

must be defined for $g_\epsilon \in rep_{\mathrm{d_{x\rightarrow}}}$.

- Thus, we redefine L_{emp} to include ϵ, so that we can define

$$\epsilon \in sol_{\mathrm{P_2}}.$$

  Accordingly, we extend the fitness function of $\mathrm{d_{x\rightarrow}}$ to yield the worst quality value for ϵ. We call the described handling of the empty-string situation the L_{emp} extension.

Eventually, we discuss alternative, less favorable approaches. Being a fixed-population-size Evolutionary Algorithm, $\mathrm{d_{x\rightarrow}}$ may not discard a produced individual $\dot{g} = (g, \epsilon)$. Therefore, $\mathrm{d_{x\rightarrow}}$ could replace $\dot{g}$ by

$$\dot{h} = (h, p_h), p_h \neq \epsilon.$$

However, L_{emp} extension saves resources that such replacement and subsequent quality evaluation of $\dot{h}$ would require. As another option, randomly creating h would introduce an undesirable external search bias, as would creating h as a copy of a real genotype. Finally, these alternatives are also more explicit than L_{emp} extension which produces a worst-quality genotype that is likely to be replaced by another genotype during subsequent selection and reproduction, anyway.

- Deleting-finalizing with L_{emp} extension shall be called open finalizing.[2] In conclusion, $\mathrm{d_{x\rightarrow}}$ shall apply deleting repairing with open finalizing to a produced genotype.

[1] While this situation is rare for a long genotype, technical soundness requires a discussion.

[2] The name comes from the notion that an irreducible transcript of a genotype g leaves the meaning of g open.

6.1.2 Variation

- We decide to implement variation as the point mutator $\vec{\mathbf{mut}}_{\dot{\mathbb{B}}}$ that randomly selects and inverts a bit of its argument (cf. 5.15, p.116)(cf. 4.9.1, p.101).

- In particular, due to the minimalism principle, $\vec{\mathrm{mut}}_{\dot{\mathbb{B}}}$ shall be the only variation operator of $\mathrm{d}_{\mathrm{x}\rightarrow}$ which is admissible since $\vec{\mathrm{mut}}_{\dot{\mathbb{B}}}$ is ergodic. (6.1)

An execution probability $\mathbf{p_x}$ of a copying or variation operator $\vec{\mathrm{o}}$ designates that an algorithm calls $\vec{\mathrm{o}}$ from the respective operator set with probability $\mathrm{p_x}$. This probability is also called a rate, and a rate is given to each $\vec{\mathrm{o}}$ in order to set a $\mathrm{d}_{\mathrm{x}\rightarrow}$ instance.

- We have determined $\mathrm{d}_{\mathrm{x}\rightarrow}$ regarding evolving genotypes. Next, evolving genetic codes is the focus. To that end, first, we must develop an appropriate principle in the context of Developmental Genetic Programming. Second, the principle's implementation needs discussion.

6.2 Genetic-code evolution

6.2.1 Biological motivation

The empirical genotype-phenotype mapping is a mathematical metaphor of biological phenomena: ontogeny, in particular, polypeptide synthesis. The universal genetic code (s. def.) is essential to this synthesis, so that its properties and origin are of interest to Developmental Genetic Programming (DGP). Like natural evolution has produced genotypes, it has yielded the universal genetic code. One has argued that natural selection favors those code properties necessary for the evolution of organisms (Maeshiro 1997).

- We conclude that natural evolution has produced the universal genetic code which potentially favors the emergence of adapted genotypes.

For a given problem, artificial evolution, gleaned from natural evolution, often results in acceptable genotypes. Thus, we suggest as hypothesis:

- Artificial evolution, appropriately implemented, produces beneficially redundant or even mirroring (cf. 5.11, p.114) genetic codes. That is, such evolution gives genetic codes that favor evolving acceptable genotypes. (6.2)

6.2.2 Technical motivation

DGP defines genotype semantics by a genetic code, a repairing method, and the employed target language. Investigating the evolution of repairing methods remains as future work (cf. 5.34, p.129), and the semantics of the target language is fixed. Thus, evolving genetic codes results as current focus regarding adapting genotype semantics for a given problem.

Extensive technical arguments for code evolution have been given (cf. 5.58, p.144). Essentially, $d_{x\rightarrow}$ should evolve codes such that its individuals move on beneficial landscapes. In particular, regarding the desired surjectivity of a genotype-phenotype mapping (cf. text units 4.5, p.75, 4.37, p.85), an objective of code evolution follows:

- An emerging genetic code $\mathsf{c} : \mathbb{A} \rightarrow \Omega_{d_{x\rightarrow}}$ should be surjective on $\Sigma \subset \Omega_{d_{x\rightarrow}}$, Σ containing exactly all problem-relevant target symbols. Thus, a genotype represents a phenotype that only consists of problem-relevant symbols. (6.3)

- We shall develop a principle of code evolution for Developmental Genetic Programming next.

6.2.3 Individual genetic code

So far, a developmental run r transcribes all genotypes of its population by use of the same global genetic code. This scenario corresponds to the current situation in organic evolution: the universal genetic code (s. def.) is essential to polypeptide synthesis in most organisms.[3] Organic evolution of different codes has resulted in the dominance of the universal genetic code. Thus, evolution has produced codes as well as it is producing genotypes.

- As we desire the emergence of artificial genetic codes, all necessary conditions for the evolution of structures must be given: existence of a population, heredity, variation, a quality measure, and quality-based selection.

- We define the code population of a run by replacing the concept of a global genetic code by that of an individual genetic code: each real individual $\dot{g} = (g, p_g)$ carries exactly one code c, and transcription of g uses c.

$\dot{g}$ shall be called a carrier of c. $\dot{g}$ is thus characterized by g and c, so that we denote it by $(\boldsymbol{g}, \mathsf{c})$.

- Summarizing, the code population of a run r is given as the set of the individual genetic codes of r's real individuals. The code-population size results as r's population size.

- We define $d_{x\rightarrow}$'s creation operator to produce an initial code c for an individual, giving an initial code population. By default, c shall be a random code. Table 6.1, p.149, represents an example of a random code. (6.4)

[3] One of the rare exceptions is mitochondrial polypeptide synthesis.

Table 6.1: Example of a random code.

000	001	010	011	100	101	110	111
*	/	*	a	a	d	+	a

6.2.4 Code variation, heredity, quality, and selection

- For reasons given for genotypic variation, we define point mutation to be the only type of code variation used by $\mathrm{d}_{\mathrm{x}\rightarrow}$.

The corresponding operator $\vec{\Gamma}\mathbf{mut}_{\dot{\gamma}}$ performs point code-mutation on a code

$$\mathsf{c} : \mathbb{A} \rightarrow \Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}},\ |\Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}}| > 1$$

as follows:

---

1. Random-select a codon $\mathrm{c_i} \in \mathbb{A}$.
2. Replace $a = \mathsf{c}(\mathrm{c_i}) \in \Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}}$ in code entry $(\mathrm{c_i}, a)$ of c's definition by a random-selected $b \in \Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}}$, $a \neq b$.

For instance, consider code c (cf. 6.1, p.149). $\vec{\Gamma}\mathrm{mut}_{\dot{\gamma}}$ randomly selects 010. Then, in c's 010–entry, it replaces $\mathsf{c}(010) =$ '$*$' by 'b'. The resulting mutant code c' maps 010 onto 'b'.

- Due to the individual-code concept, reproducing a carrier by copying implicitly reproduces the carried genetic code, which implements heredity.
- This situation honors the minimalism principle. It furthermore enhances the implicitness of $\mathrm{d}_{\mathrm{x}\rightarrow}$ because it renders an explicit code reproduction obsolete. This contributes to the strategic objective (cf. 1.5, p.14)(cf. 1.9, p.25).
- Analogously to reproduction, selection of a code c is implied by selection of c's carrier. Thus, to the end of giving rise to code evolution, there is no need for an explicit concept of code quality. However, for investigating code evolution, next, we must give such a concept in order to measure adaptation.

The quality of an individual shall also be called individual quality in order to avoid confusion with code quality. A code c, carried by $\dot{g} = (g, \mathsf{c})$, co-determines $\dot{g}$'s individual quality because it co-determines p_g. Thus, one may identify c's quality with $\dot{g}$'s quality: carrier quality defines the quality of the carried code. However, c, if carried by $\dot{h} = (h, \mathsf{c})$, $h \neq g$, may result in $p_h \neq p_g$. Thus, the individual qualities of g and h may differ, so that c's quality varies.

This variability is undesirable since the current focus is on adapting codes to a given problem which is independent from an individual. Thus, next, we must suggest a concept of code quality that is independent from carrier quality.

For subsequent experiments on P_2, which is a small synthetic problem to be determined, one may use a code-quality measure based on
i) search-space enumeration and
ii) the knowledge of a perfect solution.

Thus, such a measure is impractical for a real-world problem which has a large search space and no *a priori* known acceptable solution. For P_2, however, an even perfect solution $\hat{\boldsymbol{s}} \in \mathrm{sol}_{\mathrm{P}_2}$ will be known, and $\hat{s}$ shall be the only such solution.

- Let $\mathbf{gp}^{\mathsf{c}}_{\mathrm{d_{x\rightarrow}}}$ denote a mapping $\mathsf{gp}_{\mathrm{d_{x\rightarrow}}}$ that employs code c. We define the code fitness value of a genetic code c as

$$\boldsymbol{\gamma}\mathbf{fit}(\mathsf{c}) = \frac{|\mathsf{gp}^{\mathsf{c}}_{\mathrm{d_{x\rightarrow}}}{}^{-1}(\hat{s})|}{|\,\mathrm{sea}_{\mathrm{d_{x\rightarrow}}}\,|}.$$

For instance, if $\mathrm{sea}_{\mathrm{d_{x\rightarrow}}}$ contains 2^{12} genotypes, and if $\mathsf{gp}^{\mathsf{c}}_{\mathrm{d_{x\rightarrow}}}$ maps 200 of these onto the perfect solution, then $\gamma\mathsf{fit}(\mathsf{c}) = 200 \cdot 2^{-12}$ results. $\gamma\mathsf{fit}$ ranges $[0, 1]$, and a higher value indicates a more beneficial code.

- In summary, we have defined point code-mutation, copying, and selection of an individual genetic code. The implicit nature of a code being carried contributes to the strategic objective. A measure for code quality has been determined.

Next, we hypothesize that the introduced concepts of carried codes and their mutation, together with the underlying dynamics of genotypic evolution, yield the evolution of genetic codes. This prediction shall be corroborated by a plausibility argument (cf. 5.53, p.139).

6.3 Hypothesis on genetic-code evolution

- The code hypothesis to be investigated (cf. 6.2, p.147) is that code evolution is operational: $\mathrm{d_{x\rightarrow}}$ adapts individual genetic codes such that they are increasingly beneficially redundant.

A coarse argument for this conjecture is that artificial evolution is known to work for a wide variety of structures of interest. A more detailed reasoning follows. Let there be individuals $\dot{g} = (g, \mathsf{c}_{\dot{g}})$ and $\dot{h} = (h, \mathsf{c}_{\dot{h}})$ in a given run. Let code $\mathsf{c}_{\dot{h}}$ be more beneficially redundant than $\mathsf{c}_{\dot{g}}$. Thus, with some likelihood, phenotype p_h has a higher fitness than p_g. Therefore, since selection on phenotypes implies selection on the carried codes, $\mathsf{c}_{\dot{h}}$ has a higher probability than $\mathsf{c}_{\dot{g}}$ of being propagated over time by
i) being copied and
ii) being subjected to point code-mutation $\vec{\mathsf{m}}\mathrm{ut}_{\dot{\gamma}}(\mathsf{c}_{\dot{h}}) = \mathsf{c}'_{\dot{h}}$.

When c'_h is even more beneficially redundant than c_h, the given argument which applied to c_h applies to its mutant c'_h, and so forth.

The given reasoning applies in an analog manner to genotypes. As both the genotype and code of an individual are complementary aspects of its phenotype, the resulting process receives positive feedback.

- The given explanation shall be called the code-evolution explanation: an autocatalytic process emerges in which increasingly better genotypes carry increasingly beneficially redundant codes. That is, co-operative co-evolution is initiated: a mutual hitch-hiking of codes and their carriers begins. (6.5)

Thus, during an experiment, we expect codes to become more beneficially redundant along with an increase in the average individual fitness. Therefore, such an observation would corroborate the code hypothesis and its explanation. In particular, for problems that allow for measuring the code quality, the average code fitness should rise over time along with the average individual fitness. (6.6)

6.4 Experiment

6.4.1 Objectives

A series of $\mathrm{d}_{\mathrm{x}\rightarrow}$ runs shall be performed on problem P_2 that is to be small so that code-fitness computation is practical (cf. 6.2.4, p.150).

- Rephrased, the code hypothesis states that, for a code population, evolution increases the number of code entries that contain a problem-relevant target symbol (cf. 5.37, p.130).
- Thus, for testing the hypothesis, target alphabet $\Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}}$ must also contain problem-*ir*relevant symbols.
- In order to test the code-evolution explanation, the mean best and mean average code fitness and the mean best and mean average individual fitness must be measured (cf. 6.6, p.151).

6.4.2 Empirical function

P_2, the required small problem, shall be a symbolic regression (cf. 5.2.1, p.107) of a known problem function on a four-dimensional parameter space. The function shall be

$$\mathsf{f}_{\mathrm{P}_2}(a, m, v, q) = a^2,$$

and we introduce the three parameters m, v, q in order to represent noise, i.e., undesirable phenomena. (6.7)

- Thus, the corresponding noise symbols $\mathsf{m}, \mathsf{v}, \mathsf{q} \in \Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}}$ are the problem-irrelevant symbols required above. (6.8)

- All parameter values shall be real-valued and range $[0, 1]$.

Summarizing, the second empirical function $\mathsf{f}_{\mathbf{P_2}} : \mathbb{R}^4 \rightarrow \mathbb{R}$ results. Next, we determine all experiment parameters of $\mathrm{d}_{\mathrm{x}\rightarrow}$, some depending on $\mathsf{f}_{\mathrm{P}_2}$.

6.5 Parameters

6.5.1 Problem instance and size parameters

- Like P_1, P_2 is to be represented as a set of fitness cases. Due to $\mathsf{f}_{\mathrm{P}_2} : \mathbb{R}^4 \rightarrow \mathbb{R}$, a fitness case consists of
  four real input values and
  one real output value.

- The training set shall consist of
  100 random-generated fitness cases.

- We choose a population size of
  50 individuals for all runs of $\boldsymbol{E}_{\mathbf{P_2}}$, the experiment in the making. This relatively small value is appropriate considering P_2's simple nature.

- E_{P_2} shall have a size of
  50 runs. Each run shall have a size of exactly
  50 generations. Thus, a run will not terminate if and when it finds a perfect genotype. This property is desirable because interesting observations may also occur afterwards.

6.5.2 Alphabets

Target alphabet

- $\Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}} = \{\mathsf{m}, \mathsf{v}, \mathsf{q}, \mathsf{a}, +, *, -, /\}$

shall be the target alphabet. Thus, the smallest perfect phenotype $\mathrm{d}_{\mathrm{x}\rightarrow}$ may find equals "$\mathsf{a} * \mathsf{a}$". It is relevant that a *small* perfect phenotype can be represented because this allows for the use of a small genotype size which exponentially determines the size of search space $\mathrm{sea}_{\mathrm{d}_{\mathrm{x}\rightarrow}}$. Eventually, $\mathrm{sea}_{\mathrm{d}_{\mathrm{x}\rightarrow}}$ being small is essential to E_{P_2} because the code-fitness measure is infeasible on a large space (cf. 6.2.4, p.150). We call this situation the code-fitness-measure constraint.

The role of $\mathsf{m}, \mathsf{v}, \mathsf{q}$ as noise symbols has been mentioned (cf. 6.8, p.152). They are operand symbols. For P_2, the only other symbol type is that of an operator. Thus, we have introduced

- $+, -, / \in \Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}}$ as operator noise symbols.

Next, we can determine the source alphabet $\mathbb{A}_{\mathrm{P}_2}$ that depends on $\Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}}$.

Source alphabet

- We decide on $\mathbb{A}_{\mathrm{P}_2} = \mathbb{B}^3$.

Thus, an individual genetic code maps $2^3 = 8$ three-bit codons onto $\Omega_{\mathrm{d_{x\rightarrow}}}$. As $|\Omega_{\mathrm{d_{x\rightarrow}}}| = 8$, the determined codon size is minimal regarding $\mathrm{d_{x\rightarrow}}$'s ability to produce a surjective genetic code (cf. 4.37, p.85). Keeping the codon size minimal is desirable due to the code-fitness-measure constraint.

6.5.3 Spaces

Code space

Due to $|\mathbb{A}_{\mathrm{P}_2}| = |\Omega_{\mathrm{d_{x\rightarrow}}}| = 8$, the set of all genetic codes contains $8^8 \approx 1\,\mathtt{E}\,7.2$ elements. We call such a set the code space of a Developmental Genetic Programming algorithm $a_{\rightarrow}$ that features code evolution, denoted by $\boldsymbol{\gamma}\mathbf{sea}_{\boldsymbol{a_{\rightarrow}}}$.

Search space

As symbolic genotype size for $g \in \mathrm{sea}_{\mathrm{d_{x\rightarrow}}}$, we determine

- $|g|_s = 4$.

The only perfect phenotype that $\mathrm{d_{x\rightarrow}}$ can produce from four-codon genotypes equals

$$\hat{s} = \mathsf{a} * \mathsf{a}. \tag{6.9}$$

This small symbolic genotype size and the singular nature of $\hat{s}$ are desirable due to the code-fitness-measure constraint. In particular, as the codon size equals three, a small search-space size results:

- $|\,\mathrm{sea}_{\mathrm{d_{x\rightarrow}}}\,| = 2^{4*3} \approx 1\,\mathtt{E}\,3.6$

6.5.4 Operator execution probabilities

We determine the execution probabilities of the operators for variation and copying, denoting the last one by $\overset{\curvearrowright}{\Gamma}\mathbf{cop}$.

Table 6.2: Execution probabilities for variation and copying for the second empirical problem.

$\mathrm{p_x}(\overset{\curvearrowright}{\Gamma}\mathrm{cop})$	0.6
$\mathrm{p_x}(\overset{\curvearrowright}{\Gamma}\mathrm{mut}_{\dot{\mathbb{B}}})$	0.32
$\mathrm{p_x}(\overset{\curvearrowright}{\Gamma}\mathrm{mut}_{\dot{\gamma}})$	0.08

$\mathrm{p_x}(\overset{\curvearrowright}{\Gamma}\mathrm{mut}_{\dot{\mathbb{B}}})$ is over 50 percent of the copying rate, and $\mathrm{p_x}(\overset{\curvearrowright}{\Gamma}\mathrm{mut}_{\dot{\gamma}})$ is only 25 percent of $\mathrm{p_x}(\overset{\curvearrowright}{\Gamma}\mathrm{mut}_{\dot{\mathbb{B}}})$. Thus, $\overset{\curvearrowright}{\Gamma}\mathrm{cop}$ may yield several clones of an individual $\dot{g} = (g, \mathsf{c})$ before mutation changes clones. $\overset{\curvearrowright}{\Gamma}\mathrm{mut}_{\dot{\mathbb{B}}}$ may change the identical genotypes of clones,

thus distributing the ex-clones over the fitness landscape. Eventually, $\curlyvee\mathrm{mut}_{\dot{\gamma}}$ may change the still identical codes of ex-clones, thus possibly adapting the codes to different landscape regions. Superimposed, due to $\curlyvee\mathrm{cop}$'s top rate, copying of mutants amplifies this process. Therefore,

$$\mathrm{p_x}(\curlyvee\mathrm{cop}) > \mathrm{p_x}(\curlyvee\mathrm{mut}_{\dot{\mathbb{B}}}) > \mathrm{p_x}(\curlyvee\mathrm{mut}_{\dot{\gamma}})$$

represents different time scales of the dynamics of $\mathrm{d_{x\rightarrow}}$.

- In extended summary, we have determined all standard parameters of the second developmental search algorithm $\mathrm{d_{x\rightarrow}}$. Next, its experiment-specific parameters must be fixed.

6.5.5 Initial individual genetic code

By default, $\mathrm{d_{x\rightarrow}}$ produces random initial codes (cf. 6.4, p.148). However, for experiment E_{P_2}, each initial codon entry shall encode

- '$-$'$\in \Omega_{\mathrm{d_{x\rightarrow}}}$. Thus, all initial codes are equal to the one given by table 6.3.

Table 6.3: Initial codes for experiment on second empirical problem.

000	001	010	011	100	101	110	111
-	-	-	-	-	-	-	-

Therefore, for a genotype $g \in \mathrm{sea}_{\mathrm{d_{x\rightarrow}}}$, the primary transcript

$$tra(g) = \text{“}----\text{”}$$

results. Under deleting repairing with open finalizing (cf. 6.1.1, p.145),

$$\begin{gathered}\mathsf{gp}^{\mathsf{c}}_{\mathrm{d_{x\rightarrow}}}(g) = \epsilon \neq \hat{s} = \mathsf{a} * \mathsf{a} \\ \Longrightarrow\ \mathsf{fit}(g) = 0 \wedge \gamma\mathsf{fit}(\mathsf{c}) = 0 \\ \text{for an initial code } \mathsf{c}.\end{gathered} \tag{6.10}$$

- In conclusion, no initial code or genotype has a selective advantage. Also, genotype evolution and the predicted code evolution start under worst conditions. Thus, final code quality and individual quality will be significant.

- For $\mathrm{d_{x\rightarrow}}$, the fixed initial individual genetic code supports the upcoming empirical analysis. It also represents a global code that code mutation then diversifies into individual codes.

- In extended summary, all parameters regarding E_{P_2} are set.

6.6 Results and discussion

6.6.1 Code-evolution explanation

Two argumentation lines corroborating the code-evolution explanation follow.

Weak corroboration

Top down, figure 6.1 shows the progressions of the mean best individual fitness, mean average individual fitness, mean best code fitness, and mean average code fitness on a logarithmic fitness scale.

At `generation 0`, for each population member, individual fitness and code fitness equal zero (cf. 6.10, p.154). Therefore, no graph starts here, since log(0) is not defined.

Both individual-fitness graphs rise, i.e., there is positive progress (s. def.).

Initial progress is decreased by the combination of i) tournament selection—which is not an elitist-selection variant—and ii) the high mutation rate $\mathrm{p_x}(\overset{\curvearrowright}{\mathrm{mut}}_{\dot{\mathbb{B}}})$: a $\mathrm{d_{x\rightarrow}}$ run r can lose a good individual $\dot{g}$ before $\overset{\curvearrowright}{\mathrm{cop}}$ amplifies $\dot{g}$'s frequency in pop_{r} sufficiently for protection against extinction. We observe such loss in fact in E_{P_2}, even for a perfect individual $\dot{g} = (g, \hat{s})$.

The overall positive individual-fitness progress is relevant because, according to the code-evolution explanation, it is necessary for code adaptation.

Due to the small mutation rate $\mathrm{p_x}(\overset{\curvearrowright}{\mathrm{mut}}_{\dot{\gamma}})$, both *code-fitness* graphs only become visible at `generation 2`.

- We observe positive progress for best and average code fitness. This situation, together with the positive individual-fitness progress, corroborates the code-evolution explanation (cf. 6.6, p.151).

Also, the synchronous changes in the progress of the average code fitness and average individual fitness support the explanation that predicts that both genotype evolution and code evolution show simultaneously as co-evolution. A realization of the prediction shows in figure 6.1 at `generation 9`: the progress of both the mean average code fitness and the mean average individual fitness decline strongly (*logarithmic* scale).

- Thus, this observation backs up the code-evolution explanation.

- In summary, the observed simultaneous and synchronous positive progress of individual fitness and code fitness corroborates the code-evolution explanation.

