Poznan University of Technology
Institute of Computing Science

Algorithms for Test-Based Problems

Wojciech Jaśkowski

A dissertation submitted to

the Council of the Faculty of Computing and Information Science

in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Supervisor

Krzysztof Krawiec, PhD Dr Habil.

Poznań, Poland

2011

To my dearest wife Dobrosia

In loving memory of my father Piotr Jaśkowski

Acknowledgments

The work described here was carried out between October 2006 and May 2011 in the Laboratory of Intelligent Decision Support Systems at the Faculty of Computing and Information Science at Poznan University of Technology. Numerous people helped me through this effort and I'd like to mention some of them here. Most of all, I deeply thank my supervisor Krzysztof Krawiec for a lot of heart he put in guiding me. His enthusiasm, inspiration, insightful remarks, and, last but not least, gentle motivation were all invaluable. I will never forget the atmosphere of late-night hours spent together online writing last-minute papers for some conferences. I would like also to express my gratitude to my collaborators and friends Bartosz Wieloch and Marcin Szubert for insightful discussions, cooperation and for what I have learned from them.

I gratefully acknowledge the grant #NN516188337 from the Polish Ministry of Sciences I was awarded for the time period 09.2009–08.2011 and a scholarship from Sectoral Operational Programme 'Human Resources Development', Activity 8.2, co-financed by the European Social Fund and the Government of Poland which I was awarded in 2009.

Finally, my deepest thanks, love, and affection go to my parents Ewa and Piotr, my brother Maciej, and my wife Dobrosia. For their love, patience, and support I am forever grateful.

Poznań, in May 2011

Abstract

Problems in which some elementary entities interact with each other are common in computational intelligence. This scenario, typical for coevolving artificial-life agents, learning strategies for games, and machine learning from examples, can be formalized as a *test-based problem* and conveniently embedded in the common conceptual framework of coevolution. In test-based problems candidate solutions are evaluated on a number of test cases such as agents, opponents or examples. Although coevolutionary algorithms proved successful in some applications, they also turned out to have hard to predict dynamics and fail to sustain progress during a run, thus being unable to obtain competitive solutions for many test-based problems.

It has been recently shown that one of the reasons why coevolutionary algorithms demonstrate such undesired behavior is the aggregation of results of interactions between individuals representing candidate solutions and tests, which typically leads to characterizing the performance of an individual by a single scalar value. In order to remedy this situation, in the thesis, we make an attempt to get around the problem of aggregation using two methods.

First, we introduce *Fitnessless Coevolution*, a method for symmetrical test-based problems. Fitnessless Coevolution plays games between individuals to settle tournaments in the selection phase and skips the typical phase of evaluation and the aggregation of results connected with it. The selection operator applies a single-elimination tournament to a randomly drawn group of individuals, and the winner of the final round becomes the result of selection. Therefore, Fitnessless Coevolution does not involve explicit fitness measure and no aggregation of interaction results is required. We prove that, under a condition of transitivity of the payoff matrix, the dynamics of Fitnessless Coevolution is identical to that of the traditional evolutionary algorithm. The experimental results, obtained on a diversified group of problems, demonstrate that Fitnessless Coevolution is able to produce solutions that are equally good or better than solutions obtained using fitness-based one-population coevolution with different selection methods. In a case study, we provide the complete record of methodology that let us evolve *BrilliAnt*, the winner of the Ant Wars contest. We detail the coevolutionary setup that lead to Bril-

liAnt's emergence, assess its direct and indirect human-competitiveness, and describe the behavioral patterns observed in its strategy.

Second, we study the consequences of the fact that the problem of aggregation of interaction results may be got around by regarding every test of a test-based problem as a separate objective, and the whole problem as a multi-objective optimization task. Research on reducing the number of objectives while preserving the relations between candidate solutions and tests led to the notions of *underlying objectives* and *internal problem structure*, which can be formalized as a coordinate system that spatially arranges candidate solutions and tests. The coordinate system that spans the minimal number of axes determines the so-called *dimension of a problem* and, being an inherent property of every test-based problem, is of particular interest. We investigate in-depth the formalism of coordinate system and its properties, relate them to the properties of partially ordered sets, and design an exact algorithm for finding a minimal coordinate system. We also prove that this problem is NP-hard and come up with a heuristic which is superior to the best algorithm proposed so far. Finally, we apply the algorithms to several benchmark problems to demonstrate that the dimension of a problem is typically much lower than the number of tests. Our work suggest that for at least some classes of test-based problems, the dimension of a problem may be proportional to the logarithm of number of tests.

Based on the above-described theoretical results, we propose a novel coevolutionary archive method founded on the concept of coordinate systems, called Coordinate System Archive (COSA), and compare it to two state-of-the-art archive methods, IPCA and LAPCA. Using two different objective performance measures, we find out that COSA is superior to these methods on a class of artificial test-based problems.

Preface

Some research and portions of text presented in this dissertation appear in the following publications:

- **Chapter 3** Wojciech Jaśkowski, Bartosz Wieloch, and Krzysztof Krawiec. Fitnessless Coevolution. In Maarten Keijzer et al., editor, *GECCO '08: Proceedings of the 10th annual conference on genetic and evolutionary computation*, pages 355–362, Atlanta, GA, USA, 2008. ACM [65].

- **Chapter 4** Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch. Winning Ant Wars: evolving a human-competitive game strategy using fitnessless selection. In M. O'Neill et al., editor, *Genetic Programming 11th European Conference*, EuroGP 2008, Proceedings, volume 4971 of *LNCS*, pages 13–24. Springer-Verlag, 2008 [64]
  later extended as
  Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch. Evolving strategy for a probabilistic game of imperfect information using genetic programming. *Genetic Programming and Evolvable Machines*, 9(4):281–294, 2008 [63].

- **Chapter 5** Wojciech Jaśkowski and Krzysztof Krawiec. How many dimensions in co-optimization? In *Genetic and Evolutionary Computation Conference*, (in press), Dublin, Ireland, 2011 [61]
  and
  Wojciech Jaśkowski and Krzysztof Krawiec. Formal analysis and algorithms for extracting coordinate systems of games. In *IEEE Symposium on Computational Intelligence and Games*, pages 201–208, Milano, Italy, 2009 [59]

- **Chapter 6** Wojciech Jaśkowski and Krzysztof Krawiec. Coordinate system archive for coevolution. In *IEEE World Congress on Computational Intelligence*, pages 1–10, Barcelona, 2010 [60].

Contents

1. Introduction

1.1. Problem Setting and Motivation

This work considers problems which originated in computational intelligence [38], a field dedicated to problem solving, mostly by means of nature-inspired algorithms. Computational intelligence has much in common with artificial intelligence. The difference between the two has been pertinently expressed by Lucas [85, page 45]: "In AI research the emphasis is on producing apparently intelligent behaviour using whatever techniques are appropriate for a given problem. In computational intelligence research, the emphasis is placed on intelligence being an emergent property". The disciplines that are typically associated with computational intelligence include, among others, artificial neural networks, evolutionary computation, multi-agent systems, and swarm intelligence.

A significant part of computational intelligence research is devoted