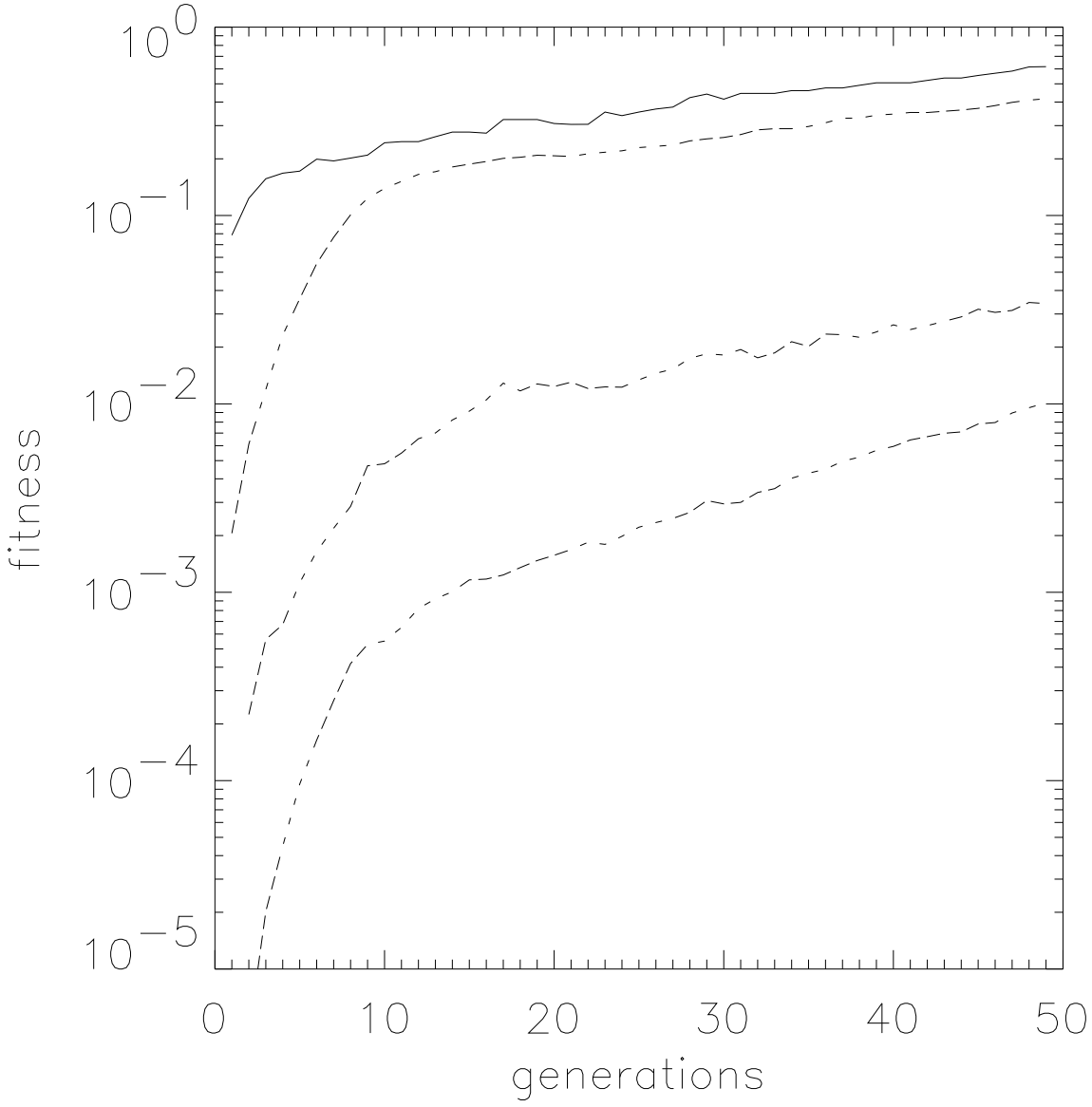

Figure 6.1: Top down, the graphs show the progressions of the mean best individual fitness, mean average individual fitness, mean best code fitness, and mean average code fitness on a logarithmic fitness scale.

Strong corroboration

Regarding individual fitness and code fitness, the code-evolution explanation (CEE) implies: if the one displays positive progress, the other one does so, too. We call this relationship the fitness-fitness correlation. It implies that

i) better, i.e., above-average, individuals carry better codes, and

ii) better codes are carried by better individuals.

It is in accordance with the observed synchronous progress of average individual (genotype) fitness and code fitness. However, progress of *averaged* fitness does not necessarily imply the fitness-fitness correlation: averaging loses the information about the genotype/code fitness-value pair of an individual. Therefore, figure 6.1 corroborates the CEE merely weakly, i.e., the former does not contradict the latter. As the CEE implies the fitness-fitness correlation, an analysis on the existence of this dependency in E_{P_2} is desirable:

- We raise the question whether indeed, over time,

  i) better individuals tend to have better codes, and

  ii) better codes are mostly being carried by better individuals. (6.11)

To approach this issue, we define the coupled fitness value of an individual $\dot{g} = (g, \mathsf{c})$:

$$\dot{\mathsf{fit}}(\dot{g}) := \mathsf{fit}(g) \cdot \gamma\mathsf{fit}(\mathsf{c})$$

Figure 6.2 illustrates the progression of the mean average coupled-fitness values. The positive progress reflects figure 6.1. However, a deviation is to be pointed out next. To that end, the mean average code quality of generation t of experiment E shall be denoted by $\overline{\overline{\boldsymbol{\gamma\mathsf{fit}}}}_{\mathsf{E}}(\mathbf{t})$. Analogously, we use $\overline{\overline{\dot{\mathsf{fit}}}}_{\mathsf{E}}(\mathbf{t})$. At the `final generation` $\boldsymbol{\omega} = 49$, we have

$$\mathrm{a} := \overline{\overline{\dot{\mathsf{fit}}}}_{E_{\mathrm{P}_2}}(\omega) = 6.4 \cdot 10^{-3}.$$

Average individual fitness and code fitness values are $4 \cdot 10^{-1}$ and 10^{-2}, respectively, resulting in

$$\mathrm{b} := \overline{\overline{\mathsf{fit}}}_{E_{\mathrm{P}_2}}(\omega)\overline{\overline{\gamma\mathsf{fit}}}_{E_{\mathrm{P}_2}}(\omega) = 4 \cdot 10^{-3},$$

$$\mathrm{q} := \mathrm{a}/\mathrm{b} = 1.6$$

Thus, for ω, the average coupled-fitness value is 1.6 times higher as expected for no dependency between individual and code fitness.

- In particular, $q > 1$ answers the above question (cf. 6.11, p.157) positively, i.e., it confirms the fitness-fitness correlation predicted by the CEE.

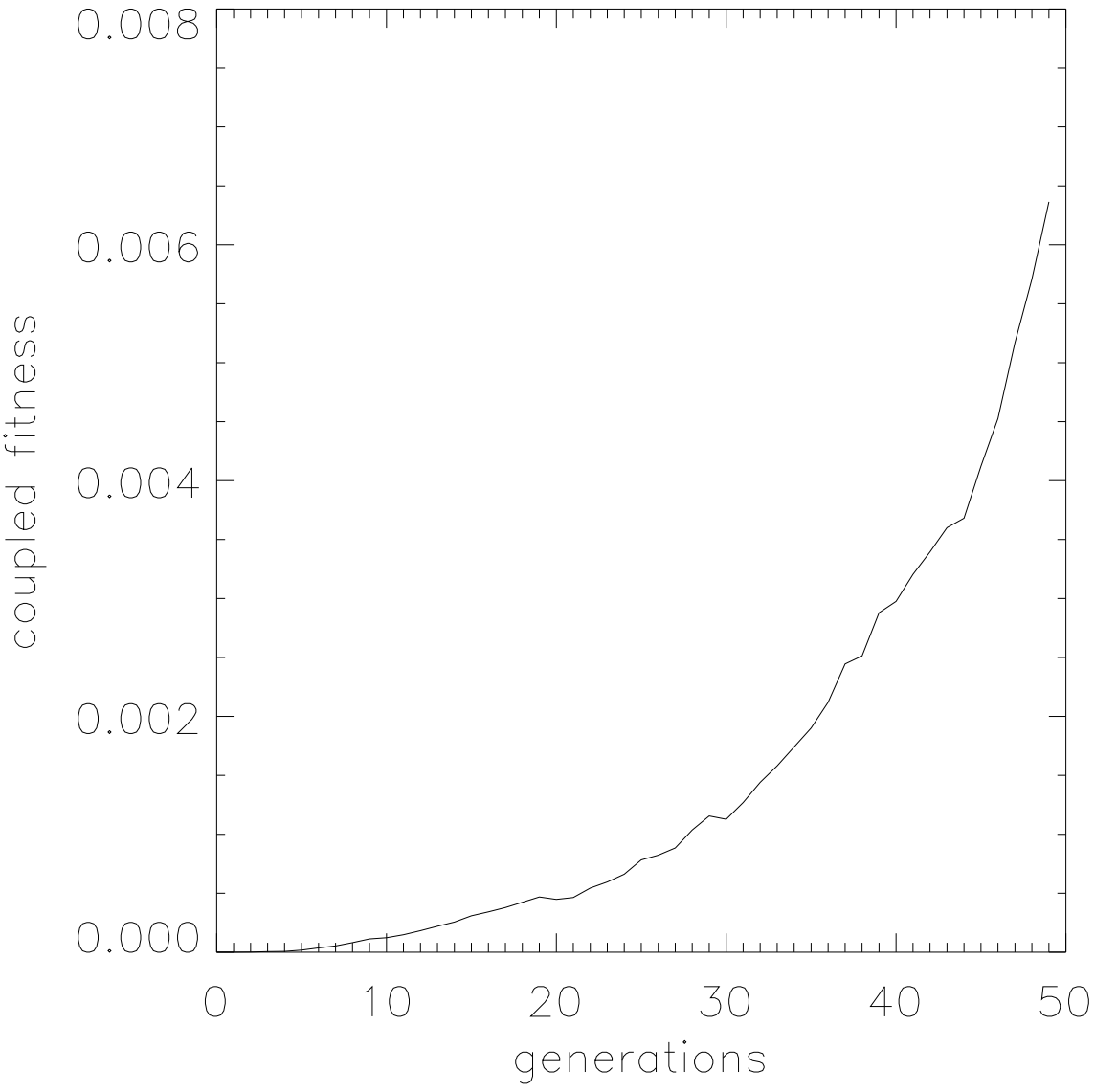

Figure 6.2: Progression of the mean average coupled-fitness values.

Individual and code fitness

Figure 6.3 and figure 6.4 visualize this correspondence. Each figure displays, for a given generation $\mathtt{t}$ of the experiment and for each of the following intervals, the mean-average numbers of $\mathtt{t}$'s individuals whose fitness and code fitness are in the respective intervals. We set a factor $r := 0.1$ to determine the fitness intervals $[0, r]$, $]r, 2r]$, .., $]9r, 10r]$, and, for $r := 0.02344$,[4] the code-fitness intervals are analogous.

Figure 6.3 reflects the situation for generation 0, with each of the 50 individuals having fitness and code fitness zero (cf. 6.10, p.154). Figure 6.4 shows the scenario of the final generation: put vividly, the population has started migrating toward a situation where a better individual may carry a better code and vice versa.

Table 6.4, p.159, summarizes the values from figure 6.4.

Table 6.4: Individual distribution. First row x/y gives co-ordinates within the fitness/code-fitness plane from figure 6.4, with $x = 0$ meaning fitness interval $[\,0, r\,]$ and so forth, y analogously for code fitness. Second row *Individuals* gives number of individuals represented by the respective population column in the figure. $\Sigma < 50$ indicates ca. $0.3/50 = 0.6\%$ of generation that is distributed over several more columns that appear with height 0.0 at chosen figure scale.

x/y	0/0	0/1	0/2	1/0	2/3	2/6	9/0	9/1	9/3	9/6	Σ
Individuals	5.8	0.2	28.8	0.7	0.6	0.1	11.8	1.1	0.2	0.4	**49.7**

- In conclusion, experiment E_{P_2} strongly corroborates the code-evolution explanation. Next, we analyze whether code evolution indeed occurs in E_{P_2}, i.e., we test the code hypothesis.

6.6.2 Code evolution

In order to test the hypothesis, the progression of code redundancy on each target symbol in $\Omega_{\mathrm{d_{x\rightarrow}}}$ is of interest. As a direct measure, we define the symbol frequency of a genetic code $\mathtt{c}$ on $s \in \Omega_{\mathrm{d_{x\rightarrow}}}$ as

$$\boldsymbol{\phi(\mathtt{c}, s)} = |\mathtt{c}^{-1}(s)|.$$

For instance, for E_{P_2}'s initial code $\mathtt{c_0}$ (cf. 6.3, p.154), $\phi(\mathtt{c_0}, \text{`}-\text{'}) = 8$ holds.
Thus, we can express the code redundancy $red_{\mathtt{c}}(s)$ (s. def.) of $\mathtt{c} : \mathbb{A} \rightarrow \Omega$ on $s \in \Omega$ as

$$red_{\mathtt{c}}(s) = \frac{\phi(\mathtt{c}, s)}{|\mathbb{A}|}.$$

In particular, $\phi(\mathtt{c}, s) = 1$ is equivalent to $\mathtt{c}$ being neutral on s. (6.12)

[4] $0.2344 = 10r$ is the maximal code fitness located over all runs of the experiment.

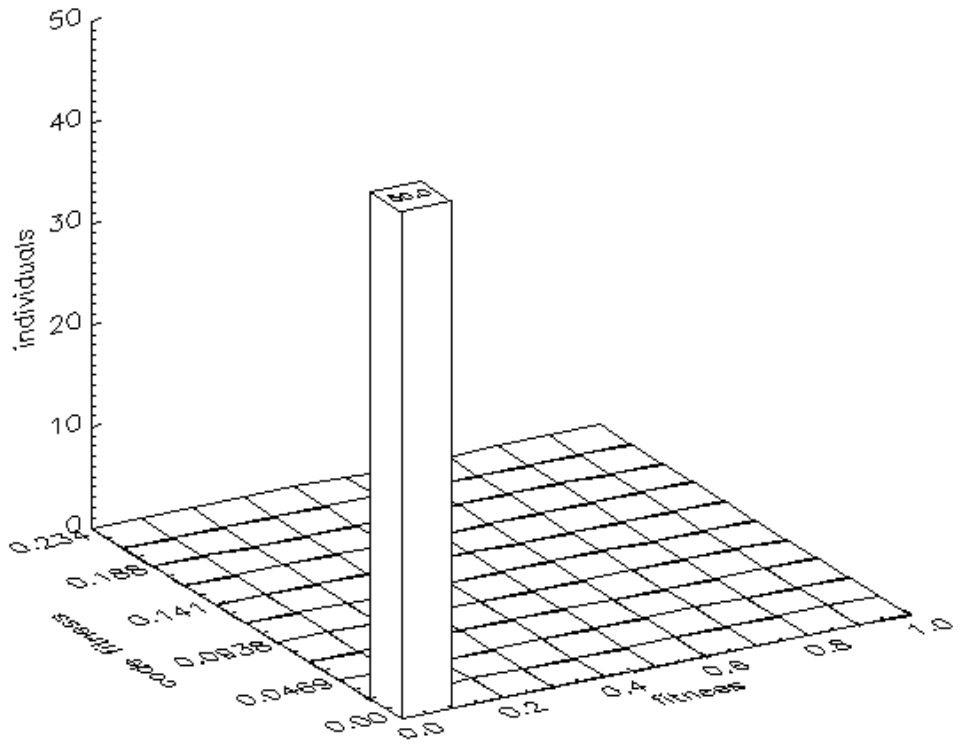

Figure 6.3: Generation 0. Population distribution in fitness/code-fitness space. Mean-average values. x-axis: individual fitness; y-axis: code fitness; z-axis: number of individuals in given generation that have a fitness and code fitness corresponding to the position of their "population column."

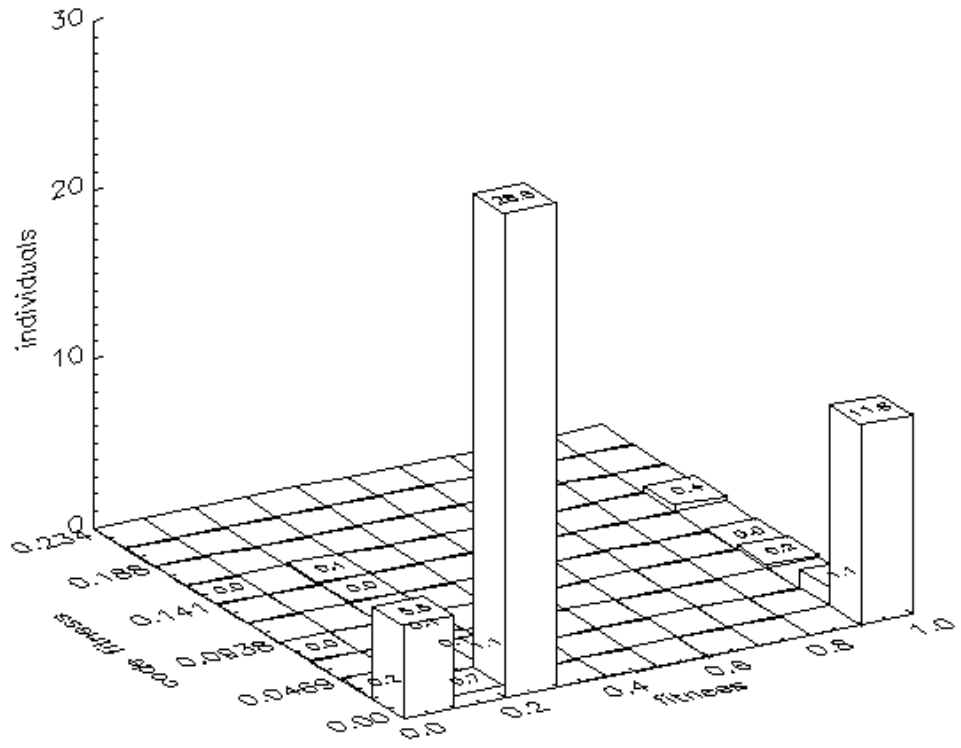

Figure 6.4: Generation 49. Ditto. Squares with top labeled "0.0" indicate column height less than 0.05. Tallest column represents strong non-global optimum 'a', second-tallest column visualizes global optimum '$\mathsf{a} * \mathsf{a}$'. Figure represents snapshot of population migration toward better code-fitness and fitness.

For given c, we call $\phi(\mathsf{c}, s)$ the s-frequency. For instance, one may refer to the a-frequency for 'a' $\in \Omega_{\mathrm{d_{x\rightarrow}}}$. Next, we define a measure over $\phi(\mathsf{c}, s)$. Let there be

- a fixed generation number i
- an experiment E on $\mathrm{d_{x\rightarrow}}$
- a fixed population size $\mathsf{p} = |pop_r|$ for each $\mathrm{d_{x\rightarrow}}$–run $r \in E$
- a fixed $\mathsf{s} \in \Omega_{\mathrm{d_{x\rightarrow}}}$.

Let $C_\mathsf{i} = \{\mathsf{c} \mid \dot{g} = (g, \mathsf{c}) \in pop_{r,\mathsf{i}}, r \in E\}$ be a multiset, i.e., repetitions *are* significant.[5] Then,

$$m_{\mathsf{i},\mathsf{s}} = \sum_{\mathsf{c} \in C_\mathsf{i}} \phi(\mathsf{c}, \mathsf{s})$$

is the sum of the s-frequencies over all $\mathsf{p}|E|$ codes in generation i of E.

$$\overline{\overline{\boldsymbol{\phi}(\mathbf{s}, \mathbf{i})}} = \frac{m_{\mathsf{i},\mathsf{s}}}{\mathsf{p}|E|}$$

shall denote the mean symbol frequency on s in generation i of E. Thus, s occurs, on average, $\overline{\overline{\phi(\mathsf{s}, \mathsf{i})}}$ times in each $\mathsf{c} \in C_\mathsf{i}$. The sum of $\overline{\overline{\phi(s, \mathsf{i})}}$ for $s \in \Omega_{\mathrm{d_{x\rightarrow}}}$ equals $|\Omega_{\mathrm{d_{x\rightarrow}}}|$.

(6.13)

Figure 6.5, p.163, visualizes the observed progression of the mean symbol frequencies on all target symbols. The logarithmic scale centers the neutral frequency value $\phi = 1$, emphasizing its meaning as lower boundary of the problem-relevance of a symbol. Initially, the maximum mean symbol frequency

$$\overline{\overline{\phi(\text{`}-\text{'}, 0)}} = |\Omega_{\mathrm{d_{x\rightarrow}}}| = 8$$

follows from the fixed initial genetic code (cf. 6.5.5, p.154). Thus, strong noise meets the start of E_{P_2}. Then, $\overline{\overline{\phi(\text{`}-\text{'}, t)}}$ falls over time t, and, accordingly, the frequencies on the other target symbols rise (cf. 6.13, p.162) while $\mathrm{d_{x\rightarrow}}$ explores the code space. Eventually, for these symbols, the first and second highest final mean symbol frequencies occur. These frequencies are on 'a' and '$*$' which compose the only perfect phenotype (cf. 6.9, p.153).

Thus, codes in the E_{P_2} population become more redundant (cf. 5.37, p.130) on the only problem-relevant target symbols 'a' and '$*$'. Put differently, individual genetic codes increase in beneficial redundancy (cf. 5.38, p.130).

- In conclusion, E_{P_2} verifies the code hypothesis, i.e., code evolution is operational, also corroborating the code-evolution explanation.

  Furthermore, codes become increasingly mirroring due to the pronounced rising of the 'a'- and '$*$'-frequencies. Therefore, the stronger instance of the hypothesis (cf. 6.2, p.147) has also been supported.

[5] For instance, a population is a multiset.

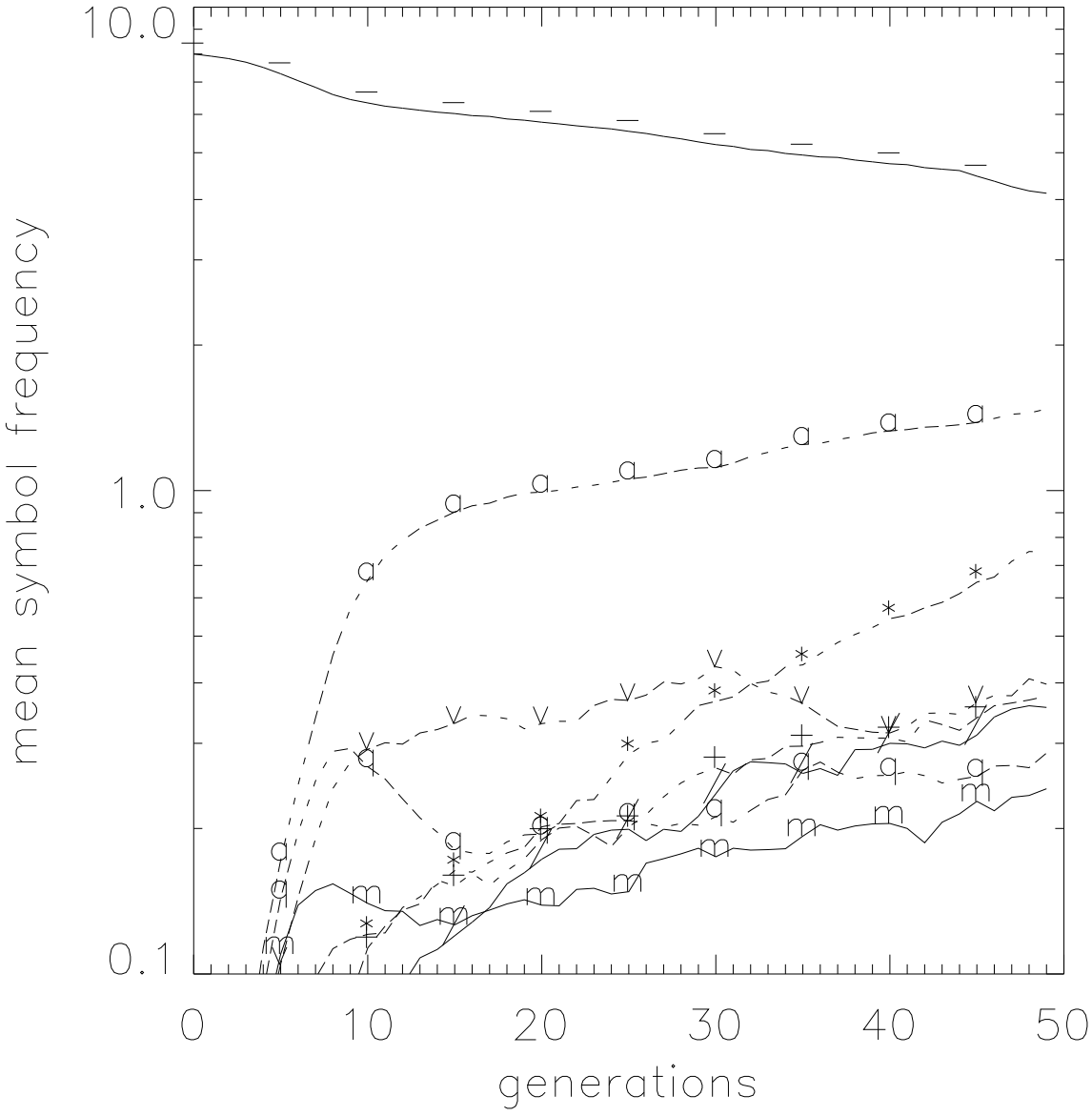

Figure 6.5: Logarithmic scale: progression of the mean symbol frequencies on all target symbols.

In particular, ‘a’ prevails in the perfect phenotype $\hat{s} = \mathsf{a} * \mathsf{a}$, which is mirrored by the top mean symbol frequency $\overline{\overline{\phi(\text{‘a’}, \omega)}} = 1.4$. In second place, $\overline{\overline{\phi(\text{‘}*\text{’}, \omega)}} = 0.74$, reflecting that ‘*’ is the only other problem-relevant symbol. Third, $\overline{\overline{\phi(\text{‘v’}, \omega)}} = 0.4$ on a problem-irrelevant symbol, and lower final frequencies follow. Last comes $\overline{\overline{\phi(\text{‘m’}, \omega)}} = 0.23$. Table 6.5 summarizes, neglecting ‘−’ and its artificially high frequency. The table gives problem-irrelevant symbols in italics.

Table 6.5: Symbol frequencies for second empirical problem.

s	a	$*$	v	..	m
$\overline{\overline{\phi(s, \omega)}}$	1.4	0.74	0.4	..	0.23

Thus, the final frequencies of the irrelevant symbols reside in a

- $0.4 - 0.23 = 0.17$ frequency range.

The absolute difference between the
i) top final frequency of the irrelevant symbols and the
ii) bottom final frequency of the relevant symbols
equals

- $|0.4 - 0.74| = 0.34$.

  As this difference is twice the frequency range, a distinct final-frequency grouping of the relevant vs. the considered irrelevant $s \in \Omega_{\mathrm{d_{x\rightarrow}}}$ exists.

- Thus, $\mathrm{d_{x\rightarrow}}$ performs a binary classification as it probabilistically decides its target alphabet regarding problem-relevance. Here, code evolution represents learning the problem-relevance of target symbols, thus identifying noise (cf. 6.7, p.151). This type of machine learning resides between unsupervised and reinforcement learning (cf. 5.2, p.108), because a genetic code is not directly tagged with quality information.

Next, we describe the

- single run of an anecdotal experiment $\boldsymbol{E'_{\mathbf{P_2}}}$ over

- 200 generations

instead of 50 generations. The run produces a

- final a-frequency: 2.9

- final $*$-frequency: 1.3

as the two single top frequencies over *all* $s \in \Omega_{\mathrm{d_{x\rightarrow}}}$. That is, for E'_{P_2}, the initial artificial top frequency of '−' eventually falls below the problem-relevant frequencies, confirming a tendency already visible in E_{P_2}.

For brevity, we define a compact form of representing a code from E_{P_2}: for $s_i \in \Omega_{\mathrm{d_{x\rightarrow}}}$, from left to right, a sequence $s_0..s_7$ shall define the code

$$\{(000, s_0), (001, s_1), .., (111, s_7)\}.$$

From E'_{P_2}, three evolved codes of good or perfect individuals and their code fitnesses follow in table 6.6.

Table 6.6: Examples of evolved codes.

Evolved $\mathsf{c} \in \gamma\mathrm{sea}_{\mathrm{d_{x\rightarrow}}}$	$\gamma\mathsf{fit}(\mathsf{c})$
`* a a a * / a -`	0.16
`* a a a a a + a`	0.16
`* a a a * / * *`	0.18

- The table illustrates an implicit effect of code evolution: An individual genetic code under a small mutation rate is a long-term memory shielded from direct selection pressure by the phenotype.

Thus, codes may remember problem-irrelevant symbols at a residual level. This is beneficial for a dynamic problem: when the relevance distribution shifts, formerly irrelevant symbols can still contribute to the synthesis of a now better phenotype. Due to the principle of code evolution, re-adaptation of codes then starts with this phenotype.

- The table also illustrates that even codes with absurdly high fitness keep irrelevant symbols, preventing the code population from being trapped.

6.6.3 Extended summary

An individual genetic code is carried by an individual (g, p_g) and controls the genotype-phenotype mapping of g onto p_g. For the second empirical problem P_2, the developmental search algorithm $\mathrm{d_{x\rightarrow}}$ adapts codes by selection on their carriers: over time, problem-relevant target symbols are increasingly used for phenotype synthesis.

- Thus, $\mathrm{d_{x\rightarrow}}$ adapts individual genotype-phenotype mappings. This renders manual creation of a beneficial problem-specific mapping obsolete, which is often costly due to incomplete problem knowledge.

In particular, code evolution adapts an individual set $S \subset \Omega_{\mathrm{d_{x\rightarrow}}}$ that $\mathrm{d_{x\rightarrow}}$ can use for phenotype synthesis. Thus, S represents the union of an individual function and terminal set. S mostly contains problem-relevant symbols. In this manner, code evolution approaches a minimal and sufficient function and terminal set, which is desirable (cf. 5.3.4, p.111).

- In summary, adding algorithmic metaphors of biological development and co-evolution to a GP algorithm enhances its autonomy (cf. 1.9, p.25).

Due to test objectives, we have designed $\mathrm{P_2}$ small and made only one perfect solution accessible to the considered $\mathrm{d_{x\rightarrow}}$ instance. Thus, next, we must discuss code evolution for a large problem. To that end, the discussion shall initially proceed along the lines of (Keller and Banzhaf 2001).

Chapter 7

Third empirical problem

- The current objective is to test whether code evolution occurs on a large problem to be given. (7.1)

- Code-fitness computation is only feasible for a small problem. Thus, we focus on observing the progression of mean symbol frequencies.

7.1 Experiment

On the small, second empirical problem P_2, final symbol frequencies are distinctly and correctly grouped regarding problem relevance. This exact classification cannot be expected on the large, synthetic problem $\mathbf{P_3}$ to be determined, at least because the required experimental run time is unavailable. (7.2)

Analogously, locating a perfect individual cannot be expected.

- However, if and only if code evolution is operational on P_3, an approximate symbol classification should occur. To test for it, we must design P_3 with problem-relevant as well as noise symbols contained in the target alphabet. (7.3)

Several properties of a problem make locating an acceptable solution within acceptable time hard for an EA run. A prominent such property is a search-space size that, by many orders of magnitude, is greater than the size of the run set, i.e., the number of all different individuals produced by the run. (7.4)

Such a large difference is implied by a *real-world* problem (cf. 2.5, p.33). Thus, designing a large P_3 turns the associated experiment $\boldsymbol{E}_{\mathbf{P_3}}$ interesting for the technical objective (cf. 1.9, p.25). Following 5.2.1, p.107, P_3 shall be a symbolic function regression of an arithmetic, random-generated objective function. In particular, all function-parameter values shall come from $[0, 1]$, and we give the third empirical function as

$$\mathbf{f_{P_3}} : \mathbb{R}^{28} \to \mathbb{R}$$

$$\begin{aligned}\mathtt{f}_{\mathrm{P}_3}(\mathtt{A,B,a,b,..,y,z}) =& \mathtt{j+x+d+j*o+e*r-t-a+h-k*u+a-k-}\\ & \mathtt{s*o*i-h*v-i-i-s+l-u*n+l+r-j*}\\ & \mathtt{j*o*v-j+i+f*c+x-v+n-n*v-a-q*}\\ & \mathtt{i*h+d-i-t+s+l*a-j*g*v-i-p*q*}\\ & \mathtt{u-x+e+m-k*r+k-l*u*x*d*r-a+t-}\\ & \mathtt{e*x-v-p-c-o-o*u*c*h+x+e-a*u+}\\ & \mathtt{c*l*r-x*t-n*d+p*x*w*v-j*n-a-e*b+a}\end{aligned} \tag{7.5}$$

Below, we shall discuss the role of those $\mathtt{f}_{\mathrm{P}_3}$ parameters that do not feature in the function expression. Next, we can set $\mathrm{d}_{\mathrm{x}\rightarrow}$ for E_{P_3}.

7.2 Parameters

7.2.1 Target and source alphabet

Following from $\mathtt{f}_{\mathrm{P}_3}$, $\mathrm{d}_{\mathrm{x}\rightarrow}$'s terminal set must contain $\{a, b, .., w, x\}$ that contains exactly all problem-relevant operands, i.e., those that feature in $\mathtt{f}_{\mathrm{P}_3}$'s expression.

$\{A, B, y, z\}$ contains all remaining operands from $\mathtt{f}_{\mathrm{P}_3}$'s parameter set: they are noise symbols.

- Thus, the terminal set shall be the union of the two described operand sets:
$$\mathbb{T} = \{a, b, .., w, x, \quad A, B, y, z\}.$$
- To provide for noise for $\mathrm{d}_{\mathrm{x}\rightarrow}$'s function set, we define
$$\mathbb{F} = \{+, -, *, \quad /\}$$
with / as noise symbol.
- Target alphabet
$$\Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}} = \mathbb{F} \cup \mathbb{T}$$
results with $|\Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}}| = 32$, containing five noise symbols.

Next, we can determine $\mathbb{A}_{\mathrm{P}_3}$ depending on $\Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}}$.

- A codon size of
$$\log_2(|\Omega_{\mathrm{d}_{\mathrm{x}\rightarrow}}|) = 5$$
bits, at least, results to have code evolution produce potentially surjective individual genetic codes (cf. 4.37, p.85).

  For E_{P_3}, we fix the codon size at five bits: this value already implies the mandatory, large search space, as will be argued later.
- $\mathbb{A}_{\mathrm{P}_3} = \mathbb{B}^5$ results as source alphabet.

We discuss the implied code space.

7.2.2 Code space

Given source alphabet $\mathbb{A}$ and target alphabet Ω for a corresponding algorithm $a_{\rightarrow}$, the size of the resulting code space is

$$|\gamma\text{sea}_{a_{\rightarrow}}| = |\mathbb{A}|^{|\Omega|}.$$

With

$$|\mathbb{A}_{\text{P}_3}| = |\Omega_{\text{d}_{\text{x}\rightarrow}}| = 32,$$
$$|\gamma\text{sea}_{\text{d}_{\text{x}\rightarrow}}| = 32^{32} \approx \texttt{E}\,48.2$$

results.

Text unit 7.4, p.167, is general regarding the structure of interest. In particular, it does not distinguish between a genotype and a genetic code.

- Thus, P_3 is hard for $\text{d}_{\text{x}\rightarrow}$ regarding finding desirable genetic codes, since a run of E_{P_3} must have a run set r with $|r| \lll |\gamma\text{sea}_{\text{d}_{\text{x}\rightarrow}}|$ in order to have feasible time consumption.

Next, we can discuss $\text{sea}_{\text{d}_{\text{x}\rightarrow}}$ which depends on $\mathbb{A}_{\text{P}_3}$.

7.2.3 Search space

- We choose as
  symbolic genotype size: $|g|_s = 400$ codons,

while the length of f_{P_3}'s expression equals 200 target symbols only. This genotypic over-sizing models practice where, due to an unknown minimal length of an acceptable solution, the user may set an unnecessarily large genotype size as parameter. This action detrimentally blows up search space.

- A Developmental-Genetic-Programming algorithm may counter negative blow-up implications with a beneficially redundant genotype-phenotype mapping (cf. 5.39, p.131).

- Here, the chosen genotype size implies the desired large search space (cf. 7.1, p.167), as follows.

With codon size $c = 5$ bits and $|g|_s = 400$ codons, we get

$$|\,\text{sea}_{\text{d}_{\text{x}\rightarrow}}| = 2^{c\cdot|g|_s} \approx 10^{\mathbf{602}}.$$

- Thus, P_3 may be hard for $\text{d}_{\text{x}\rightarrow}$ regarding locating an acceptable genotype, for the analogous reason that code search is hard (s. above).

$\vec{\text{mut}}_{\mathbb{B}}$ is $\text{d}_{\text{x}\rightarrow}$'s only genotypic-variation operator (cf. 6.1, p.147). Thus

$$\mathsf{dim}(\text{sea}_{\text{d}_{\text{x}\rightarrow}}) = \log_2(|\,\text{sea}_{\text{d}_{\text{x}\rightarrow}}|) = c\cdot|g|_s = 2{,}000$$

results as dimensionality which is, like the search-space size, rather large.

Next, we shall determine further experiment parameters.

7.2.4 Problem instance and size parameters

Due to $\mathsf{f}_{\mathrm{P}_3}$, a fitness case is from $[0,1]^{28} \times \mathbb{R}$. Let the

- training set consist of 100 random-generated fitness cases. A
- population size of $\mathsf{p} = 1,000$ individuals shall be given for all E_{P_3} runs.
- E_{P_3}'s size shall be $\mathsf{e} = 30$ runs, and
- run size shall be exactly $\mathsf{r} = 200$ generations:

  there is no quality-dependent run termination, so that phenomena of interest can be measured until a time-out occurs after the production of the 200th generation.

The union of all run sets for a given experiment E shall be called the experiment set. E_{P_3}'s experiment set contains at most $\mathsf{e} \cdot \mathsf{r} \cdot \mathsf{p} = \mathbf{6 \cdot 10^6}$ individuals. In particular, each run set contains at most $\mathsf{rp} = \mathbf{2 \cdot 10^5}$ individuals. Thus,

$$|\,\mathrm{sea}_{\mathrm{d_{x\rightarrow}}}\,| \gg |\gamma\mathrm{sea}_{\mathrm{d_{x\rightarrow}}}\,| \gg \mathsf{erp} > \mathsf{rp}.$$

- Therefore, for $\mathrm{d_{x\rightarrow}}$, P_3 has a real-world quality (cf. 7.4, p.167) regarding locating desirable genetic codes and genotypes.

Next, we discuss further E_{P_3} parameters.

7.2.5 Operator execution probabilities

We determine the execution probabilities of the operators for variation and copying.

Table 7.1: Execution probabilities for variation and copying for the third empirical problem.

$\mathrm{p_x}(\vec{\mathrm{r}}\mathrm{cop})$	0.85
$\mathrm{p_x}(\vec{\mathrm{r}}\mathrm{mut}_{\vec{\mathbb{B}}})$	0.12
$\mathrm{p_x}(\vec{\mathrm{r}}\mathrm{mut}_{\dot{\gamma}})$	0.03

Note that the point code-mutation rate is only 25 percent of the point-mutation rate. For a discussion of differences in the determined rates, see 6.5.4, p.153.

7.2.6 Initial individual genetic code

- The initial individual genetic codes of each E_{P_3} run shall be random-produced (cf. 6.4, p.148).

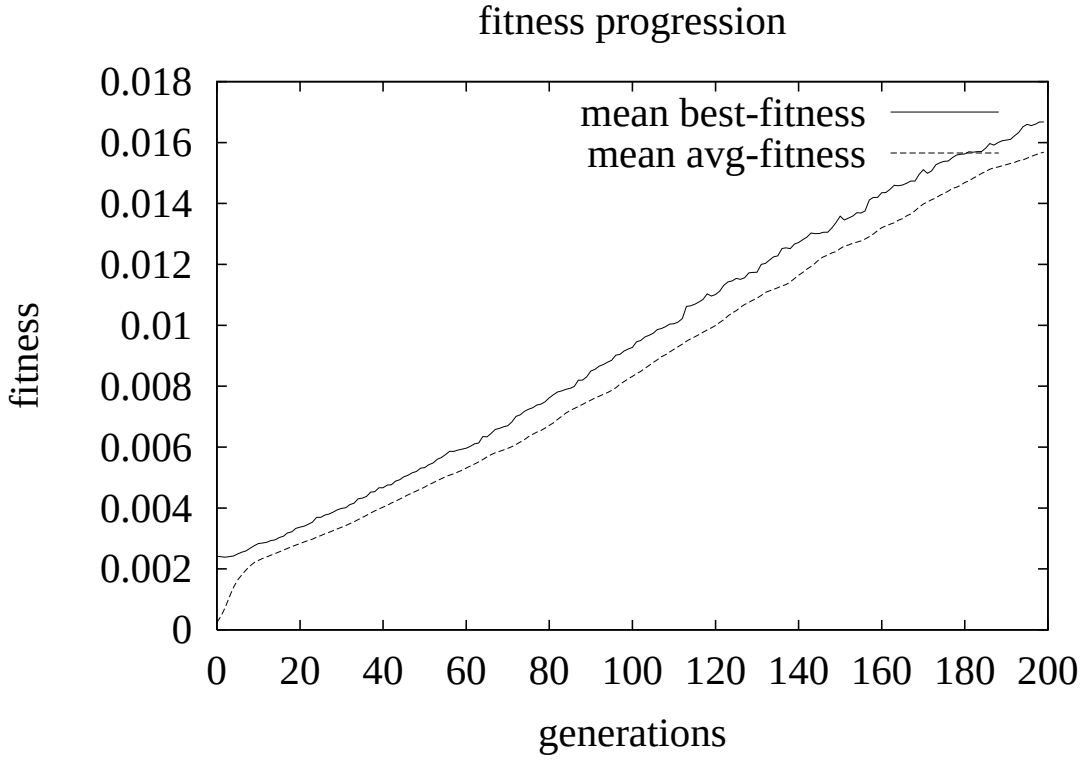

Figure 7.1: Top down: mean best individual fitness and mean average individual fitness.

Thus, each of the $|\Omega_{\mathrm{d_{x\rightarrow}}}|$ mean symbol frequencies on $s \in \Omega_{\mathrm{d_{x\rightarrow}}}$ is expected to equal about

$$\overline{\overline{\phi(s,0)}} = 1$$

in the initial generation: we expect an initial code to be about neutral on each target symbol (cf. 6.12, p.159). In this manner, an initial code population does not represent knowledge on the problem-relevance of symbols, which contributes to $\mathrm{P_3}$'s practical character.

- In summary, $\mathrm{P_3}$ has a real-world character for $\mathrm{d_{x\rightarrow}}$ regarding finding genotypes and individual genetic codes.

We discuss $E_{\mathrm{P_3}}$'s outcome next.

7.3 Results and discussion

Top down, figure 7.1 visualizes the observed progression of the mean best and the mean average individual fitness, respectively.

- Both progressions show positive progress (s. def.), which, due to the previously corroborated CEE, predicts code evolution.

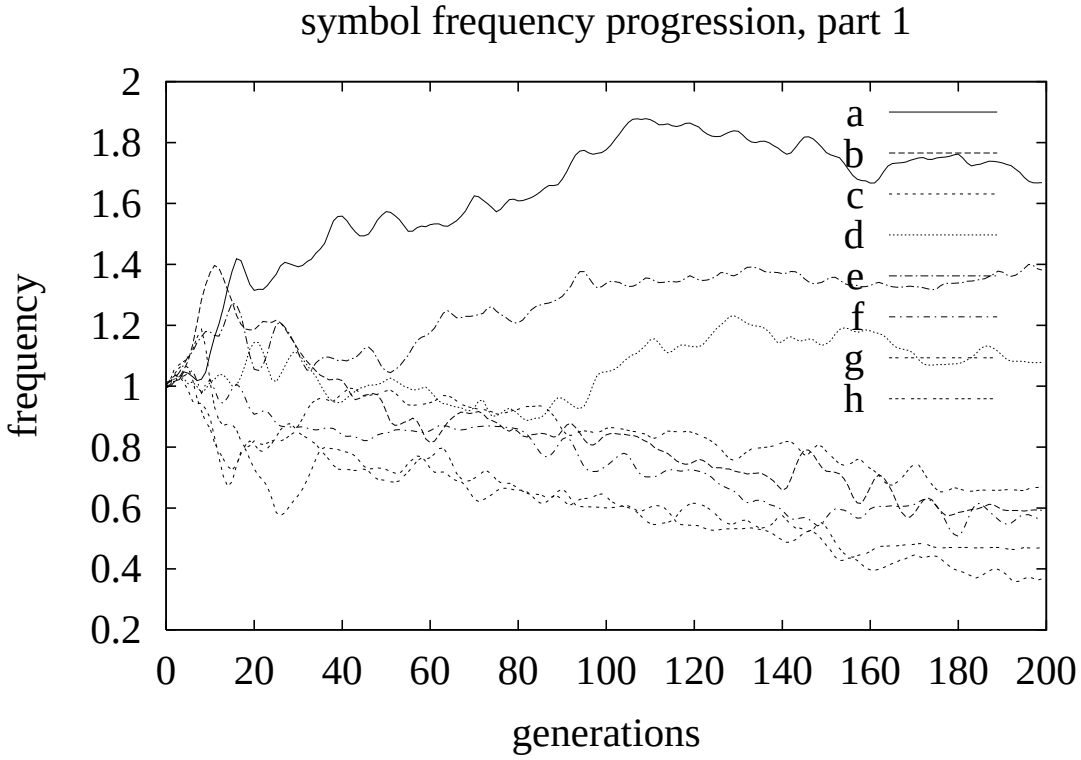

Figure 7.2: Observed progressions of the mean symbol frequencies on the mentioned target symbols.

In particular, after `generation` 5, progress of both progressions is about constant until and including the final generation, indicating a stable and continuing genotype evolution.

Figure 7.2, figure 7.3, figure 7.4, and figure 7.5 synoptically show the frequency progression — short for the progression of the mean symbol frequency $\overline{\overline{\phi(s,t)}}$ — for each $s \in \Omega_{\mathrm{d_{x\rightarrow}}}$. For legibility, each figure gives the progressions for eight symbols only.

$0 \leq \overline{\overline{\phi(s,t)}} \leq |\mathbb{A}_{\mathrm{P_3}}| = 32$ follows. (7.6)

- According to expectation (cf. 7.2.6, p.170), for $s \in \Omega_{\mathrm{d_{x\rightarrow}}}$, its frequency progression starts at $\overline{\overline{\phi(s,0)}} \approx 1$: an unbiased initial situation is given (cf. 6.12, p.159) as desired.

  For a later generation $\mathtt{t} > 0$,

  $$\overline{\overline{\phi(s,\mathtt{t})}} < 1$$

  signifies less problem-relevance of s: the virtual code-generation $\mathtt{t}$ represents the collective memory of the search process since $\mathtt{t} = 0$ and classifies s as less important. (7.7)

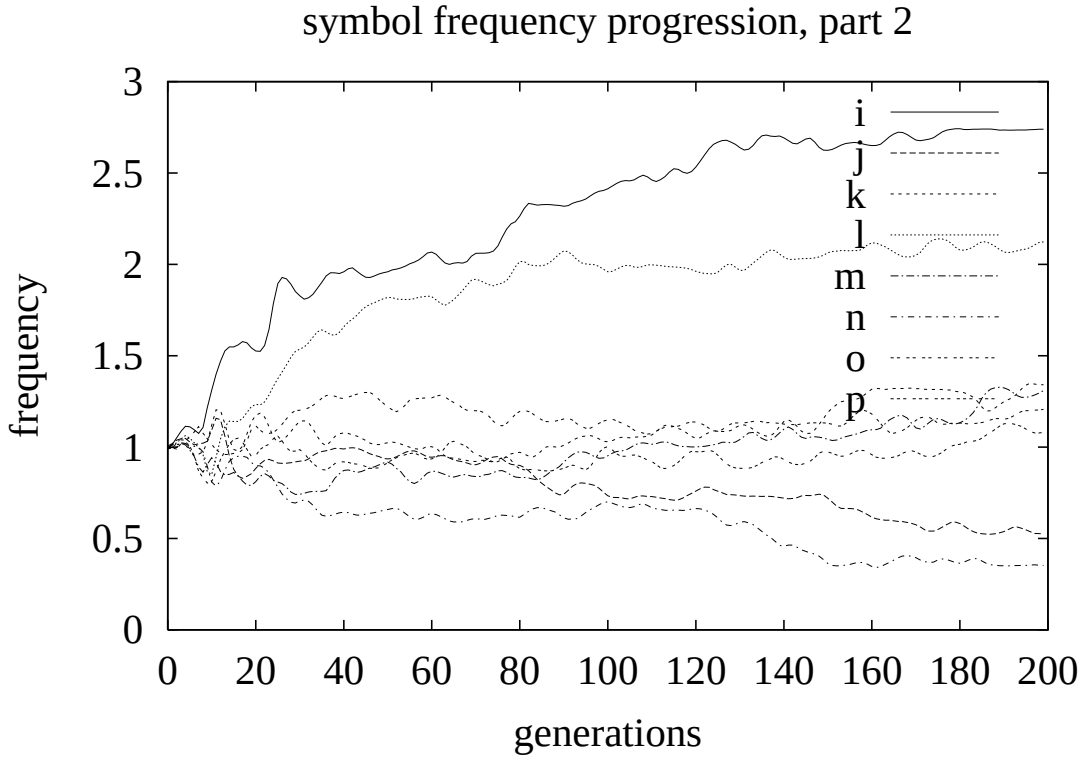

Figure 7.3: As before.

After `generation` 5, the progress of both *fitness* progressions is positive and about constant. Notably however, earlier, the mean-*best* fitness progress is virtually zero. Often, for an EA experiment with random-only individuals at $\mathtt{t} = 0$, the mean-best fitness immediately rises. This is because, initially, it is likely that, soon, an individual is located that is the best-so-far individual, because random structures do not represent problem knowledge. The present lack of this ubiquitous observation emphasizes the hardness of P_3: producing individuals that are better than the best initial individuals consumes several generations.

Synchronized with fitness progression, the behavior of *early* frequency progressions is of interest: the majority of the progressions begins fanning out, or, spreading out, between $\mathtt{t} = 5$ and $\mathtt{t} = 10$.

- If code evolution is present — it is, as one will see — then the fan-out complies with and thus corroborates the CEE: the fan-out visualizes large shifts in the symbol frequencies of codes that, in turn, foster locating better individuals.

The CEE postulates co-operative co-evolution (cf. 6.5, p.151). Thus, fitness-progress change and fanning out mutually predict each other. Such duality is typical of co-evolving structures of interest, and, in particular, co-evolution begins in all structure populations almost simultaneously.

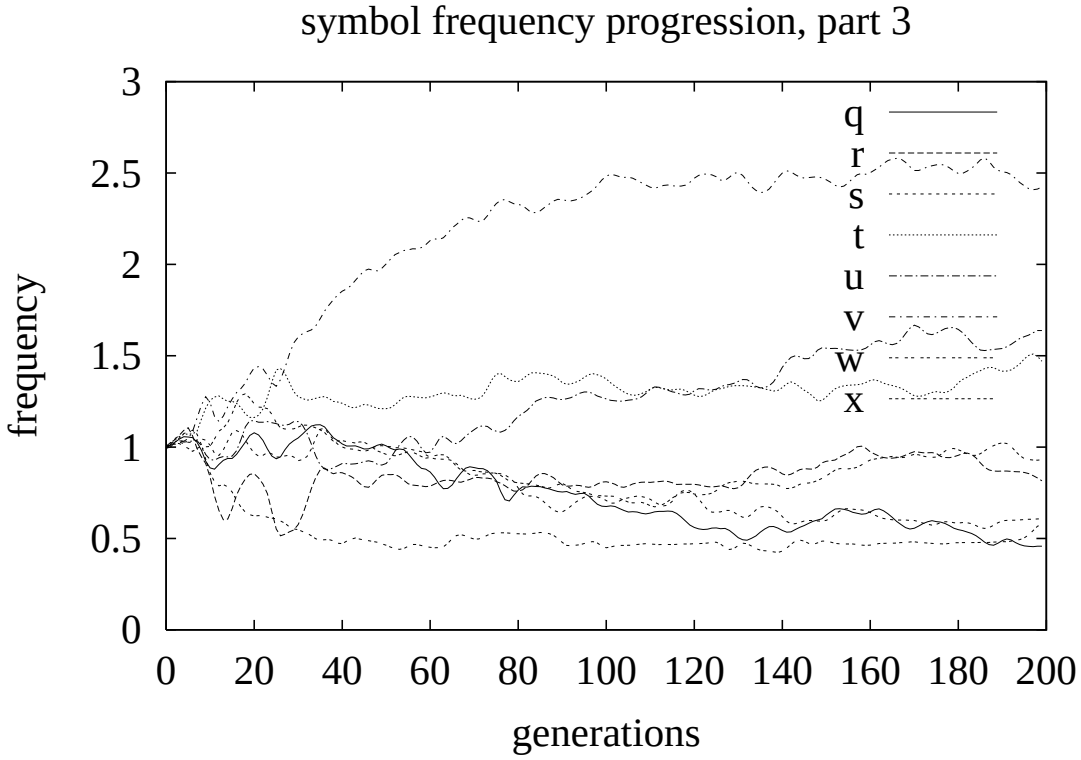

Figure 7.4: As before.

In the present case, on the one hand, an above-average genotype may result from a mutation, leading to a proliferation of the associated individual genetic code. On the other hand, an above-average code may come from a code mutation, leading to a proliferation of the associated genotype.

After strong frequency oscillation during initial fanning out, the frequency distribution begins stabilizing. Initial oscillation phenomena are typical for learning processes, where, after an early exploratory period, a phase of exploitation sets in. Such fluctuating progress of observables indicates early instability before the system may ease into a dynamic equilibrium. For the instance of frequency progressions, we may explain the observed oscillations as follows.

Vividly, code evolution implies a conflict among instances of target symbols that compete for limited resources: i) 32 positions in a code, and ii) a fixed number of codes. Formally, this conflict is reflected in (cf. 6.13, p.162).

In particular, if the frequency on a less problem-relevant $s \in \Omega_{\mathrm{d}_{\mathrm{x}}\rightarrow}$ rises, this means an increase in the number of s-instances in the code population. This, in turn, limits the potential of s-containing codes and their carriers to prevail in the long term, because s is less important. Thus, after some time, a more relevant symbol may rise in frequency, having the s-frequency sink. In total, the s-frequency first rises and then sinks, starting its oscillation. Eventually, a phase of leveling out of all frequency progressions begins.

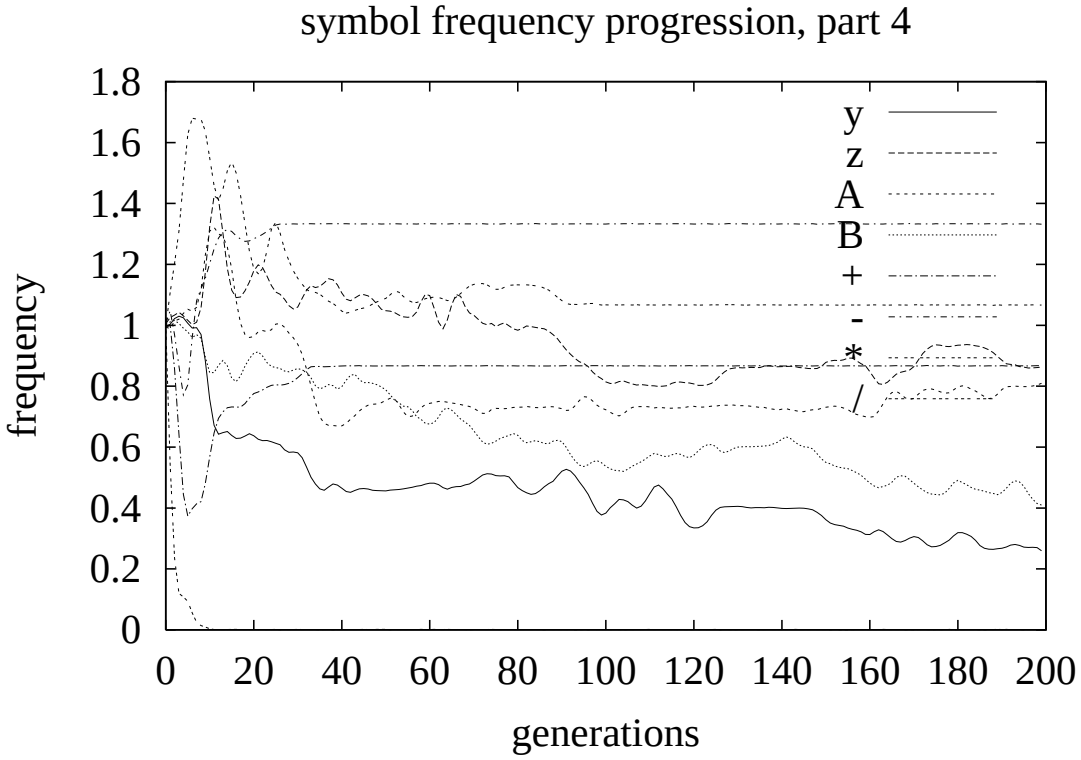

Figure 7.5: As before.

Table 7.2, p.176, lists all $s \in \Omega_{\mathrm{d_{x\rightarrow}}}$ sorted by their rounded $\overline{\overline{\phi(s,\omega)}}$ values, with $\omega = 199$ for $E_{\mathrm{P_3}}$.

Given perfect classification by $E_{\mathrm{P_3}}$, a noise symbol s would score (cf. 7.7, p.172)

$$\phi = \overline{\overline{\phi(s,\omega)}} = 0,$$

and a relevant symbol would give

$$\phi = \overline{\overline{\phi(s,\omega)}} \geq 1.$$

- However, ϕ is a mean average emerging from a search process. Thus, there is a window of uncertainty $]0,1[$: ϕ closer to zero indicates that $E_{\mathrm{P_3}}$ considers s as probably less relevant.

- Accordingly, $[0,0]$ and $[1,32]$ shall be called a window of certainty each.

- Therefore, in order to assess $E_{\mathrm{P_3}}$'s learning success, we must identify a frequency range

$$[\mathbf{0}, \phi^{\mathbf{T}}],\ 0 \leq \phi^T < 1,$$

by fixing a $\boldsymbol{\phi^T}$ value.

Table 7.2: Final mean symbol frequencies of the third empirical problem. Noise and operator symbols are marked ×.

$s \in \Omega_{\mathrm{d_x}\rightarrow}$	$\overline{\overline{\phi(s,\omega)}}$	irrelevant	operator
/	0	×	×
y	0.2	×	
B	0.4	×	
c	0.4		
g	0.5		
j	0.5		
n	0.5		
q	0.5		
b	0.6		
f	0.6		
s	0.6		
w	0.6		
h	0.7		
A	0.8	×	
r	0.8		
+	0.9		×
d	0.9		
x	0.9		
z	0.9	×	
*	1.1		×
p	1.1		
o	1.2		
-	1.3		×
k	1.3		
m	1.3		
e	1.4		
t	1.5		
u	1.6		
a	1.7		
l	2.1		
v	2.4		
i	2.7		
$\sum$	$32 = \lvert\Omega_{\mathrm{d_x}\rightarrow}\rvert$	5	4

Thus, ϕ^T has this meaning:

- $\phi^T < \overline{\overline{\phi(s,\omega)}} \quad \Leftrightarrow \quad$ We consider E_{P_3} to classify s as "relevant".

Therefore, the smaller ϕ^T is, the better E_{P_3} must approximate perfect classification regarding noise symbols to avoid classification error. Analogously, a greater ϕ^T value implies more rigor regarding relevant symbols.

- Thus, for a balanced compromise, classification error should be least for $\phi^T \approx 0.5$. (7.8)

As worst case, giving a noise symbol a frequency $\phi \geq 1$ or giving a relevant symbol $\phi = 0$ constitutes an absolute error.

To the end of assessing classification quality, we define

- $\boldsymbol{N} := \{A, B, y, z, /\} \subset \Omega_{\mathrm{d_{x\rightarrow}}}$, $|N| = 5$, the set of noise symbols,
- $\boldsymbol{R} := \Omega_{\mathrm{d_{x\rightarrow}}} \setminus N$, $|R| = 27$, the set of relevant symbols, and,
- n and r, the numbers of *correctly* recognized noise and relevant symbols, depending on ϕ^T.

$$\text{Let} \quad \mathfrak{c} := \frac{\frac{n}{|N|} + \frac{r}{|R|}}{2}$$

denote the classification ratio. $0 \leq \mathfrak{c} \leq 1$ follows, and the better E_{P_3} approximates perfect classification, the closer $\mathfrak{c}$ is to one. Especially, the ratio weighs the approximation quality equally regarding noise and relevant symbols.

Table 7.3, p.177, gives the rounded ratios for ϕ^T values that range over zero and the uncertainty window in 0.1-steps.

Table 7.3: Symbol-classification ratios.

ϕ^T	n	r	$\mathfrak{c}$
0.0	1	27	0.6
0.1	1	27	0.6
0.2	2	27	0.7
0.3	2	27	0.7
0.4	3	26	0.8
0.5	3	22	0.7
0.6	3	18	0.6
0.7	3	17	0.6
0.8	4	16	0.7
0.9	5	13	0.7

- For $\phi^T = 0$, implementing maximal rigor, a ratio of still $\mathfrak{c} = 0.6$ results.
- The sole instance of the maximum $\mathfrak{c} = 0.8$ occurs at $\phi^T = 0.4$, which represents a good compromise (cf. 7.8, p.177).
- All five noise symbols reside in $[0, 0.9]$ that is the lower third of the total window $[0, 2.7]$ containing all emerged frequencies.
- In particular, 60% of all noise falls into $[0, 0.4]$ that resides within the lowest sixth of the total window.
- Especially, the noise symbols in $\{B, y, /\}$ score the three lowest frequencies, with '/' at virtually zero, on average eliminating this operator from a population.
- Thus, E_{P_3} places all noise symbols in the uncertainty window and zero, avoiding an absolute error.
- Likewise, it does not eliminate a relevant symbol, giving the minimal relevant frequency $\overline{\overline{\phi(\text{`}\mathsf{c}\text{'}, \omega)}} = 0.4$.

Given more run time, further classification improvement is likely, considering the unbroken positive fitness progress that indicates continuing search dynamics.

We note a conspicuous feature of all *operator* progressions, i.e., they abruptly experience freezing: discontinuously, progress changes permanently to practically zero. For brevity, we shall also speak of freezing symbols. Table 7.4, p.178, gives the freezing time for each operator, marking those that freeze in their correct certainty window.

Table 7.4: Freezing time for all operators.

Operator	$*$	$+$	$-$	$/$
Generation	90	33	28	9
True certainty	×		×	×

Freezing is only observed for operators, with the sole reliable exception of operand i at `generation 187`. Notably, $\overline{\overline{\phi(\mathsf{i}, \omega)}}$ is the maximum of all emerged frequencies. Also, the only noise operator, '/', freezes first.

Eventually, E_{P_3} perfectly classifies three of all four operators, i.e., their final frequencies lie in their correct certainty windows. However, for operands, perfect classification percentage is lower, and E_{P_3} scores 11 of 28, i.e., ca. 40%.

- In conclusion, E_{P_3} delivers perfect classification easier for operators than for operands. While such classification requirement is unrealistic for a real-world problem, the result indicates a trend that is also valid for less rigorous classification assessment, i.e., $\phi^T > 0$.

We offer an explanation of this dichotomy between operators and operands. For $\mathsf{f}_{\mathrm{P}_3}$ and its parameter values, a mutation of an operator often changes the mutant's expression value stronger than an operand mutation does.

- Thus, we conjecture there is higher selection pressure regarding operators that therefore settle in local optima sooner than operands. With frozen operators, E_{P_3} then continues fitting the operands. Observations support the conjecture:

  At `generation 9`, '/' freezes, and, simultaneously, the `i`-progression starts its first strong rise, moving away from the nearby uncertainty window.

  At `generation 28`, '−' freezes, and, at `generation 29`, the `A`-progression starts dropping, crossing from its false certainty window into more desirable uncertainty.

  At `generation 33`, '+' freezes, and, simultaneously, the `a`-progression experiences its first strong rise after initial oscillation, going further into its correct certainty window.

  At `generation 90`, '$*$' freezes, and, simultaneously, the `i`-progression, having had stalled since `generation 80`, starts rising again, going to the maximum value over all progressions which also is in `i`'s correct certainty window.

- In summary, $\mathrm{d}_{\mathrm{x}\rightarrow}$ yields code evolution that is operational on the large third problem P_3 (cf. 7.3, p.167) which has real-world quality regarding evolving genotypes and individual genetic codes.

- Occurring code evolution is in accordance with the observed fitness progression, corroborating the co-evolutionary nature of code and genotype evolution as postulated by the code-evolution explanation.

Also, the steadily rising fitness progression means that the relevant yet uncertainly classified symbols are not essential for improving fitness, at least during the observation window and for the given fitness cases.

- Thus, in practice, a $\mathrm{d}_{\mathrm{x}\rightarrow}$ instance may be able to ignore some relevant symbols, thus focusing on a proper subset of the solution space where it may still find an acceptable solution.

Also, we have observed a solidifying of the frequency distribution, which may be an analogy to the frozen thus universal genetic code in nature.

- We interpret the observed dynamics in the frequency progressions as implicit competitive and co-operative co-evolution:

  symbol instances compete for a limited number of code entries, and problem-relevant instances together in the same code instance may lead to a good carrier that promotes the symbol instances.

- Thus, $d_{x\rightarrow}$ autonomously adapts its symbol set to the given problem, which further approaches the technical objective.

At the technical core of the present work, there are: i) the model of Developmental Genetic Programming (DGP) with its geno/phenotype mapping, and ii) the automatic regulating of mapping redundancy. i) is based on genetic codes and repairing algorithms. A repairing method is the only essential DGP component that is solely concerned with phenotypes, while, so far, the focus has been on the genotypic level. Therefore, next, we consider deleting repair, dealing with the phenotypic level only. We also take this step as a prompt to consider another problem domain, and to go for some of its well-known benchmark problems.

Chapter 8

Further problems

8.1 Overview

A heuristic is a method for finding a good problem solution within acceptable time, while one cannot show that a found solution cannot be bad, or that the heuristic will always perform reasonably fast. A metaheuristic is a heuristic that uses heuristics. The label hyperheuristic (Ross 2005), see (Soubeiga 2003) for its first use, names a heuristic that roams the space of metaheuristics that approach a given problem.

For example, (Chakhlevitch and Cowling 2005) propose a method of synthesizing low-level heuristics for personnel scheduling, where useless heuristics are filtered so that one is left with practical heuristics that constitute a solver. (Burke, Kendall, and Soubeiga 2003) suggest tabu search (cf. (Glover and Laguna 1997)) on the space of heuristics for an instance of scheduling. (Gaw, Rattadilok, and Kwan 2004) present a timetabling application of a hyperheuristic with special emphasis on its parallel implementation. (Dowsland, Soubeiga, and Burke 2007) choose simulated annealing as learning method for a hyperheuristic that targets a practical example of a p-median problem. (Oltean 2005) uses Genetic Programming (GP) for producing Evolutionary Algorithms that are applied to discrete optimization. For the bin-packing problem, (Burke, Hyde, and Kendall 2006) suggest a hyperheuristic with a GP engine. While it does not add loops to the heuristics it builds, some of them emulate a hand-crafted bin-packing strategy. The approaches from the mentioned papers did not use grammars to direct the exploration of a search space, as is, instead, the case with the hyperheuristic to be presented here.

Hyperheuristics (HH) have started gaining attention as one expects them to build optimizers that are more malleable when employed for different, real-world domains. However, one cannot create a fixed HH that works efficiently in all domains (Wolpert and Macready 1997).

Given the above, we see the objective of a domain-independent framework that (i) one can specialize for a problem domain at hand, that (ii) places modest demands on its user, and that yet (iii) yields effective metaheuristics. Especially, it would be perfect if the user was only required to offer simple heuristics with which a hyperheuristic could produce effective metaheuristics within feasible time.

To that end, we suggest a hyperheuristic that takes, as input, the description of

Algorithm 1 GP-based HYPERHEURISTIC.

given: grammar G, population size p, length l
repeat
  produce next random primitive-sequence $\sigma : |\sigma| = \mathsf{l}$
  EDITING(σ,G) $\rightarrow g$ `genotype`
until p genotypes created
while time available **do**
  Selection: 2-tournament
  Reproduction: Copy winner g into loser's place $\rightarrow g'$
  Exploration: with a given probability
    Mutate copy $g' \rightarrow \delta$
    EDITING(δ,G)$\rightarrow g''$ `genotype`
end while

patterns of metaheuristics for D, an arbitrary, fixed problem domain. In principle, one can change this description in order to use the hyperheuristic on a different domain. As description, we suggest a grammar, G, that can generate sentences that are built from D's low-level heuristics and well-known metaheuristics. In this manner, $\sigma \in \mathsf{L}(G)$ defines a metaheuristic for D. Thus, any form of grammar-based GP (e.g., (Wong and Leung 1995) (Montana 1995) (Whigham 1996) (Janikow 1999) (Keller and Banzhaf 2001) (O'Neill and Ryan 2003)) that evolves programs in $\mathsf{L}(G)$ is a hyperheuristic for D. Here, we shall realize this idea as an instance of linear GP (Brameier and Banzhaf 2006).

An advantage of the introduced framework is that domain knowledge becomes a free commodity for the GP hyperheuristic that does not have to reinvent the user-given component heuristics. Moreover, by working a preference for desirable patterns into G, one can guide the search engine towards metaheuristics that hold promise.

In Section 8.2, we describe the hyperheuristic in some detail. In 8.3, we explain problems that serve as challenges to the hyperheuristic. In 8.4, we give it grammars that we experiment with in 8.5.

8.2 A repairing hyperheuristic over linear phenotypes

For a domain of interest, we represent a metaheuristic as a genotype/phenotype $g \in \mathsf{L}(G)$ with grammar $G = (N, T, P, S)$. N is the set of non-terminals, T is the terminal set, P is the set of production rules, and $S \in N$ is G's start symbol. Thus, $\mathsf{L}(G) \subset \mathbf{T}^*$, the set of all strings over T. We call a terminal $t \in T$ a primitive, and it may designate a hand-crafted metaheuristic, a low-level heuristic, or a part of them.

Executing a metaheuristic, g, with $g = i_0 i_1 ... i_n$ and $i_j \in T$, means the execution of the i_j from left to right, building a whole pattern, s, that is a possible solution to

the underlying problem. s is developed from an initial, complete structure, $i_0()$:

$$s = i_n(...(i_1(\ i_0()\ ))...).$$

g's primitives, with the exception of i_0, take a complete structure as input. All primitives of g yield a whole structure as output. Especially, i_0, in some simple manner, delivers an initial, complete structure.

We define g's fitness, i.e., its quality $\mathsf{q}(g)$, as the value of an objective function, $\mathbf{o}$, on the argument s, so that we get $\mathsf{q}(g) = \mathsf{o}(s)$.

At the start of a run of the GP hyperheuristic, given population size p, an *initialization* algorithm produces p random primitive-sequences from T^*. All such sequences have the same length, l. *Mutation* of a genotype $g \in \mathsf{L}(G)$ randomly selects a locus, j, of g, and replaces the primitive at that locus, i_j, with a random primitive, $t \in T, t \neq i_j$.

Naturally, both initialization and mutation may result in a sequence of primitives, $\sigma = i_0..i_j..i_k \in T^*$, that is not a valid genotype, i.e., $\sigma \notin \mathsf{L}(G) \subset T^*$. If this is the case, the sequence is passed to an operator, EDITING, a minimal flavor of *deleting repairing* (cf. def.). This routine attempts to turn σ into an element from $\mathsf{L}(G)$. To that end, the operator starts reading σ from left to right. If EDITING reads a primitive, p, that represents a syntax error in its current locus, EDITING replaces it with the no-operation primitive, $\mathbf{n}$. These steps are repeated until the last primitive has been processed. Then, either the current version of σ is in $\mathsf{L}(G)$, and EDITING ends, or still $\sigma \notin \mathsf{L}(G)$. In the latter case, EDITING keeps repeating the above steps on σ, but this time processing it from right to left. The result is either a $\sigma \in \mathsf{L}(G)$ or a σ that consists of $\mathtt{n}$-instances only. In this latter, unlikely case, EDITING then assigns the lowest available fitness value to σ. This way, σ will most likely disappear from the population during selection. The selection scheme used by the GP hyperheuristic is "tournament." We summarize the hyperheuristic in Algorithm 1.

Note that we initialize the population using primitive-sequences of a fixed length, l. However, over time, the application of the EDITING operator effectively leads to a population containing genotypes of *variable* lengths, which in particular allows the evolution of parsimonious heuristics.

8.3 Problem domain

To study the GP hyperheuristic, we select a certain NP-hard domain from discrete optimization, i.e., the set of traveling-salesperson problems (TSP) (Lawler, Lenstra, Kan, and Shmoys 1985). In its simplest form, a TSP involves finding a minimum-cost Hamiltonian cycle ("tour") in a given, weighted graph. Let the n nodes of such a graph be numbered from 0 to $n-1$. Then, one describes a tour involving edges (v_0, v_1), (v_1, v_2), ..., (v_{n-1}, v_0) as a permutation $p = (v_0, ..., v_{n-1})$ over $\{0, ..., n-1\}$. We call permutation $(0, 1, ..., n-1)$ the *natural* Hamiltonian cycle of the graph ("natural cycle"). In the following sections, for empirical studies, we will use 600 symmetrical, Euclidean, random-generated problems of ten nodes. The weight of an

```
metaheuristic       ::= NATURAL
                      | NATURAL search

search              ::= heuristic
                      | heuristic search

heuristic           ::= NATURAL
                      | 2-CHANGE
```

Figure 8.1: Grammar N2O

edge (i, j) is the distance between i and j. To demonstrate the practical applicability of the GP hyperheuristic (GP HH), we will also use TSP problems of more realistic sizes.

8.4 Target languages

8.4.1 Grammar *N2O*

We start by providing the GP HH with two low-level heuristics only: `NATURAL` that creates the natural cycle for a problem, and `2-CHANGE` which executes a minimal change of a tour H into a different tour. That is, given two of H's edges—$(a, b), (c, d) : a \neq d,\ b \neq c$ — `2-CHANGE` replaces them with $(a, c), (b, d)$. When the hyperheuristic, executing an evolved metaheuristic, encounters a `2-CHANGE` primitive as component of the metaheuristic, the HH randomly selects two appropriate edges, (a, b) and (c, d), from all edges of the underlying tour. The hyperheuristic then gives these edges as arguments to `2-CHANGE` and executes this primitive.

The primitives `NATURAL` and `2-CHANGE` are terminal symbols for a grammar, which we call *N2O*, that guides the HH in creating and mutating metaheuristics. We give *N2O* in pseudo Backus-Naur form (BNF) (Control Data Corporation 1979) in Figure 8.1. We shall employ the top two rules of *N2O*, `metaheuristic` and `search`, in several upcoming grammars where these rules shall be represented by "**Preamble**". Note that there is no implicit beneficial bias in $\mathsf{L}(N2O)$, the language generated by *N2O*. For instance, the sequence
`NATURAL 2-CHANGE 2-CHANGE NATURAL`, where the final execution of `NATURAL` destroys whatever improvement may have been made by the repeated execution of `2-CHANGE`, is in $\mathsf{L}(N2O)$. Also, we have not provided a non-destructive local heuristic, for instance, one that only executes `2-CHANGE` if this results in a shorter cycle. So, the system has no domain-specific, built-in intelligence, and has no way of evolving it over the given grammar. Can the GP hyperheuristic still create an effective metaheuristic, specific to the given instance of a problem? We will see this in Section 8.5.1.

```
Preamble

heuristic   ::= NATURAL
              | 2-CHANGE
              | IF_2-CHANGE /* added */
```

Figure 8.2: Grammar If

```
Preamble

heuristic           ::= 2-CHANGE
                      | IF_2-CHANGE
```

Figure 8.3: Grammar NoNatural

8.4.2 Grammars *If* and *NoNatural*

Next, we extend grammar $N2O$, giving a grammar we call *If*, shown in Figure 8.2. `IF_2-CHANGE` is a conditional `2-CHANGE` that executes `2-CHANGE` only if the execution will shorten the tour under construction. As every greedy operator, `IF_2-CHANGE` is a boon and a curse, but its introduction is safe here since the grammar holds a randomizing counterweight in the form of `2-CHANGE`.

Hoping to define an even more apt language, we do away with `NATURAL` in rule `heuristic`, and we define grammar *NoNatural* shown in Figure 8.3.

8.4.3 Grammar ThreeChange

We extend the target language by a further low-level TSP heuristic that is known as a *3-change*: delete three mutually disjoint edges from a given cycle, and reconnect the resulting three paths such that a different cycle is obtained.

Given this method, we define the heuristic `IF_3-CHANGE`: 1) randomly select edges as arguments for `3-change` 2) if `3-change` will better the cycle, execute `3-change`.

Since `IF_3-CHANGE` introduces a greedy bias, it may or may not be helpful to provide a counter-bias, for instance by allowing for an occasional worsening of a cycle. We decide in favor of making this step available, and to this end we define the heuristic `IF_NO_IMPROVEMENT`:
if none of the latest $1,000$ individuals produced has found a better best-so-far tour, execute a `2-change`.

We add `IF_3-CHANGE` and `IF_NO_IMPROVEMENT` to the previous grammar and call the resulting one *ThreeChange*. It is shown in Figure 8.4.

8.4.4 Grammar NoNoImprove

A small language is desirable, as it implies a small search-space size for the hyper-heuristic. To understand whether the heuristic `IF_NO_IMPROVEMENT` does or does

```
Preamble

heuristic   ::= 2-CHANGE
              | IF_2-CHANGE
              | IF_3-CHANGE
              | IF_NO_IMPROVEMENT
```

Figure 8.4: Grammar ThreeChange

```
Preamble

heuristic           ::= 2-CHANGE
                      | IF_2-CHANGE
                      | IF_3-CHANGE
```

Figure 8.5: Grammar NoNoImprove

not improve the effectiveness of the GP hyperheuristic, we remove it from grammar *ThreeChange*. We call the resulting grammar and its language *NoNoImprove*. The grammar is shown in Figure 8.5.

8.4.5 Grammar DoTillImprove

So far, only sequential and conditional execution of user-provided heuristics are available to evolved metaheuristics. Therefore, a loop element is required for completing the set of essential control structures. To that end, we introduce the primitive

`REPEAT_UNTIL_IMPROVEMENT` p: execute primitive p until it has lead to a shorter cycle or until it has been executed ι times for user-given ι.

An example for the use of the primitive `REPEAT_UNTIL_IMPROVEMENT` in a grammar, *DoTillImprove*, is shown in Figure 8.6. Note that this grammar allows for evolving a metaheuristic that represents one of a few simple search types, such as memory-less iterated local search (Lourenco, Martin, and Stutzle 2002), greedy search, or random search, without enforcing the evolution of only one such type. This is because the grammar contains primitives that represent elements of such search types, but it does not describe primitive-sequences of exclusively one such type. Thus, at any point during a hyperheuristic run, the population may contain metaheuristics that represent different search types, while there is only one underlying grammar.

8.5 Experiments

Initially, we are interested in whether or not, on average, for small problems from the chosen domain, an evolved metaheuristic does at least better than blind search,

```
Preamble

heuristic ::= 2-CHANGE
            | loop IF_2-CHANGE
            | loop IF_3-CHANGE

loop      ::= REPEAT_UNTIL_IMPROVEMENT
            | /* empty */
```

Figure 8.6: Grammar DoTillImprove

the minimal approach to any problem. This is to see whether the hyperheuristic manages to acquire any domain information, at all, and to represent it in the structures of evolved metaheuristics. Later, we shall go for bigger problems. There, a comparison with the second-simplest approach, a pure hill climber, is not ambitious, as it performs poorly on larger TSPs. Indeed, literature ignores it. Instead, we shall compare evolved solvers to a state-of-the-art hybrid Genetic Algorithm.

Here, we compare the search performed by evolved metaheuristics to a form of blind search that operates on the solution space of a problem to be given to the hyperheuristic. For this purpose, blind search creates $n > 0$ random permutations, with n to be given. That is, each of the n search steps creates a random cycle, i.e., a candidate solution.

For a meaningful comparison in terms of efficiency, one must ensure that the processes to be compared perform approximately equal numbers of their respective elementary operations. Thus, for the present purpose, we must set $n = 10$ for blind search, as this number equals the genotype size which equals the number of simplest search steps that an evolved metaheuristic performs during its evaluation. In this manner, the best of the n cycles found by a run of blind search can be compared with the single cycle constructed by n calls of heuristics as orchestrated by a metaheuristic.

To compare an achieved tour length l to a given length L, we define $\delta(l, L) = 1 - l/L$. Thus, for l and m with $\delta(l, L) > \delta(m, L)$, l is a better result than m. Also, $\delta(l, L) < 0$ implies that l is worse than L, a δ value of zero indicates $l = L$, and positive values signal an improvement.

8.5.1 Grammar *N2O*

For the small TSP problems, we will employ a restricted version of the GP hyperheuristic where all produced primitive-sequences shall be genotypes of the same size. For the larger problems, we shall drop this limitation.

The genotype size shall be $\mathsf{l} = 10$. The initial genotypes shall all be identical: $g =$ `NATURAL`...`NATURAL`, $|g| = \mathsf{l}$. Thus, initialization of the population of a hyperheuristic run does not effect random search, so that any discovered, good individual must have come from evolutionary search alone.

Popul. size	Genotype size	Offspring	Mut. prob.
10	10	2,500	0.3

Table 8.1: Basic Parameters

Length	Mean	SD	δ
Natural	$3,827$	539	n.a.
Best	$2,654$	265	0.31

Table 8.2: Hyperheuristic, 600 10-node random problems, 1,000 runs per problem. Improvement δ relative to mean natural length. "n.a.": not applicable

For the present setup of the hyperheuristic, its random choice of an element from a set, e.g., for a mutation, shall be uniform.

We number the loci of genotypes, beginning with zero. In the context of $\mathsf{L}(N2O)$, the described setup implies that we exclude position zero from mutation, as only `NATURAL` is legal there. We set mutation probability to 0.3 (cf. Algorithm 1, step 9). The GP hyperheuristic shall measure time in terms of the number of individuals produced after creation of $\mathsf{p} = 10$ initial individuals. We set this offspring number to 2,500 (cf. Algorithm 1, step 6). Table 8.1 reports the used parameters.

On a single, random-generated ten-node problem, P, we observe tour lengths found by evolved metaheuristics. These lengths are, on average over 1,000 independent runs, better by 0.3 [δ] than P's natural length, opposed to 0.2 for random search.

Interestingly, the hyperheuristic produces such metaheuristics despite the fact that the underlying primitives do not allow for direct use of problem-specific information. For example, an informed decision on the choice of arguments to a `2-CHANGE` execution is not an option. To establish whether this phenomenon is independent from P, we perform 1,000 independent runs of the hyperheuristic for each of 600 randomly created, ten-node problems, collecting the natural lengths of these instances and the lengths of the shortest tours found by the runs. Table 8.2 shows the results.

We notice a mean improvement of about 0.31, so that the good performance of the GP hyperheuristic, observed for P, appears to be rather general.

Let us shift the focus from the problem space to the metaheuristic space. We therefore consider, for P, both the GP hyperheuristic and pure random search in the space of metaheuristics, i.e., $L(N2O)$. As the given genotype size equals ten, the genotype that starts with `NATURAL`, followed by the longest possible unbroken chain of nine `2-CHANGE` calls, offers most opportunities for improving a tour. On P, the 1,000 runs of the hyperheuristic locate this genotype 14,171 times, while the expected value for 1,000 runs of 2,510 random samples each is merely

$1,000 \cdot 2,510/2^9 \approx 4,902.$

Table 8.3 collects the values.

Approach	Hits
GP hyperheuristic	14,171
Random	4,902

Table 8.3: GP hyperheuristic v ideal random search on $\mathsf{L}(N2O)$. Hits of optimal genotype.

Popul. size	Genotype size	Offspring	Mut. prob.
10	20	5,000	0.3

Table 8.4: Basic Parameters

8.5.2 Grammars N2O, *If* and *NoNatural*

For further experiments, we change some of the basic parameters. The genotype size, which equals the number of times an evolved metaheuristic calls a provided heuristic, shall be twenty. 5,000 offspring shall be produced per run. Table 8.4 reports the used parameters.

For a comparison of the resulting behaviors of the hyperheuristic when it is fed with the different grammars, we consider forty symmetrical, Euclidean, random-generated problems of ten nodes. On each problem, the HH performs 500 independent runs. Regarding the shortest tour found during a run, we give means and standard deviations (SD) over all runs over all problems, rounded to the nearest integer. Table 8.5 summarizes the results of the experiments. Its first line gives the values over the natural lengths of the random problems.

The improvements in the mean-best values show that one can improve performance of the GP hyperheuristic by increasing expressiveness of the target language L, or by changing a bias in L, as we have done over the series of grammars discussed in Section 8.4.

Language *NoNatural*, the one of the three giving the best results, is still very simple in terms of the structural and functional sophistication of its sentences when one compares them to procedures used by state-of-the-art TSP solvers. We therefore ask whether or not, even with this quite rudimentary language at hand, the GP hyperheuristic can still be of use given a real-world problem.

So, we consider problem `eil51` from (Reinelt 2007). Its dimension is $n = 51$ nodes, its best known solution has a length of 428.87 (Jayalakshmi, Sathiamoorthy,

40 problems	**Mean best**	SD	δ
Natural length	3,774	508	n.a.
N2O	2,600	277	0.31
If	2,530	251	0.33
NoNatural	2,423	204	0.36

Table 8.5: Performance of hyperheuristic over different languages, on 40 ten-node random problems, 500 runs per problem. Improvements δ relative to natural length.

Popul. size	Genotype size	Offspring	Mut. prob.
100	500	100,000	0.5

Table 8.6: Basic Parameters

eil51	**Mean best**	SD	Best	δ
Natural length	1,313.47	n.a.	n.a.	n.a.
NoNatural	922.00	26.96	853.73	0.3

Table 8.7: Performance for eil51 over *NoNatural*, 100 runs. Improvement δ of mean best relative to natural length.

and Rajaram 2001), with natural length of approximately 1,313.47. All values delivered by the GP hyperheuristic shall be rounded off to the nearest hundredth. We shall only be dealing with real-valued tour lengths, emphasizing differences between very good vs. excellent results. Literature often mentions integer lengths that result from measuring a distance between two nodes by rounding it to the nearest integer, which implies that tours with different real-valued lengths can have the same integer length.

For a symmetrical TSP instance, the number of tours that are different in human terms equals $(n-1)!/2$. The evolved metaheuristics operate on permutations of n nodes, so that the search-space size is $n!$. So, while the previous problem dimension, $n = 10$, yields a size of $n! \approx 3.6 \times 10^6$, $n = 51$ gives about 1.6×10^{66} search points and 1.52×10^{64} different tours. To accommodate for the larger search space, we set the basic parameters as shown in Table 8.6.

Table 8.7 gives the results obtained for `eil51`. Column "Best" gives the length of the shortest cycle found over all runs. On average, even the simple language *NoNatural* guides the GP hyperheuristic to an improvement of about 0.3 relative to the original tour.[1]

8.5.3 Grammars *ThreeChange* and *NoNoImprove*

Table 8.8 gives the results obtained with grammar *ThreeChange*. We notice another improvement in the mean-best value. This means that the combination or the individual effects of the added primitives, `IF_3-CHANGE` and `IF_NO_IMPROVEMENT`, increase the search power of the hyperheuristic.

Table 8.9 gives the results obtained using grammar *NoNoImprove*. This grammar lacks the primitive `IF_NO_IMPROVEMENT`. Interestingly, we see that removing this primitive yields even better search performance.

[1] Naturally, longer runs may produce even better results. For example, in one run with 10^6 produced offspring, the shortest tour found had a length of about 749. However, we did not perform enough such runs to be able to report reliable results here.

eil51	**Mean best**	SD	Best	δ
Natural length	1,313.47	n.a.	n.a.	n.a.
ThreeChange	874.96	26.55	810.73	0.33

Table 8.8: Performance for eil51 over *ThreeChange*, 30 runs. Improvement δ of mean best relative to natural length.

eil51	**Mean best**	SD	Best	δ
Natural length	1,313.47	n.a.	n.a.	n.a.
NoNoImprove	798.32	15.98	763.30	0.39

Table 8.9: Performance for eil51 over *NoNoImprove*, 30 runs. Improvement δ of mean best relative to natural length.

8.5.4 Grammar DoTillImprove

Successful optimization algorithms all use the iteration of effective heuristics. We therefore predict that the looping construct available in grammar *DoTillImprove*–that is otherwise identical to the previous grammar, *NoNoImprove*–will further improve the performance of the GP hyperheuristic.

Table 8.10 gives the results of our experiments with grammar *DoTillImprove*. One sees that search performance, in terms of the mean-best value, indeed clearly improves, even when only a small ι, i.e., number of loop iterations, is used. For comparison, the bottom row of the table gives the best known result for `eil51`, taken from (Jayalakshmi, Sathiamoorthy, and Rajaram 2001).[2]

We have provided the GP hyperheuristic with very trivial, low-level heuristics only, and with a simple grammar. It is therefore most remarkable that, for $\iota = 70$, one run of the hyperheuristic evolves a metaheuristic that locates a tour whose rounded length equals the best known result. We call such a method a *best metaheuristic*. Unfortunately, (Jayalakshmi, Sathiamoorthy, and Rajaram 2001) does not mention the precision of its result. We report that our full-precision value equals 428.871765.

Actually, for $\iota = 800$, the hyperheuristic routinely produces best metaheuristics, i.e., each of its 100 runs produces at least one such MH, as evidenced by standard deviation *zero*. Also, with much less effort, given $\iota = 480$, *almost* every run produces at least one best MH.

For $\iota = 800$, a run lasts 10.1 minutes on average on common hardware; here: a single core of an Intel Xeon 3.2 GHz dual core machine without additional heavy processes, but with I/O for storing about 1 GB of data about states of the hyperheuristic. Even so, as 100,100 (initial individuals plus offspring) /10.1/60 $\approx$ 165, this many metaheuristics are produced each second on average, including their execution that locates tours as solutions to the given problem.

[2](Jayalakshmi, Sathiamoorthy, and Rajaram 2001) presents a hybrid Genetic Algorithm, only applicable to Euclidean TSPs, that uses a specialized crossover together with several non-trivial, handcrafted heuristics.

eil51	**Mean best**	S.D.	Best	ι	P.%
Nat. length	1,313.47	n.a.	n.a.	n.a.	206.26
DTI	528.89	8.98	508.75	10	23.32
”	477.82	8.11	454.12	20	11.41
”	458.61	5.79	444.21	30	6.93
”	439.42	2.90	431.95	60	2.46
”	436.61	3.07	**428.87**	70	1.8
”	429.28	0.56	*428.87*	240	0.1
”	428.88	0.03	*428.87*	480	0.002
”	**428.87**	**0.00**	*428.87*	**800**	**0.0**
”	*428.87*	*0.00*	*428.87*	$\mathbf{10^4}$	*0.0*
Hybrid GA	n.a.	n.a.	***428.87***	*best known*	n.a.

Table 8.10: Performance for problem eil51 over grammar *DoTillImprove* (*DTI*) for different values of ι, ι iterations at most, 100 runs per value. **P**: Mean best or natural length in terms of % of best known result. All real values rounded off to nearest hundredth except where higher precision required for distinction.

ϕ.	S.D.	*Primitive*
190.37	11.9	`n` “no operation”
0.999	0.00008	`NATURAL`
90.77	2.87664	`REPEAT_UNTIL_IMPR.`
104.27	7.5	`IF_3-CHANGE`
99.61	7.89451	`IF_2-CHANGE`
14.98	3.6872	`2-CHANGE`
309.63	$\mathsf{l}.-\phi(\mathtt{n})$	*mean effective MH size*

Table 8.11: For problem eil51, grammar *DoTillImprove*, $\iota = 800$, ι iterations at most, 100 runs: average primitive frequency ϕ. Fixed max. genotype size $\mathsf{l} = 500$.

Next, we are interested in the structure of an evolved metaheuristic. For a primitive p and a string s, we define the *primitive-frequency* $\phi(p)$ as the number of p's occurrences in s. We focus on the average primitive-frequencies over all runs with $\iota = 800$, i.e., we ask how many instances of a given primitive occur, on average, in a produced genotype. Table 8.11 summarizes the results. While, as seen before, each run yields at least one best metaheuristic, the evolved metaheuristics do not, on average, exhaust the maximal genotype size, l, as evidenced by $\mathsf{l} - \phi(\mathtt{n})$ (bottom row of table). This means that the hyperheuristic works against bloat, a typical nuisance with many GP flavors, i.e., strong growth of undesirable parts of an evolved structure. Hence, the mean effective size of a produced metaheuristic is less than 62% of l, the fixed maximal size of a MH.

$\phi(\mathtt{NATURAL})$ does not exactly equal 1 due to rare, evolved strings that only consist of `n`-instances and that never got selected for competition with a metaheuristic. $\mathtt{IF\_3-CHANGE}$ has the highest ϕ value among the non-`n` primitives.

eil76	**Mean best**	S.D.	Best	ι	**P**.%
Nat. length	1,974.71	n.a.	n.a.	n.a.	262.75
DTI	600.09	12.37	*576.60*	*800*	10.24
”	592.93	12.10	564.48	2×10^3	8.92
”	589.63	11.98	562.27	5×10^3	8.31
”	586.29	12.81	559.78	1.5×10^4	7.7
”	586.39	12.55	561.79	5×10^4	7.72
”	586.10	12.47	559.09	5×10^5	7.67
”	586.48	12.81	557.90	5×10^6	7.74
Hybrid GA	n.a.	n.a.	***544.37***	*best known*	n.a.

Table 8.12: Performance for eil76, details as given in caption of Table 8.10.

This is of interest because human MH designers prefer the use of 3-CHANGE over 2-CHANGE: while the former runs longer, the latter often does not shorten a tour as much. $\phi(\texttt{IF\_2}-\texttt{CHANGE})$ is the second-highest frequency, so that IF_3-CHANGE and IF_2-CHANGE, the two non-destructive, tour-changing primitives, are the most frequent effective elements of an evolved MH.

ϕ(REPEAT_UNTIL_IMPROVEMENT), the third-highest frequency, still equals about 29% of the mean effective MH size. This shows that the hyperheuristic makes strong use of this loop-based primitive, approximating a k-opt heuristic that is widely used in hand-crafted TSP solvers. This heuristic executes a k-change until no further improvement occurs, while REPEAT_UNTIL_IMPROVEMENT, together with its primitive to be repeated, runs until a first improvement happens or ι is exhausted.

Eventually, $\phi(2-\texttt{CHANGE})$ equals about 4.8% only. This shows that the hyperheuristic mostly does not use this tour-changing primitive that represents the only destructive operator among all given heuristics.

Next, we consider eil76 (Reinelt 2007), a 76-node problem with a size of about 1.9×10^{111} search points. We use the same basic parameters as given in Table 8.6, and, again, grammar *DoTillImprove*.

Results are summarized in Table 8.12. Despite the vastly larger search space, already $\iota = 800$, the value that results in zero standard deviation over eil51, yields a promising result as given in column “Best”, i.e., $\beta = 576.7$. 1,974.71 is the length of eil76's natural cycle that an evolved metaheuristic starts changing into its output cycle. Relative to this natural length, β is already in the vicinity of the best known result from literature (Jayalakshmi, Sathiamoorthy, and Rajaram 2001), i.e., $\alpha = 544.37$.

We therefore keep increasing ι over some orders of magnitude, up to 5×10^6, which results in a new $\beta = 557.9$ that is within $\beta/\alpha - 1 \approx 2.49\%$ of α.

Over ι's magnitude orders 4, 5, and 6, the stagnation of P.%, of the corresponding mean best values, and of their standard deviations suggests increasing the number of offspring to be evolved. We therefore set this basic parameter to 1×10^6, keeping all other parameter values. Judging by table 8.12, already $\iota = 2\times 10^3$ is a promising

Popul. size	Genotype size	**Offspring**	Mut. prob.
100	500	1×10^6	0.5

Table 8.13: Basic Parameters

eil76	Mean best	S.D.	Best	ι	P.%
Nat. length	1,974.71	n.a.	n.a.	n.a.	262.75
DTI	548.99	1.67	**544.37**	2×10^3	0.85
Hybrid GA	n.a.	n.a.	**544.37**	**best known**	n.a.

Table 8.14: Performance of metaheuristics evolved over language DTI, on problem eil76. 100 runs of GP hyperheuristic for given ι value. Best evolved metaheuristics at least match effectiveness of hand-crafted Hybrid GA. Other details as given in caption of Table 8.10.

value. Table 8.13 summarizes the values of the basic parameters. Table 8.14 shows results. The mean best over all runs is well within one percent of $\alpha = 544.37$, the best known solution. Our best evolved metaheuristics yield tour lengths that are actually shorter than α. Unfortunately again, (Jayalakshmi, Sathiamoorthy, and Rajaram 2001) does not specify whether α is a rounded value. As tour lengths can be close to each other – the hyperheuristic finds, for instance, tour lengths 547.*737*183 and 547.*740*601 – full-precision values can be critical to comparing effectiveness of TSP solvers. Thus, we report that our GP hyperheuristic has found an overall best tour length of $\alpha_{HH} = \mathbf{544.36908}$.

This result and also the best GP-HH result over problem eil51 easily beat best results from (Oltean 2005) that compares its evolved evolutionary algorithms (EA) against a standard GA, while our evolved metaheuristics must compete with the sophisticated, specialized GA from (Jayalakshmi, Sathiamoorthy, and Rajaram 2001) that has produced α. In particular, the evolved EAs start with initial tours gained from the Nearest-Neighbor heuristic, while our evolved metaheuristics begin with the natural tour as given by the problem statement.

To our knowledge, our best results on both benchmark problems are the best over all GP work done on TSP.

For $\iota = 2,000$, on average, the hyperheuristic produces 97 metaheuristics per second on eil76. On eil51, the previous problem, we obtained a value of 165. Thus, an increase in problem size by factor $76/51 \approx 1.5$ has increased effort by factor $165/97 \approx 1.7$. This is a moderate price to pay, considering that the search space of eil76 is larger by about factor 10^{111-66}.

For eil76, we are interested in whether or not, at a reasonable expense of more run time, the hyperheuristic can find more very good results. Thus, we run the GP-HH again, changing ι from 2,000 to 15,000. Table 8.15 shows results. The mean best's P.% value drops down to 0.26%, less than a third of its previous value. On average, the GP hyperheuristic produces 56 metaheuristics per second, which is still about 60% of the previous rate. While we consider only 30 runs this time, the very

eil76	Mean best	S.D.	Best	ι	P.%
Nat. length	1,974.71	n.a.	n.a.	n.a.	262.75
DTI	545.77	0.97	**544.37**	1.5×10^4	0.26
Hybrid GA	n.a.	n.a.	**544.37**	**best known**	n.a.

Table 8.15: Performance over 30 runs of GP hyperheuristic for given ι value. Other details as given in caption of Table 8.14.

low standard deviation indicates reliable search behavior. In view of these values, the more than sevenfold increase of ι has paid off.

8.6 Summary and conclusions

In this chapter, we have introduced a domain-independent, linear GP hyperheuristic that produces metaheuristics from provided heuristics by use of deleting repair. The metaheuristics are expressions from a user-given language.

We demonstrated this approach for the domain of traveling-salesperson problems. To this end we provided the hyperheuristic with simple heuristics from this domain and with a progression of simple grammars. The grammar over which the GP-HH has been seen to do best describes sequences of heuristics and loops, with a fixed maximum number of loop iterations of a single heuristic.

While this grammar is simple, for the considered, realistic benchmark TSPs, the GP hyperheuristic evolves metaheuristics that find tours with lengths that are highly competitive with the best known lengths from literature. These lengths are often yielded by specialized, man-made solvers that use sophisticated, handcrafted heuristics and that are initialized with tours that are provided by some smart, manually produced initialization heuristic. None of this is available to the presented GP hyperheuristic.

In addition, by setting the iteration number and the number of produced metaheuristics appropriately, the hyperheuristic has been able to routinely produce metaheuristics capable of matching best known results, with modest run times.

The approach produces parsimonious metaheuristics whose sizes are, on average, considerably smaller than the maximally available size. Out of the provided primitives, the GP hyperheuristic strongly favors the non-destructive primitives and the loop-based primitive. These proved critical to giving competitive effectiveness to the produced metaheuristics.

However, the GP hyperheuristic did not ignore the destructive primitive but discovered a good ratio of all primitives. In particular, it did not wander off into producing very greedy metaheuristics only, as, for instance, some greedy hyperheuristic would do.

As the principles of the GP hyperheuristic are domain-independent, one may hope it will work for other domains when one supplies it with domain-specific heuristics and languages. The user of this approach can influence the search performance of the hyperheuristic at an abstract, problem-oriented level by modifying the used

language, i.e., by adding or removing heuristics, and by describing optional patterns for the order of execution of the given heuristics.

To that end, for an arbitrary domain, one may manually develop sophisticated heuristics or, saving expensive manpower, merely provide trivial heuristics, or simply extract smart methods from existing solvers, and represent all of them as primitives. These are the terminals of a grammar that generates the language used by the hyperheuristic. The latter will then perform the tedious task of searching for a promising metaheuristic, i.e., an effective combination of the provided heuristics.

Also, one may allow a generous maximal size of the metaheuristics that are to be evolved, since the presented GP hyperheuristic displays an implicit tendency toward keeping them parsimonious. In particular, for different metaheuristics, deleting repair, embodied as EDITING operator, often leads to a different effective size, i.e., the number of effective primitives in a metaheuristic. This variance is beneficial as it is a necessary condition for the emergence of metaheuristics that are both parsimonious and good.

EDITING requires an only small computational effort, since the elimination of an erroneous primitive i_j from a metaheuristic does not force i) the calculation of the set of primitives that are proper in locus j, and ii) subsequent selection of one of them with which to replace i_j.

Thus, only given point mutation, EDITING, and a fixed maximal size of its products, the hyperheuristic may benefit from parsimonious metaheuristics and their varying sizes, without risking bloat.

In summary, we have demonstrated the efficacy of deleting repair, in terms of avoiding bloat and supporting diversity of phenotype size, on a realistic problem domain. In feasible time, such problems yield to metaheuristics only, if at all. Therefore, and as deleting repair relies on an underlying grammar, we have taken this situation as opportunity to apply Genetic Programming as a novel hyperheuristic that is guided by a grammar that describes problem-specific metaheuristics, using a loop-based primitive, i.e., grammar terminal.

As an aside, we suggest some directions of future work on this GP hyperheuristic. One may, for instance, test it on further real-world problems coming from several domains, comparing the produced metaheuristics to other domain-specific algorithms. Additionally, one may identify a problem type for which few or no well-established solvers exist, such as fully autonomous real-time control of agents in dynamic, hazardous environments. One may also break up low-level heuristics into their components and represent them as primitives. In this way, in principle, the hyperheuristic would be able to produce even more novel and powerful metaheuristics. Furthermore, the hyperheuristic could, on the fly, represent an evolved metaheuristic as a further primitive of the underlying grammar, thus enriching the language used for expressing metaheuristics.

Next, we give a summary of the present work.

Chapter 9

Summary

Real-world problems meet with insufficient flexibility of current artificial production systems, which yields our long-term strategic objective: realizing a matrix from which material autopoietic systems emerge that represent acceptable solutions. For computer science, this implies autopoietic programming: a computing system that emerges and self-stabilizes in a problem environment.

A system's capability of performing its ontogeny—development as sequence of endogenous structural modifications—is necessary for its autopoiesis. Ontogeny is therefore essential to the evolution of complex systems, i.e., their implicit adaptation. *Natural* evolution is an instance that is autopoietic and produces similar systems, and it is a paragon for the well-established Evolutionary Algorithms (EA): these probabilistic direct search methods use heuristics that are distorted but still effective metaphors of biological phenomena. Thus, Genetic Programming (GP), given by EAs that find problem solutions represented as programs, lends itself to approximating autopoietic programming.

Ontogeny occurs under influence of information, resulting in a structure that, in its environment, exhibits behavior that effects adaptive pressure on the information. In nature, a genotype co-determines cellular structures that are the material basis of an associated phenotype. Previous work has extended GP by a fixed genotype-phenotype mapping. However, a static EA component is of limited practical value. In particular, self-adaptation, i.e., endogenous self-maintaining modification, is necessary for autopoiesis. Thus, we see a self-adaptive genotype-phenotype mapping as a near objective. However, an engineering approach toward this goal raises the conflict of recursion: one cannot manually produce a fully self-adaptive algorithm, which would require an infinite hierarchy of control layers. The explicit and therefore fixed semantics of an algorithm is the dilemma's root, so that co-evolution of *i) semantics-giving* and ii) -carrying components of a GP algorithm may support its implicit adaptation, weakening the conflict. A technical objective results: investigating whether introducing evolution-related phenomena—in particular, ontogeny and co-evolution—to a GP algorithm may support its self-adaptation and performance. To that end, cybernetics suggests the co-evolution of both an *i) ontogenetic apparatus* and ii) genotypes as a focus, because (i) performs a mapping from (ii) onto phenotypes whose behavior might feed back selective pressure to (i), proposing a circle of mutual control in place of a hierarchy of command layers.

A theoretical discussion analyzes and extends the concepts of the GP-oriented mapping to a model that can be used by a generic search algorithm. A common search algorithm identifies genotypes with phenotypes, opposed to a developmental search method. Specialized for the present work, the model sees a GP algorithm producing phenotypes from a solution space that is a subset of a given target language, while the genotypes come from a decision space.

An initial mapping step reads genotypes and writes primary transcripts by use of a manually given genetic code. A full developmental GP algorithm maps the entire decision space onto solutions that are final transcripts, synthesized by alternative repairing algorithms that we suggest.

Of these methods, the generic model recommends deleting repairing which best supports autonomy of the search algorithm. This repairing type enables the algorithm to accept also those genotypic introns that represent phenotypic introns that are syntax errors. As a side effect, this tolerance allows the algorithm to maintain maximal genetic diversity. Also, the necessary deleting of all—differently sized—erroneous phenotypic introns implicitly yields desirable structural diversity of the resulting solutions.

To emphasize the efficacy of deleting repairing, the present work describes a further, domain-independent, linear GP method that uses this repairing type. The method produces metaheuristics, only using elementary, provided heuristics. The metaheuristics are expressions from a user-given target language.

This approach is shown to perform well for a domain of discrete combinatorial problems. The given language is simple. Within modest run time, however, and for realistic benchmark problems, the GP method evolves metaheuristics that find solutions that are highly competitive with the best known solutions from literature. The latter are often yielded by specialized, man-made solvers that use sophisticated, hand-crafted heuristics.

The approach produces parsimonious metaheuristics whose sizes are, on average, considerably smaller than the maximally available size. Thus, one may allow a generous maximal size which gives enough space for the synthesis of valuable subexpressions, divided by erroneous strings that deleting repair will eliminate. In this manner, for different metaheuristics, deleting repair often leads to a different effective size, i.e., the number of effective components of a metaheuristic. This variance is beneficial as it is a necessary condition for the emergence of metaheuristics that are both parsimonious and good.

Deleting repair requires an only small computational effort, since the elimination of an erroneous component i_j from a metaheuristic does not force i) the calculation of the set of components that are proper in locus j, and ii) subsequent selection of one of them with which to replace i_j.

Thus, only given point mutation, deleting repair, and a fixed maximal size of an evolved product, a search method may benefit from parsimonious products and their varying sizes, without risking bloat.

The generic model also implies that, without information on the problem-relevance of a target symbol, the mapping of genotypes must be surjective. To

that end, as undesirable constraint, the employed genetic code must contain all target symbols. Thus, the model suggests redundant mappings, which requires a code or repairing to raise neutrality of genotypes. We focus on the according *adaptation of a code*. As for representation, genotypes should be high-dimensional over phenotypes. This may establish neutral genotypic networks as diversity pools between which developmental search may outrun common search through tunnels.

For empirical investigation, we develop a linear, compiling, developmental as well as common GP system with genotypic fixed-size binary strings and phenotypic sentences of an arbitrary LALR(1) language (here: ISO-C). Technically, from a final transcript, a proposed editing algorithm produces input to compiling or interpreting phases that yield phenotypic behavior. Cybernetically, only the final transcript matters as it represents genotypic semantics that results in feedback of adaptive pressure to the genotype.

For system design, we follow minimalism and efficiency as principles. Notably, they, as well as the model, argue for algorithms that, upon their completion, are recognized as metaphors of biological phenomena, e.g., deleting repairing that mimics so-called intron splicing. This situation corroborates the well-known heuristic EA-design principle of selecting, without deep analysis, a biological phenomenon as paragon for an algorithmic component, because one assumes that cybernetic quality of the phenomenon has emerged from natural evolution.

As targets of empirical research, we implement static symbolic regression problems. Results confirm model predictions. In particular, a beneficially redundant code and high-dimensional genotypic representation foster performance over common search. However, for practical problems, the user often lacks complete knowledge on symbol relevance, which makes a manual *ad hoc* design of a desirable genetic code infeasible.

Therefore, with codes and redundant mappings in the center of interest, the focus moves to co-evolving individual codes and genotypes. Theory supports this objective since code variability fosters variability of fitness-landscape topologies whose individual properties—notably, locally strong causality—are critical for search performance. The EA–design principle also recommends this approach because natural evolution has produced the Universal Genetic Code that favors the evolution of organisms.

In our accordingly extended system, an initial individual consists of its genotype and a random genetic code. Explicit mutation of codes and their implicit cloning and selecting via their carriers are to drive their evolution. The suggested code hypothesis defines this evolution as a process producing individual codes that are increasingly beneficially redundant. The accompanying code-evolution explanation (CEE) views this process as an instance of cooperative co-evolution: autocatalytic mutual hitch-hiking of codes and their carriers.

Code evolution is observed on a small problem with tractable code-fitness measuring. Simultaneous positive progress of both individual and code fitness corroborates the CEE. Since adapted codes are components of the underlying developmental search algorithm $d_{x\rightarrow}$, code evolution appears as $d_{x\rightarrow}$'s implicit self-adaptation,

which approaches the technical objective. This effect results from codes and genotypes co-operating as well as from instances of target symbols competing for limited space of a code. Code evolution is also observed on a large and high-dimensional problem that is hard for $d_{x\rightarrow}$ regarding both code and genotypic search. Occurring positive fitness progress predicts this evolution, thus corroborating the CEE.

From a machine-learning perspective, the observed process performs a good, unsupervised, probabilistic binary classification of target symbols by their problem relevance, thus filtering noise from the problem representation.

Code evolution implicitly adapts individual genotype-phenotype mappings, reducing the need for their difficult or infeasible manual design. In particular, it approaches the GP-typical task of providing a minimal, sufficient function-and-terminal set: an optimal code is a surjection onto this set. Thus, code evolution shrinks the solution space and increases density of acceptable phenotypes there, supporting performance. It also weakens the well-known curse of dimensionality, since it fosters mapping genotypes onto good phenotypes, thus increasing the number of favorable directions for the search process to choose from for the next exploratory step away from a located individual. Eventually, an individual genetic code under a small mutation rate represents a long-term process memory shielded from direct selection pressure by the phenotype, adding process inertia for stability.

In summary, theoretical and empirical observations indicate that co-evolving genotypes with their developmental machinery as part of a GP algorithm can support its autonomy and performance.

Chapter 10

Conclusions and outlook

> Architecture is frozen music. — Goethe
>
> The brain is an orchestra without a conductor,
> playing a mysterious, wonderful symphony. — Unknown

A phenotype is i) a structure whose interpretation yields behavior of interest, or ii) a model of such behavior, or iii) such behavior. A computer-program representation is but an instance of (i). Thus, while our focus is on GP, the summarized arguments on development and search dynamics apply to processes adapting arbitrary phenotypical structures in an evolvable medium, e.g., a target language whose sentences manipulate material items and represent a dynamic physical structure. In particular, literature's ubiquitous distinction between different EA flavors is secondary in the light of ontogeny, since their prominent structures, e.g., real-valued vectors, finite automata, or program representations, can result as problem-specific expressions of universal genotypes, i.e., binary strings that carry a search process that is independent from phenotypic technicalities.

- As for exploiting our presented results to the end of autonomy and performance of a search process, we recommend such approaches to real-world problems that feature self-adaptive development. Individual genotype-phenotype mappings and their co-evolution with genetic information, as has been introduced here for GP, can weaken the dilemma that a process user tries compensating for incomplete problem knowledge by adding search mechanisms that subsequently, however, require manual control.

- In particular, for GP-like processes, some possibilities come to mind: Redundancy adaptation of *limited-size* codes, eliminating or introducing user-given target symbols, results from an implicit pressure toward a sublanguage of minimal sufficient expressiveness, implementing Ockham's Razor. This addresses prominent conflicts of GP-usage, for instance: a given Turing-complete target alphabet may unnecessarily have an infinite loop emerge, if the underlying

problem does not require completeness.[1] Then, also a code without irrelevant, iteration-raising symbols may result in acceptable solutions, and it can be more redundant on relevant symbols. In general, adaptation of codes may gradually shift the active part of the given target language to a subset that properly models the problem at hand.

- Code evolution implicitly yields *automatically defined functions* if a code entry may contain a non-trivial symbol *sequence*. Code variation would turn the sequence into a building block, i.e., a problem-relevant macro-symbol, increasing genotypic functional abstraction. As side effect, to genotypic recombination, this phenotypic building block appears encapsulated and thus protected against well-known, undesirable *disruptive crossover*, provided a crossover point resides between codons only.

Also, DGP is independent from the class and level of a target language. One may therefore supply an arbitrary decidable language, if one expects problem-specific advantages. For instance, short genotypes and a context-*sensitive* language may lead to elegant parsimonious and acceptable solutions, while repair time remains acceptable although the word problem is not efficiently decidable. For another instance, the native code of a given processor type, supplied as target language, would have DGP emulate *machine-language GP*.

Regarding further empirical work, measuring causality progression during code evolution would test the implicit CEE prediction that code adaptation smooths the individual fitness landscapes. As an issue of particular theoretical interest, one may investigate whether prominent algebraic structures exist in evolved desirable codes. Eventually, further work could extend the mathematical model for predicting search dynamics.

As for exploring beyond the realm of presented results, the principles behind co-adapting codes apply to other components of a DGP algorithm as well. Further parameters can be softened and become part of the evolving medium, such as a codon that might change in length, directly changing size and dimensionality of the decision space. However, no EA design can fully annihilate the conflict of recursion in this way because the subalgorithms that modify the parameters require additional explicit control, thus maintaining the dilemma. Also co-evolving such subalgorithms offers no escape either, since an EA that modifies its algorithms may crash or cease to represent an Evolutionary Algorithm, and user-given protective measures that address these risks introduce further explicit components.

Thus, in the short term, coming EAs will be made to mimic ever more complex biological phenomena, such as regulatory networks, in order to increase their autopoietic potential. For practicality, such design regression, stacking control layers

[1] A GP system may bluntly terminate an evolved, possibly infinite loop after a user-chosen number of iterations, or it may use man-designed, sophisticated flexible mechanisms to address this issue, like a metaphor of limited energy, as one of my former master students suggested (Nabuurs 2004). As best option, however, a system only provides a loop statement if the latter is required by a problem.

of adaptation, is ended by an autopoietic system—namely, the user's brain—that supplies problem-specific values to the outer layer, thus closing the algorithm cybernetically.

In the long term, one should set up an environment that brings forth a fully autopoietic computing system. Inspired by the presented concept of self-adaptive ontogeny, the system may be represented as a binary string that *is* the computing medium. With a given, minimal Turing-complete bit-based language that implies laws that govern the medium, the latter permanently maps itself onto itself, neither reaching chaos nor a fixed point that would mean death, i.e., static equilibrium. Problem data, feeding into the medium, creates a structured disturbance, giving rise to selection, while external noise gives variability, and stability follows from the physical implementation, e.g., RAM. Thus, with all necessary conditions for evolution given, problem-relevant patterns may emerge. At a freely chosen region of the medium, the environment accepts patterns that occur there, and it interprets them, i.e., it applies them to the given problem. The resulting, more or less desirable reaction must feed back to the medium as less or more external noise, respectively, stabilizing or disturbing the emerging patterns. In this manner, without any manual setting up, the process carried by the medium learns about the problem, its representation, and the solution representation expected by the environment.

One or more virtual CPUs raise the process by interpreting and thus animating the medium that therefore is a permanently self-rewriting sentence of the given language.[2] This allows for higher interpreters to emerge, i.e., for abstraction build-up. Systemic singularities, that is, conditions where the underlying real machine crashes, do not exist by design of the virtual machine.

In conclusion, the behavior of this hypothetical process is only biased by problem data and therefore autopoietic. Both origin and adaptation of order in the carrying medium do not require an external source, and, as side effect of the continuous adaptation, a problem solution is approached. However, system transparency, as enforced by explicit programming of a computing medium, is traded in for such flexibility. At any given point in time, the semantics of the carried behavior is hard to decode, since each single state of the hypothetical medium merely represents a snapshot of an implicit semantics definition that is distributed over space, i.e., memory, and time. Semantic analysis would meet obstacles well known from research on organic neural networks and their plasticity.

- A *material* analogy of the sketched digital medium would reach the strategic objective.
- An empirical proof of principle requires implementing the medium as next step. To that end, work by the author is in progress.

[2] *Cum grano salis*, because, actually, over time, the dynamic medium represents a set of sentences, and each exists for just one point in this time, because the next state change, i.e., bit flip, creates another sentence. Thus, each sentence is a complete architecture as well as but a freeze frame, a slice, taken out of the high-dimensional sculpture the underlying developmental process chisels out of the phase space of the medium.

Appendix A

Mathematical conventions

The notation $\boldsymbol{A \subset B}$ shall denote that A is a proper or improper subset of B. A mapping of sets A and B shall be a binary relation of the sets, i.e., a subset of $A \times B$, their Cartesian product. Especially, a function f from A into B, denoted by $\boldsymbol{f : A \rightarrow B}$, is a mapping such that each $a \in A$ is paired with exactly one $b \in B$, denoted by $\boldsymbol{f(a) = b}$, also expressed as mapping a onto b. B is called the range set of f [$\boldsymbol{rng(f)}$], and A is called the domain set of f [$\boldsymbol{dom(f)}$].

$$\{b \in rng(f) \mid \exists\, a \in dom(f) : f(a) = b\}$$

is called the image set of f, denoted by $\boldsymbol{img(f)}$ or $\boldsymbol{f(dom(f))}$, and ergo $img(f) \subset rng(f)$.

$f : A \rightarrow B$ with $f(a) = a$ is called the identity function of A, denoted by $\boldsymbol{id_A}$, and $dom(id_A) = img(id_A) \subset rng(id_A)$ follows.

Let $f : A \rightarrow B$, let $C \subset A$.

$\{(a, b) \in f \mid a \in C\}$ is called the restriction of f to C, or f restricted to C [$\boldsymbol{f|_C}$]. Thus, a restriction of a function is a function, and $dom(f|_C) = C$ follows.
$\boldsymbol{rng(f|_C)}$ shall equal $rng(f)$. (A.1)

$img(f|_C) = \{b \in B \mid \exists c \in C : f|_C(c) = b\}$ follows,
implying $b \in img(f|_C) \Rightarrow b \in img(f)$, since $f(c) = f|_C(c) = b$.
This implies $img(f|_C) \subset img(f)$. (A.2)

$img(f|_C) = \{x \in rng(f|_C) \mid \exists a \in dom(f|_C) : f|_C(a) = x\}$ holds.
This implies $img(f|_C) = f(C)$, because $rng(f|_C) = rng(f)$ and $f(a) = f|_C(a)$. (A.3)

Let $f : A \rightarrow B$ and $A_1, A_2 \subset A$, then $f(A_1 \cap A_2) \subset f(A_1) \cap f(A_2)$ holds. (A.4)

A function f is called an injection (adj. injective) if, for each $b \in rng(f)$, there is, at most, one $a \in dom(f) : f(a) = b$. f is a surjection (surjective) if $img(f) = rng(f)$, and, eventually, it is a bijection (bijective) if it is injective and surjective.

Let $n \geq 0$, $n \in \mathbb{N}$, $a_0, .., a_{n-1} \in A$, then $t = (a_0, .., a_{n-1})$ shall be called an n-tuple over A. An a_i, $0 \leq i \leq n-1$, is called a component of t [$\boldsymbol{a_i \in t}$], and n is the size of t [$|\boldsymbol{t}|$].[1]

[1] Usually, the smallest component index equals one. Here, we acknowledge computer science where index counting starts at zero. Also, mathematically, the $\in$-notation applies to sets only. Both differences, however, do not violate the connotations of the common definitions.

About the Author

Robert Keller has been involved with informatics in research, R&D, teaching, and industry. From the University of Dortmund, Germany, he holds a diploma in computer science with emphasis on mathematics and software engineering. His interest is in self-organizing systems and their use as, for instance, optimizers, controllers, and autonomous agents. At the Computer Science Department in Dortmund, and at the Informatics Center Dortmund, he has contributed to the Collaborative Research Center "Design and management of complex technical processes and systems with Computational Intelligence" (German Research Foundation) as well as to the European Network of Excellence in Evolutionary Computing. He has also done research and taught at the Faculty of Mathematics and Natural Sciences at the University of Leiden, the Netherlands. Since 2005, he has been with the Department of Computing and Electronic Systems at the University of Essex, U.K. In the past, he taught systems analysis, evolutionary computing, artificial life, object-oriented software engineering, and programming. He reviews for journals and conferences on computational intelligence.

Selected publications

Robert E. Keller, Riccardo Poli, "**Improved Benchmark Results from Subheuristic Search**", Parallel Problem Solving from Nature (PPSN) 2008, Dortmund, 2008 Sep; Workshop on Hyperheuristics (long paper)

Robert E. Keller, Riccardo Poli, "**Toward Subheuristic Search**", Proceedings of the 2008 Congress on Evolutionary Computation (IEEE CEC) within 2008 World Congress on Computational Intelligence (IEEE WCCI), Hong Kong, 2008 Jun; IEEE press, 2008 (long paper)

Robert E. Keller, Riccardo Poli, "**Self-adaptive Hyperheuristic and Greedy Search**", Proceedings of the 2008 Congress on Evolutionary Computation (IEEE CEC) within 2008 World Congress on Computational Intelligence (IEEE WCCI), Hong Kong, 2008 Jun; IEEE press, 2008 (long paper)

Robert E. Keller, Riccardo Poli, "**Cost-benefit investigation of a Genetic-Programming Hyperheuristic**", Proceedings of the 8th International Conference on Artificial Evolution (EA) 2007, Tours, France, 2007 Oct (long paper)

Robert E. Keller, Riccardo Poli, "**Linear Genetic Programming of Parsimonious Metaheuristics**", Proceedings of the 2007 Congress on Evolutionary Computation (IEEE CEC), Singapore, 2007 Sep (long paper)

Robert E. Keller, Walter A. Kosters, Martijn van der Vaart, Martijn D.J. Witsenburg, "**Genetic Programming Produces Strategies for Agents in a Dynamic Environment**", Proceedings of the Belgium/Netherlands Artificial Intelligence Conference (BNAIC) 2002, The Netherlands, 2002 Oct (long paper)

Robert E. Keller, "**Evolutionary methods for data mining**", in: "DEALING WITH THE DATA FLOOD: Mining data, text and multimedia", Meij, J.M. (ed.), STT Netherlands Study Centre for Technology Trends, The Hague, The Netherlands, 2002 Apr (book chapter)

F. Lohnert, A. Schütte, J. Sprave, I. Rechenberg, I. Boblan, U. Raab, I. Santibéñez Koref, W. Banzhaf, R.E. Keller, J. Niehaus, H. Rauhe, "**Genetisches Programmieren für Modellierung und Regelung dynamischer Systeme**", Daimler-Chrysler AG Forschung und Technologie FT3/AI; Technische Universität Berlin, Bionik und Evolutionstechnik; Leiden Institute of Advanced Computer Science; Informatik Centrum Dortmund e.V., 2001 (final project report)

Robert E. Keller, "**Autopoietic solutions for real-world problems**", in: EvoDebates: Fit for the Future; EvoNet—The European Network of Excellence in Evolutionary Computing, 2001 (invited position statement)

Robert E. Keller, Wolfgang Banzhaf, "**The evolution of genetic code on a hard problem**", in: Lee Spector, Erik D. Goodman, Annie Wu, W.B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, Edmund Burke (eds.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), San Francisco; Morgan Kaufmann Publishers, San Francisco CA, 2001 Jul (long paper)

Robert E. Keller, Wolfgang Banzhaf, "**The evolution of genetic code in genetic programming**", in: W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, R. E. Smith (eds.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999), Orlando FL; Morgan Kaufmann Publishers, San Francisco CA, 1999 Jul (long paper)

Robert E. Keller, Wolfgang Banzhaf, Jörn Mehnen, Klaus Weinert, "**CAD Surface Reconstruction from Digitized 3D Point Data with a Genetic Programming/Evolution Strategy hybrid**", in: Advances in Genetic Programming 3, Chapter 3, MIT Press, 1999 (book chapter)

Stefan Klahold, Steffen Frank, Robert E. Keller, Wolfgang Banzhaf, "**Exploring the Possibilities and Restrictions of Genetic Programming in Java Bytecode**", Late Breaking Papers at the Genetic Programming 1998 Conference, John R. Koza (ed.), University of Wisconsin, Madison; Stanford University Bookstore, 1998

Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, Frank D. Francone, "**Genetic Programming—An Introduction**; On the Automatic Evolution of Computer Programs and its Applications", Morgan Kaufmann Publishers, San Francisco CA; dpunkt.verlag, Heidelberg, Germany, 1998 (book)

Robert E. Keller, Wolfgang Banzhaf, "**Genetic Programming using Genotype-Phenotype Mapping from Linear Genomes into Linear Phenotypes**", Genetic Programming 1996: Proceedings of the First Annual Conference; Stanford University; John R. Koza, David E. Goldberg, David B. Fogel, Rick L. Riolo (eds.), MIT Press, Cambridge MA, 1996 (long paper)

Acknowledgments

Foremost, my gratitude belongs to *Wolfgang Banzhaf*, head of the Computer Science Department at the Memorial University of Newfoundland, St. John's, Canada, for encouragement and opportunities that I have enjoyed in his Emergent Computation research group[2], and that I hopefully could and will pass on to my students.

Special acknowledgments go to *Hans-Paul Schwefel*, former head of the Systems Analysis group at the former Department of Computer Science at the University of Dortmund, for friendly support and discussions on artificial evolution and its great natural paragon.

Many thanks are due to *Ulrich "Batchman" Hermes* for technical support and countless heated XPilots[3] battles during nocturnal coffee breaks.

Special thanks belong to *Elena* for challenging my over-simplifications of molecular biology and for keeping several Linux flavors happy that live on my laptops.

Last, by far not least, I owe a bowl of his favorite gourmet food to *Timothy*, the 20-pound cat, for his mostly patient and quiet company. His autopoietic behavior[4] keeps reminding me of the embarrassing gap between technical and natural systems.

[2]Computer Science Department at the University of Dortmund

[3]A real-time, 2-D, multiplayer conflict environment

[4]in particular, when he most selectively stole the ham from my sandwich

Zusammenfassung

Kapitel 1 nennt Zielsetzungen der vorliegenden Arbeit, die sich für künstliche Systeme interessiert, die praxisrelevante Problemumgebungen angreifen. Dort ist der Selbsterhalt, i.e., die Aufrechterhaltung des problemspezifischen Systemverhaltens, von überragender Bedeutung, und umgebungsabhängig werden dazu möglicherweise Änderungen der Systemstruktur notwendig. Autopoiese bezeichnet die Selbsterzeugung und -bewahrung einer Entität. Emergenzfähigkeit und Selbstorganisation gegenwärtiger künstlicher Systeme und die sich ergebende Systemleistung erscheinen unzureichend in praktischen Umgebungen. Dort sehen wir als ein Idealsystem eine Entität, die selbstbestimmt Problemstellungen erkennen und behandeln würde und womöglich ähnlich unabhängige Subsysteme als Lösungen erzeugen würde. Für die Informatik bezeichnen wir dieses strategische Ziel als autopoietisches Programmieren und setzen seine Machbarkeit als Arbeitshypothese voraus. Um unmittelbaren praktischen Nutzen zu erhalten, entscheiden wir, einen Vertreter des maschinellen Lernens in Richtung vollkommener Selbstorganisation zu erweitern, wobei wir Grenzen dieses Vorgehens diskutieren, das durch die Natur derzeitiger Programmerzeugung beschränkt ist.

Zur Annäherung an das Ziel bespricht Kapitel 2 den autopoietischen Vorgang der natürlichen Evolution, dem seinerseits selbsterhaltende Systeme entspringen. Daher approximiert künstliche Evolution—ein manuell erzeugtes Zusammenspiel struktureller Variation, Reproduktion und Selektion in technischen Medien—die beabsichtigte Emergenz künstlicher autopoietischer Systeme. Nachfolgend konzentrieren wir uns deswegen auf Evolutionäre Algorithmen: probabilistische, iterative direkte Suchverfahren, die Wirkungsweisen der organischen Evolution (u.a. Phylogenese) verwenden. Unter diesen Algorithmen bietet sich angesichts des strategischen Ziels besonders das Genetische Programmieren (GP) an, da Algorithmen dieser Klasse wiederum Algorithmen produzieren können. Dennoch kämpft ein Benutzer des GP mit dessen unerwünschten Eigenschaften, wie sie typisch sind für alle gegenwärtigen, nur halbautomatischen Problemlöser: kostenintensive manuelle Erzeugung und Wartung als auch problemspezifische Anpassung, wobei letztere besonders kritisch ist, weil der Benutzer gewöhnlich nur unvollständiges Wissen über eine gegebene praktische Problemumgebung besitzt. Zur Verbesserung dieser Situation und zur Anhebung der Leistung eines GP–Systems erscheint uns dessen Selbstanpassung—im Sinne einer automatischen Verhaltensspezialisierung auf ein vorliegendes Problem durch eine Anreicherung des Problemmodells, das ein GP-Lauf erzeugt—als wünschenswert, da sie Autopoiese unterstützt.

Ontogenese, oder Entwicklung, ist die Historie des Strukturwandels eines Systems. Ein biologisches System besticht durch seine *endogene* Entwicklung, die wesentlich für seine Selbstanpassung ist. Systeminhärente genotypische Information, wie sie während der biologischen Phylogenese entsteht, leitet Ontogenese, die die phänotypische Struktur des Systems erzeugt und wandelt, was wiederum dessen Verhalten hervorbringt und spezialisiert.

Kapitel 2–4 schlagen ein formales Basismodell einer nichttrivialen Genotyp-Phänotyp-Abbildung für Suchverfahren vor. Das Modell sowie natürliche ontogenetische Phänomene inspirieren den Entwurf vorteilhafter Abbildungen, die wir in ein GP–Rahmenwerk einbetten, das wir implementieren. Dieser Rahmen repräsentiert Developmental Genetic Programming, eine kleine Teilmenge von GP–Ansätzen, die ontogenetische Aspekte betonen. Gibt man dem Rahmenwerk die triviale Abbildung—die Identität—, so kollabiert es in einen der vielen gewöhnlichen GP–Ansätze.

Kapitel 5–7 entwerfen Spiel- und Echtprobleme für Gedankenexperimente und Versuche mit dem Rahmenwerk, und sie bewerten die empirischen Ergebnisse. In Tendenz zeigt sich, dass manuell fest gegebene, problemspezifische Ontogenese die GP-Leistung fördert. In einer praktischen Umgebung erfordert die Autopoiese eines Systems jedoch, dass dessen strukturelle Komponenten im Fluss verharren. Da diese Elemente die Gesamtfunktion des Systems tragen, die seine Autopoiese beinhaltet, zwingt sich das Konzept der selbstanpassenden Ontogenese auf. Dazu schlagen wir die Auflösung des populationsweiten Entwicklungsmechanismus in individuelle Genotyp-Phänotyp-Abbildungen vor, die durch ebenso individuelle Informationen (genetische Codes) gesteuert werden. Durch Verschmelzung eines Codes mit dem Genotyp des jeweiligen Individuums erhalten wir eine Koevolution beider Informationsarten und damit insbesondere die Anpassung der Ontogenese. Da ein Code einem Genotyp problemspezifische Bedeutung gibt, resultiert also eine automatische Einordnung aller phänotypischen Bausteine nach ihrer vom Verfahren wahrgenommenen Problemrelevanz. Dies löst u.a. die wohlbekannte Aufgabe der Ermittlung einer problemspezifischen, minimalen ausreichenden Symbolmenge für ein GP-Verfahren.

Zusammengefasst besteht der technische Kern der vorliegenden Arbeit aus i) dem Modell des Developmental Genetic Programming (DGP) mit seiner Geno/Phänotyp-Abbildung, und aus ii) der automatischen Regulierung der Redundanz solcher Abbildungen. i) fusst auf genetischen Codes und Reparaturverfahren. Solch ein Verfahren ist der einzige der wesentlichen DGP-Bausteine, der auch dann Bedeutung hat, wenn Geno- und Phänotypen identisch sind. Daher demonstriert Kapitel 8 einen vorteilhaften Verfahrenstyp, das "deleting repair", auf einer weiteren, realistischen Problemdomäne und benutzt dazu die Identität als Abbildung.

Man sieht klar, dass evolvierte Individuen aufgrund des *deleting repair* recht speichersparend ausfallen, was dem problematischen Phänomen des *bloat* vorbeugt, bei dem es sich um unnötige Bestandteile eines Individuums handelt, die oft als Nebeneffekt des Genetischen Programmierens entstehen. Insbesondere kann man daher eine grosszügige maximale Individuengrösse (Länge) festlegen, die GP im frühen Stadium einer Suche Spielraum gibt. Diese Situation führt zu Lösungen unter-

schiedlicher Grösse, was eine notwendige Bedingung für die Auffindung von kürzeren und guten Lösungen ist, die speicherschonend sind. Solche Lösungen können auch kürzere Laufzeiten bedeuten. Darüberhinaus spart *deleting repair* selbst Ressourcen, da es lediglich fehlerhafte Komponenten einer Lösung eliminiert, statt alternative, korrekte Ersatzkomponenten zu berechnen und unter diesen solche zu bestimmen, mit denen dann die fehlerhaften Komponenten ersetzt werden.

Kapitel 9 fasst Ergebnisse zusammen, und Kapitel 10 diskutiert Schlussfolgerungen, wie man die begrenzte Selbstorganisation derzeitiger Suchverfahren nutzen kann, und schlägt, inspiriert von Developmental Genetic Programming, einen Ausweg zum vollständig autopoietischen Rechnen vor.

Bibliography

Adami, C. (1998). *Introduction to Artificial Life*. Telos.

Aguirre, H. E., K. Tanaka, and T. Sugimura (1999, 13-17 July). Cooperative crossover and mutation operators in genetic algorithms. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 1, Orlando FL, pp. 772. Morgan Kaufmann.

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers*. London: Addison-Wesley.

Aiyarak, P., A. S. Saket, and M. C. Sinclair (1997, 1-4 September). Genetic programming approaches for minimum cost topology optimisation of optical telecommunication networks. In *Second International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, University of Strathclyde, Glasgow, UK. IEE.

Alander, J. T. (1994). *An Indexed Bibliography of Genetic Algorithms: Years 1957–1993*. Vaasa, Finland: Art of CAD Ltd.

Alander, J. T. (1995). An indexed bibliography of evolutionary strategies. Technical Report 94-1-ES, University of Vaasa, Department of Information Technology and Production Economics, Vaasa, Finland.

Albuquerque, P., B. Chopard, C. Mazza, and M. Tomassini (2000, 15-16 April). On the impact of the representation on fitness landscapes. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty (Eds.), *Genetic Programming, Proceedings of EuroGP'2000*, Volume 1802 of *LNCS*, Edinburgh, pp. 1–15. Springer-Verlag.

Alliot, J.-M., E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers (Eds.) (1996). *Artificial Evolution*, Volume 1063 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag.

Altenberg, L. (1994). The evolution of evolvability in genetic programming. In K. E. Kinnear, Jr. (Ed.), *Advances in Genetic Programming*, Chapter 3, pp. 47–74. MIT Press.

Andrews, M. and R. Prager (1994). Genetic programming for the acquisition of double auction market strategies. In K. E. Kinnear, Jr. (Ed.), *Advances in Genetic Programming*, Chapter 16, pp. 355–368. MIT Press.

Angeline, P. J. (1993). *Evolutionary Algorithms and Emergent Intelligence*. Ph. D. thesis, Ohio State University.

Angeline, P. J. (1994). Genetic programming and emergent intelligence. In K. E. Kinnear, Jr. (Ed.), *Advances in Genetic Programming*, Chapter 4, pp. 75–98. MIT Press.

Angeline, P. J. (1998, 22-25 July). Subtree crossover causes bloat. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (Eds.), *Genetic Programming 1998: Proceedings of the Third Annual Conference*, University of Wisconsin, Madison, Wisconsin, USA, pp. 745–752. Morgan Kaufmann.

Ashby, W. R. (1956). *An Introduction to Cybernetics*. London: Chapman&Hall.

Ayala, F. and J. Valentine (1979). *Evolving: The theory and process of organic evolution*. Menlo Park, CA: Benjamin.

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice*. Oxford University Press.

Bäck, T., D. B. Fogel, and Z. Michalewicz (Eds.) (1997). *Handbook of Evolutionary Computation*, Bristol, United Kingdom. IOP Publishing and Oxford University Press.

Bäck, T., U. Hammel, and H.-P. Schwefel (1997, May). Evolutionary computation: Comments on the history and current state. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION 1*(1), 3–17.

Ball, P. (1998). *Made to measure*. Princeton, New Jersey: Princeton University Press.

Banzhaf, W. (1993). Genetic programming for pedestrians. MERL Technical Report 93-03, Mitsubishi Electric Research Labs, Cambridge, MA.

Banzhaf, W. (1994, 9-14 October). Genotype-phenotype-mapping and neutral variation – A case study in genetic programming. In Y. Davidor, H.-P. Schwefel, and R. Männer (Eds.), *Parallel Problem Solving from Nature III*, Volume 866 of *Lecture Notes in Computer Science*, Jerusalem, pp. 322–332. Springer-Verlag, Berlin.

Banzhaf, W. and W. B. Langdon (2002, March). Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines 3*(1), 81–91.

Banzhaf, W., P. Nordin, R. E. Keller, and F. D. Francone (1998). *Genetic Programming—An Introduction; On the Automatic Evolution of Computer Programs and Its Application*. dpunkt.verlag, Heidelberg, Germany. Morgan Kaufmann, San Francisco, CA.

Baydar, C. M. and K. Saitou (2000, 10-12 July). A genetic programming framework for error recovery in robotic assembly systems. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, Las Vegas, Nevada, USA, pp. 756. Morgan Kaufmann, San Francisco, CA.

Beielstein, T., C.-P. Ewald, and S. Markon (2003, 12-16 July). Optimal elevator group control by evolution strategies. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller (Eds.), *Genetic and Evolutionary Computation Conference (GECCO–2003)*, Volume 2724 of *Lecture Notes in Computer Science (LNCS)*, Chicago IL, pp. 1702–1714. Springer-Verlag, Berlin.

Bennett III, F. H., J. R. Koza, J. Yu, and W. Mydlowec (2000, 17-19 April). Automatic synthesis, placement, and routing of an amplifier circuit by means of genetic programming. In J. Miller, A. Thompson, P. Thomson, and T. C. Fogarty (Eds.), *Evolvable Systems: From Biology to Hardware, Third International Conference, ICES 2000*, Volume 1801 of *Lecture Notes in Computer Science (LNCS)*, Edinburgh, Scotland, UK, pp. 1–10. Springer-Verlag, Berlin.

Bersano-Begey, T. F. (1997, 13–16 July). Controlling exploration, diversity and escaping local optima in GP: Adapting weights of training sets to model resource consumption. In J. R. Koza (Ed.), *Late Breaking Papers at the 1997 Genetic Programming Conference*, Stanford University, CA, USA, pp. 7–10. Stanford Bookstore.

Beyer, H.-G. (1995). Toward a Theory of Evolution Strategies: The (μ, λ)-Theory. *Evolutionary Computation 2*(4), 381–407.

Bleuler, S., M. Brack, L. Thiele, and E. Zitzler (2001, 27-30 May). Multiobjective genetic programming: Reducing bloat using SPEA2. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, COEX, World Trade Center, Seoul, Korea, pp. 536–543. IEEE Press, Piscataway NJ.

Blickle, T. (1996a, 22-26 September). Evolving compact solutions in genetic programming: A case study. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel (Eds.), *Parallel Problem Solving From Nature IV. Proceedings of the International Conference on Evolutionary Computation*, Volume 1141 of *Lecture Notes in Computer Science (LNCS)*, Berlin, pp. 564–573. Springer–Verlag, Berlin.

Blickle, T. (1996b, November). *Theory of Evolutionary Algorithms and Application to System Synthesis*. Ph. D. thesis, Swiss Federal Institute of Technology, Zurich, Switzerland.

Blickle, T. and L. Thiele (1995, December). A comparison of selection schemes used in genetic algorithms. TIK-Report 11, TIK: Institut für Technische Informatik und Kommunikationsnetze/Computer Engineering and Networks Laboratory, ETH/Swiss Federal Institute of Technology, Zurich, Switzerland.

Box, G. (1957). Evolutionary operation: A method for increasing industrial productivity. *Journal of the Royal Statistical Society C 6*(2), 81–101.

Brameier, M. and W. Banzhaf (2002, 3-5 April). Explicit control of diversity and effective variation distance in linear genetic programming. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi (Eds.), *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, Volume 2278 of *Lecture Notes in Computer Science (LNCS)*, Kinsale, Ireland, pp. 37–49. Springer–Verlag, Berlin.

Brameier, M. and W. Banzhaf (2003, 14-16 April). Neutral variations cause bloat in linear GP. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa (Eds.), *Genetic Programming, Proceedings of the 6th European Conference, EuroGP 2003*, Volume 2610 of *Lecture Notes in Computer Science (LNCS)*, Essex, U.K., pp. 290–299. Springer–Verlag, Berlin.

Brameier, M. and W. Banzhaf (2006). *Linear Genetic Programming*. Number 1 in Genetic and Evolutionary Computation. Springer-Verlag.

Bremermann, H. (1962). Optimization through evolution and recombination. In Yovits, Jacobi, and Goldstein (Eds.), *Self-Organizing Systems*, pp. 93–106. New York: Spartan Books.

Bremermann, H. (1963). Limits of genetic control. *IEEE Transactions MIL-7*, 200–205.

Brooks, S. (1958). A discussion of random methods for seeking maxima. *Operations research 7*, 430–457.

Burke, E., G. Kendall, and E. Soubeiga (2003, Dec). A tabu-search hyperheuristic for timetabling and rostering. *Journal of Heuristics 9*(6), 451–470.

Burke, E. K., M. R. Hyde, and G. Kendall (2006, 9-13 September). Evolving bin packing heuristics with genetic programming. In T. P. Runarsson, H.-G. Beyer, E. Burke, J. J. Merelo-Guervos, L. D. Whitley, and X. Yao (Eds.), *Parallel Problem Solving from Nature - PPSN IX*, Volume 4193 of *LNCS*, Reykjavik, Iceland, pp. 860–869. Springer-Verlag.

Busch, J., J. Ziegler, W. Banzhaf, A. Ross, D. Sawitzki, and C. Aue (2002, 3-5 April). Automatic generation of control programs for walking robots using genetic programming. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi (Eds.), *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, Volume 2278 of *Lecture Notes in Computer Science (LNCS)*, Kinsale, Ireland, pp. 258–267. Springer–Verlag, Berlin.

Cao, H., L. Kang, Y. Chen, and J. Yu (2000, October). Evolutionary modeling of systems of ordinary differential equations with genetic programming. *Genetic Programming And Evolvable Machines 1*(4), 309–337.

Chakhlevitch, K. and P. Cowling (2005, 30 March–1 April). Choosing the fittest subset of low level heuristics in a hyperheuristic framework. In G. R. Raidl and J. Gottlieb (Eds.), *Evolutionary Computation in Combinatorial Optimization – EvoCOP 2005*, Volume 3448 of *LNCS*, Lausanne, Switzerland, pp. 23–33. Springer Verlag.

Chellapilla, K. (1997, September). Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation 1*(3), 209–216.

Claus, V., J. Hopf, and H.-P. Schwefel (Eds.) (1996). *Evolutionary Algorithms and their Application, Dagstuhl-Seminar, März 1996,* Bericht Nr. 140, Wadern. IBFI GmbH, Schloss Dagstuhl.

Comisky, W., J. Yu, and J. R. Koza (2000, 8 July). Automatic synthesis of a wire antenna using genetic programming. In D. Whitley (Ed.), *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, Las Vegas, Nevada, USA, pp. 179–186. Morgan Kaufmann, San Francisco, CA.

Control Data Corporation (1979). *Control Data Corporation: Algol-60, Version 5, Reference Manual, Appendix D.* Control Data Corporation.

Costa, L. D. and M. Schoenauer (2009, 8-12 July). Bringing evolutionary computation to industrial applications with guide. In F. Rothlauf (Ed.), *GECCO-09: Proceedings of the Genetic and Evolutionary Computation Conference*, Montreal, Quebec, Canada, pp. 1467–1474. ACM.

Creighton, T. E. (2002). *Proteins* (2nd ed.). New York: W.H. Freeman and Company.

Crepeau, R. L. (1995, 9 July). Genetic evolution of machine language software. In J. P. Rosca (Ed.), *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, Tahoe City, CA, pp. 121–134. Morgan Kaufmann, San Francisco, CA.

Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection, or The Preservation of Favoured Races in the Struggle for Life.* London, UK: Murray.

Davidson, J. W., D. A. Savic, and G. A. Walters (1999, 1-2 July). Symbolic and numerical regression: a hybrid technique for polynomial approximators. In R. John and R. Birkenhead (Eds.), *Proceedings of Recent Advances in Soft Computing'99*, De Montfort University, Leicester, UK, pp. 111–116. Physica Verlag, Heidelberg, Germany.

Davis, L. (1990). *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold.

Dawkins, R. (1989). *The Selfish Gene*. Oxford, UK: Oxford University Press.

Dittrich, P. (2000). *On Artificial Chemistries*. Ph. D. thesis, University of Dortmund, Dortmund, Germany.

Dittrich, P., A. Bürgel, and W. Banzhaf (1999). Random morphology robot—a test platform for online evolution. Submitted to *Robotics and Autonomous Systems*.

Dowsland, K., E. Soubeiga, and E. Burke (2007). A simulated annealing based hyperheuristic for determining shipper sizes for storage and transportation. *European Journal of Operational Research 179*, 759–774.

Drexler, K. E. (1992). *Nanosystems: Molecular Machinery, Manufacturing, and Computation*. New York: Wiley Interscience.

Duffy, J. and J. Engle-Warnick (1999, 24-26 June). Using symbolic regression to infer strategies from experimental data. In D. A. Belsley and C. F. Baum (Eds.), *Fifth International Conference: Computing in Economics and Finance*, Boston College, MA, pp. 150.

Dunning, T. E. and M. W. Davis (1996, 28–31 July). Evolutionary algorithms for natural language processing. In J. R. Koza (Ed.), *Late Breaking Papers at the Genetic Programming 1996 Conference*, Stanford University, CA, pp. 16–23. Stanford Bookstore, Stanford University, CA.

Ebner, M., M. Shackleton, and R. Shipman (2001). How neutral networks influence evolvability. *Complexity 7*(2), 19–33.

Eigen, M. (1992). *Steps toward Life: a perspective on evolution.* Oxford, UK: Oxford University Press.

Feynman, R. P., R. B. Leighton, and M. Sands (1963). *The Feynman lectures on physics*. Reading, Massachusetts; Palo Alto, CA; London, UK: Addison-Wesley publishing company, Inc.

Fincham, J. R. (1994). *Genetic Analysis*. Oxford, UK: Blackwell Science.

Fogel, L., A. Owens, and M. Walsh (1966). *Artificial Intelligence through Simulated Evolution*. New York: Wiley.

Gathercole, C. (1998). *An Investigation of Supervised Learning in Genetic Programming*. Ph. D. thesis, University of Edinburgh, UK.

Gaw, A., P. Rattadilok, and R. Kwan (2004). Distributed choice function hyperheuristics for timetabling and scheduling. In *Proceedings of the 2004 International Conference on the Practice and Theory of Automated Timetabling (PATAT 2004), Pittsburgh USA*, pp. 495–497.

Gerdes, I. (1996, 2–5 September). Application of evolutionary algorithms to the free flight concept for aircraft. In H.-J. Zimmermann (Ed.), *EUFIT'96: FOURTH EUROPEAN CONGRESS ON INTELLIGENT TECHNIQUES AND SOFT COMPUTING*, Aachen, Germany, pp. 1445–1449. Wissenschaftsverlag, Mainz, Germany.

Gillett, B. (1977). *Introduction to Operations Research*. McGraw-Hill, New York.

Globus, A., J. Lawton, and T. Wipke (1998, November 12-15). Automatic molecular design using evolutionary techniques. In A. Globus and D. Srivastava (Eds.), *The Sixth Foresight Conference on Molecular Nanotechnology*, Westin Hotel in Santa Clara, CA.

Glover, F. and M. Laguna (1997). *Tabu Search*. Springer.

Goldberg, D. (1988). *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading MA: Addison-Wesley.

Gordon, R. (1994). Evolution escapes rugged fitness landscapes by gene or genome doubling: the blessing of higher dimensionality. *Computers & Chemistry 18*(3), 325–332.

Gruau, F. (1994). *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm*. Ph. D. thesis, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon, France.

Guyaguler, B. (2000, June). Regression on petroleum well test data with the reservoir model as a parameter. In J. R. Koza (Ed.), *Genetic Algorithms and Genetic Programming at Stanford 2000*, pp. 188–197. Stanford CA: Stanford Bookstore.

Hansen, J. V. (2003, March). Genetic programming experiments with standard and homologous crossover methods. *Genetic Programming and Evolvable Machines 4*(1), 53–66.

Harbison, S. P. and G. L. Steele (1995). *C - A Reference Manual* (4th ed.). Englewood Cliffs, NJ: Prentice Hall.

Hemmi, H., J. Mizoguchi, and K. Shimohara (1994, 6-8 July). Development and evolution of hardware behaviours. In R. Brooks and P. Maes (Eds.), *Artificial Life IV, Proceedings of the fourth International Workshop on the Synthesis and Simulation of Living Systems*, pp. 371–376. Cambridge MA: MIT press.

Heylighen, F. (2002). Web dictionary of cybernetics and systems. Internet.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: The University of Michigan Press.

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems*. Cambridge, MA: MIT Press. Note: This edition adds, in particular, a correction of a proof and thoughts on the evolution of ecological systems. R.K.

Holland, J. H. (1995). *Hidden order*. Reading MA: Addison-Wesley.

Hopcroft, J. E. and J. D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison Wesley.

Iba, H. (1999, 13-17 July). Bagging, boosting, and bloating in genetic programming. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, Volume 2, Orlando FL, pp. 1053–1060. Morgan Kaufmann, San Francisco, CA.

Janikow, C. Z. (1999, 13 July). Constrained genetic programming. In T. S. Hussain (Ed.), *Advanced Grammar Techniques Within Genetic Programming and Evolutionary Computation*, Orlando, Florida, USA, pp. 80–82.

Jayalakshmi, G., S. Sathiamoorthy, and R. Rajaram (2001). An hybrid genetic algorithm — a new approach to solve traveling salesman problem. *International Journal of Computational Engineering Science 2*(2), 339–355.

Kargupta, H. (2001). A striking property of genetic code-like transformations. *Complex Systems 11*(1), 1–29.

Kargupta, H., R. Ayyagari, and S. Ghosh (2003). Learning functions using randomized expansions: Probabilistic properties and experimentations. *IEEE Transactions on Knowledge and Data Engineering 16*(8), 894–908.

Kauffman, S. A. (1993). *The Origins of Order: Self-Organization and Selection in Evolution*. New York: Oxford University Press.

Keller, R. E. (2001). Autopoietic solutions for real-world problems. In *EvoDebates: Fit for the Future*. EvoNet – The European Network of Excellence in Evolutionary Computing. Internet.

Keller, R. E. (2002). Evolutionary methods for data mining. In J. Meij (Ed.), *Dealing with the data flood*. The Hague, Netherlands: STT Netherlands Study Centre for Technology Trends.

Keller, R. E. and W. Banzhaf (1994, June). Explicit maintenance of genetic diversity on genospaces. Internet.

Keller, R. E. and W. Banzhaf (1996, 28–31 July). Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, pp. 116–122. MIT Press, Cambridge MA.

Keller, R. E. and W. Banzhaf (1999, 13–17 July). The evolution of genetic code in genetic programming. In W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Honavar, M. Jakiela, and R. Smith (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, Orlando FL, pp. 1077–1082. Morgan Kaufmann, San Francisco, CA.

Keller, R. E. and W. Banzhaf (2001, 7–11 July). The evolution of genetic code on a hard problem. In L. Spector, W. B. Langdon, A. Wu, H.-M. Voigt, and M. Gen (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, San Francisco, CA, pp. 50–56. Morgan Kaufmann, San Francisco, CA.

Keller, R. E., W. Banzhaf, J. Mehnen, and K. Weinert (1999). CAD surface reconstruction from digitized 3D point data with a genetic programming/evolution strategy hybrid. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline (Eds.), *Advances in Genetic Programming 3*, Chapter 3, pp. 41–65. Cambridge MA: MIT Press.

Keller, R. E., W. A. Kosters, M. van der Vaart, and M. D. Witsenburg (2002, 21-22 October). Genetic programming produces strategies for agents in a dynamic environment. In H. Blockeel and M. Denecker (Eds.), *Proceedings of the Fourteenth Belgium/Netherlands Conference on Artificial Intelligence (BNAIC'02)*, Leuven, Belgium, pp. 171–178. Katholieke Universiteit Leuven, Belgium.

Kimura, M. (1968). Evolutionary rate at the molecular level. *Nature 217*, 624–626.

Kimura, M. (1983). *The Neutral Theory of Molecular Evolution*. Cambridge, UK: Cambridge University Press.

Kinnear, Jr., K. E. (1994, 27-29 June). Fitness landscapes and difficulty in genetic programming. In *Proceedings of the 1994 IEEE World Conference on Computational Intelligence*, Volume 1, Orlando, FL, pp. 142–147. IEEE Press, Piscataway, NJ.

Knickmeier, F. (1992). Auslegung eines Rechnernetzwerkes mit minimalem Kommunikationsaufwand mittels evolutionärer Algorithmen. Master's thesis, Universität Dortmund, Germany.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge MA: MIT Press.

Koza, J. R., M. A. Keane, F. H. Bennett, J. Yu, W. Mydlowec, and O. Stiffelman (1999, 13-17 July). Searching for the impossible using genetic programming. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (Eds.), *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 2, Orlando FL, pp. 1083–1091. Morgan Kaufmann, San Francisco, CA.

Koza, J. R., W. Mydlowec, G. Lanza, J. Yu, and M. A. Keane (2001, 3-7 January). Reverse engineering of metabolic pathways from observed data using genetic programming. In *Pacific Symposium on Biocomputing 6*, Hawaii, pp. 434–445. World Scientific press.

Langdon, W. B. (2000, April). Size fair and homologous tree genetic programming crossovers. *Genetic Programming And Evolvable Machines 1*(1/2), 95–119.

Langdon, W. B. and S. J. Barrett (2004). Genetic programming in data mining for drug discovery. In A. Ghosh and L. C. Jain (Eds.), *Evolutionary Computing in Data Mining*. Heidelberg, Germany: Physica Verlag. To be published.

Langdon, W. B. and R. Poli (1997, 23-27 June). Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant (Eds.), *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*, pp. 13–22. Springer-Verlag, London, UK.

Langdon, W. B. and R. Poli (1998, January). Why ants are hard. Technical Report CSRP-98-4, University of Birmingham, School of Computer Science, UK.

Lawler, E., J. Lenstra, A. R. Kan, and D. Shmoys (Eds.) (1985). *The Travelling Salesman Problem*. Chichester: Wiley.

Lohnert, F., A. Schütte, J. Sprave, I. Rechenberg, I. Boblan, U. Raab, I. S. Koref, W. Banzhaf, R. Keller, J. Niehaus, and H. Rauhe (2001). Genetisches Programmieren für Modellierung und Regelung dynamischer Systeme. Technical report, DaimlerChrysler AG, Forschung und Technologie FT3/AI, Berlin, Germany; Technische Universität Berlin, Bionik und Evolutionstechnik; Universiteit Leiden, Leiden Institute of Advanced Computer Science, The Netherlands; Informatik Centrum Dortmund e.V., Germany.

Lourenco, H. R., O. C. Martin, and T. Stutzle (2002). Iterated local search. *ISORMS 57*, 321–353.

Luke, S. and L. Spector (1997, 13-16 July). A comparison of crossover and mutation in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, CA, pp. 240–248. Morgan Kaufmann, San Francisco, CA.

Luke, S. and L. Spector (1998, 22-25 July). A revised comparison of crossover and mutation in genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (Eds.), *Genetic Programming 1998: Proceedings of the Third Annual Conference*, University of Wisconsin, Madison, pp. 208–213. Morgan Kaufmann, San Francisco, CA.

MacLennan, B. J. (2002, October). Universally programmable intelligent matter. Technical report, University of Tennessee.

Madou, M. (1997). *Fundamentals of Microfabrication*. New York: CRC Press.

Maeshiro, T. (1997). *Structure of Genetic Code and its Evolution*. Ph. D. thesis, School of Information Science, Japan Adv. Inst. of Science and Technology.

Margetts, S. and A. J. Jones (2001, 18-20 April). An adaptive mapping for developmental genetic programming. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon (Eds.), *Genetic Programming, Proceedings of EuroGP'2001*, Volume 2038 of *Lecture Notes in Computer Science (LNCS)*, Lake Como, Italy, pp. 97–107. Springer–Verlag, Berlin.

Maturana, H. and F. Varela (1980). *Autopoiesis and Cognition*. Dordrecht: H.D. Reidel.

McPhee, N. F. and N. J. Hopper (1999, 13-17 July). Analysis of genetic diversity through population history. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (Eds.), *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, Volume 2, Orlando FL, pp. 1112–1120. Morgan Kaufmann, San Francisco, CA.

Meinhardt, H. (1982). *Models of biological pattern formation*. London/New York: Academic Press.

Miller, J. F. and P. Thomson (2000, 15-16 April). Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty (Eds.), *Genetic Programming, Proceedings of EuroGP'2000*, Volume 1802 of *Lecture Notes in Computer Science (LNCS)*, Edinburgh, UK, pp. 121–132. Springer-Verlag, Berlin, Germany.

Minister, J.-B. H., N. P. Williams, T. G. Masters, J. F. Gilbert, and J. S. Haase (1995, 1-3 March). Application of evolutionary programming to earthquake hypocenter determination. In J. R. McDonnel, R. G. Reynolds, and D. B. Fogel (Eds.), *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*, San Diego, CA, pp. 3–17. MIT Press, Cambridge MA.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. Cambridge MA: MIT Press.

Mitchell, T. M. (1997). *Machine Learning*. New York: McGraw-Hill.

Montana, D. J. (1995). Strongly typed genetic programming. *Evolutionary Computation 3*(2), 199–230.

Motoki, T. (2002). Calculating the expected loss of diversity of selection schemes. *Evolutionary Computation 10*(4), 397–422.

Mühlenbein, H. (1993). Evolutionary algorithms: Theory and applications. In E. Aarts and J. Lenstra (Eds.), *Local Search in Combinatorial Optimization*. New York: Wiley.

Mukai, T. (1985). Experimental verification of the neutral theory. In T. Ohta (Ed.), *Population Genetics and Molecular Evolution*. Berlin: Springer-Verlag.

Nabuurs, R. (2004). Energy-bound genetic programming. Master's thesis, University of Leiden, The Netherlands.

Nissen, V. and J. Biethahn (1995). An introduction to evolutionary algorithms. In J. Biethahn and V. Nissen (Eds.), *Evolutionary Algorithms in Management Applications*, pp. 3–43. Berlin: Springer-Verlag.

Nordin, P. (1994). A compiling genetic programming system that directly manipulates the machine code. In K. E. Kinnear, Jr. (Ed.), *Advances in Genetic Programming*, Chapter 14, pp. 311–331. MIT Press, Cambridge MA.

Nordin, P. (1997). *Evolutionary Program Induction of Binary Machine Code and its Applications*. Ph. D. thesis, University of Dortmund, Computer Science Dept.

Nordin, P. and W. Banzhaf (1995, 15-19 July). Complexity compression and evolution. In L. Eshelman (Ed.), *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, Pittsburgh PA, pp. 310–317. Morgan Kaufmann, San Francisco, CA.

Nordin, P., W. Banzhaf, and F. D. Francone (1999, June). Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline (Eds.), *Advances in Genetic Programming 3*, Chapter 12, pp. 275–299. Cambridge MA: MIT Press.

Nordin, P., F. Francone, and W. Banzhaf (1995, 9 July). Explicitly defined introns and destructive crossover in genetic programming. In J. P. Rosca (Ed.), *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, Tahoe City, CA, pp. 6–22.

Oltean, M. (2005, Fall). Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation 13*(3), 387–410.

O'Neill, M. (2001, August). *Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution*. Ph. D. thesis, University Of Limerick, Ireland.

O'Neill, M. and C. Ryan (2003). *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, Volume 4 of *Genetic programming*. Kluwer Academic Publishers.

O'Neill, M. and C. Ryan (2004, 5-7 April). Grammatical evolution by grammatical evolution: The evolution of grammar and genetic code. In M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule (Eds.), *Genetic Programming, Proceedings of the 7th European Conference, EuroGP 2004*, Volume 3003 of *Lecture Notes in Computer Science (LNCS)*, Coimbra, Portugal, pp. 138–149. Springer–Verlag, Berlin.

O'Reilly, U.-M. and F. Oppacher (1995, 31 July–2 August). The troubling aspects of a building block hypothesis for genetic programming. In L. D. Whitley and M. D. Vose (Eds.), *Foundations of Genetic Algorithms 3*, Estes Park, CO, pp. 73–88. Morgan Kaufmann, San Francisco, CA.

Ortega-Sanchez, C., D. Mange, S. Smith, and A. Tyrrell (2000, July). Embryonics: A bio-inspired cellular architecture with fault-tolerant properties. *Genetic Programming And Evolvable Machines 1*(3), 187–215.

Paterson, N. and M. Livesey (1997, 13-16 July). Evolving caching algorithms in C by genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, CA, pp. 262–267. Morgan Kaufmann, San Francisco, CA.

Paterson, N. R. and M. Livesey (1996, 28–31 July). Distinguishing genotype and phenotype in genetic programming. In J. R. Koza (Ed.), *Late Breaking Papers at the Genetic Programming 1996 Conference*, Stanford University, CA, pp. 141–150. Stanford Bookstore, Stanford CA.

Platel, M. D., M. Clergue, and P. Collard (2003, 14-16 April). Maximum homologous crossover for linear genetic programming. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa (Eds.), *Genetic Programming, Proceedings of the 6th European Conference, EuroGP 2003*, Volume 2610 of *Lecture Notes in Computer Science (LNCS)*, Essex, UK, pp. 200–210. Springer–Verlag, Berlin, Germany.

Podgorelec, V. and P. Kokol (2000, 15-16 April). Fighting program bloat with the fractal complexity measure. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty (Eds.), *Genetic Programming, Proceedings of the 3rd European Conference, EuroGP 2000*, Volume 1802 of *Lecture Notes in Computer Science (LNCS)*, Edinburgh, pp. 326–337. Springer–Verlag, Berlin, Germany.

Poli, R. (2001, 18-20 April). General schema theory for genetic programming with subtree-swapping crossover. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon (Eds.), *Genetic Programming, Proceedings of the 4th European Conference, EuroGP 2001*, Volume 2038 of *Lecture Notes in Computer Science (LNCS)*, Lake Como, Italy, pp. 143–159. Springer-Verlag, Berlin, Germany.

Porter, M., M. Willis, and H. Hiden (1996). Computer-aided polymer design using genetic programming. Technical report, Chemical Engineering, Newcastle University, UK.

Prusinkiewicz, P. and A. Lindenmayer (1990). *The Algorithmic Beauty of Plants*. Berlin: Springer.

Quagliarella, D., J. Periaux, C. Poloni, and G. Winter (Eds.) (1998). *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, Chichester, UK. John Wiley and Sons.

Rechenberg, I. (1971). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Ph. D. thesis, Technische Universität Berlin, Germany.

Rechenberg, I. (1989). Evolution strategy: Nature's way of optimization. In H. Bergmann (Ed.), *Optimization: Methods and Applications, Possibilities and Limitations*, Volume 47 of *DLR lecture notes in engineering*, pp. 106–126. Berlin, Germany: Springer.

Rechenberg, I. (Ed.) (1994). *Evolutionsstrategie '94*, Volume 1 of *Werkstatt Bionik und Evolutionstechnik*. Frommann-Holzboog, Stuttgart, Germany.

Reinelt, G. (2007). *URL: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/* .

Rosca, J. (1996, 28–31 July). Generality versus size in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, pp. 381–387. MIT Press, Cambridge MA.

Rosca, J. P. (1995, 9 July). Entropy-driven adaptive representation. In J. P. Rosca (Ed.), *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, Tahoe City, CA, pp. 23–32.

Ross, P. (2005). Hyperheuristics. In E. Burke and G. Kendall (Eds.), *Search Methodologies*, pp. 529–556. Berlin, Heidelberg, New York: Springer-Verlag.

Rudolph, G. (1996). *Convergence Properties of Evolutionary Algorithms*. Ph. D. thesis, Computer Science Dept., University of Dortmund, Germany.

Samsonova, E. (2002, June). HELIX: An Artificial Life system modeling DNA. Master's thesis, Faculty of Mathematics and Natural Sciences, University of Leiden, The Netherlands.

Schoenauer, M., M. Sebag, F. Jouve, B. Lamy, and H. Maitournam (1996). Evolutionary identification of macro-mechanical models. In P. J. Angeline and K. E. Kinnear, Jr. (Eds.), *Advances in Genetic Programming 2*, Chapter 23, pp. 467–488. Cambridge MA: MIT Press.

Schrödinger, E. (1944). *What is Life?* Cambridge, UK: Cambridge University Press.

Schwefel, H.-P. (1975, May). *Evolutionsstrategie und numerische Optimierung*. Ph. D. thesis, Technische Universität Berlin, Germany.

Schwefel, H.-P. (1981). *Numerical optimization of computer models*. New York: Wiley.

Schwefel, H.-P. (1995). *Evolution and optimum seeking*. New York: Wiley Interscience.

Schwefel, H.-P., I. Wegener, and K. Weinert (Eds.) (2003). *Advances in Computational Intelligence—Theory and Practice*. Springer, Berlin, Germany.

Sims, K. (1994). Evolving 3d morphology and behavior by competition. In R. Brooks and P. Maes (Eds.), *Artificial Life IV*. MIT Press, Cambridge MA.

Smith, J. M. and E. Szathmáry (1995). *The major transitions in evolution*. Oxford, UK: W.H. Freeman/Spektrum.

Smolin, L. (2004, January). Atoms of space and time. *Scientific American*, 56–65.

Soubeiga, E. (2003). *Development and application of hyper-heuristics to personnel scheduling*. Ph. D. thesis, Computer Science, University of Nottingham.

Spector, L. and K. Stoffel (1996, 28–31 July). Ontogenetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, pp. 394–399. MIT Press, Cambridge MA.

Stoffel, K. and L. Spector (1996, 28–31 July). High-performance, parallel, stack-based genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, pp. 224–229. MIT Press, Cambridge MA.

Student project team 272 "Philia" (1997, September). Final report: Implementation and Application of a Genetic Programming/Artificial Life System (German original: "Endbericht: Realisierung und Anwendung eines GP/AL-Systems"). Technical report, Computer Science Department, University of Dortmund, Germany.

Teller, A. (1996). Evolving programmers: The co-evolution of intelligent recombination operators. In P. J. Angeline and K. E. Kinnear, Jr. (Eds.), *Advances in Genetic Programming 2*, Chapter 3, pp. 45–68. Cambridge MA: MIT Press.

Vester, F. (1980). *Neuland des Denkens*. Stuttgart: DVA.

Watson, J. D., N. H. Hopkins, J. W. Roberts, J. A. Steitz, and A. M. Weiner (1992). *Molecular Biology of the Gene*. Menlo Park, CA: Benjamin Cummings.

Weber, B. H., D. J. Depew, and J. D. Smith (Eds.) (1988). *Entropy, Information, and Evolution*. MIT Press, Cambridge MA.

Wegener, I. (1993). *Theoretische Informatik*. Stuttgart: B.G. Teubner.

Whigham, P. A. (1996, 28–31 July). Search bias, language bias, and genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, USA, pp. 230–237. MIT Press.

Wiener, N. (1948). *Cybernetics*. New York: Wiley.

Wolfram, S. (1994). *Cellular Automata and Complexity*. Reading, MA: Addison-Wesley.

Wolpert, D. H. and W. G. Macready (1997, April). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation 1*(1), 67–82.

Wong, M. L. and K. S. Leung (1995, 29 November - 1 December). Applying logic grammars to induce sub-functions in genetic programming. In *1995 IEEE Conference on Evolutionary Computation*, Volume 2, Perth, Australia, pp. 737–740. IEEE Press.

Wood, D. (1987). *Theory of Computation*. New York: Wiley.

Wu, A. S. and I. Garibay (2002, June). The proportional genetic algorithm: Gene expression in a genetic algorithm. *Genetic Programming and Evolvable Hardware 3*(2), 157–192.

Yu, T. and P. Bentley (1998, 27-30 September). Methods to evolve legal phenotypes. In A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel (Eds.), *Fifth International Conference on Parallel Problem Solving from Nature*, Volume 1498 of *Lecture Notes in Computer Science (LNCS)*, Amsterdam, pp. 280–291. Springer, Berlin, Germany.

Yu, T. and J. Miller (2001, 18-20 April). Neutrality and the evolvability of boolean function landscape. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon (Eds.), *Genetic Programming, Proceedings of the 4th European Conference, EuroGP 2001*, Volume 2038 of *Lecture Notes in Computer Science (LNCS)*, Lake Como, Italy, pp. 204–217. Springer, Berlin, Germany.

Yu, T. and J. F. Miller (2002, 3-5 April). Needles in haystacks are not hard to find with neutrality. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi (Eds.), *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, Volume 2278 of *Lecture Notes in Computer Science (LNCS)*, Kinsale, Ireland, pp. 13–25. Springer–Verlag, Berlin.

Zibo, Z. and F. Naghdy (1995, 29 November–1 December). Application of genetic algorithms to system identification. In *1995 IEEE International Conference on Evolutionary Computation*, Perth, Australia, pp. 777–782. IEEE press, Piscataway NJ.

Zomorodian, A. (1995, 10–12 November). Context-free language induction by evolution of deterministic push-down automata using genetic programming. In E. V. Siegel and J. R. Koza (Eds.), *Working Notes for the AAAI Symposium on Genetic Programming*, MIT, Cambridge MA, pp. 127–133. AAAI press, Menlo Park, CA.

List of Figures

List of Tables

Name Index

Subject Index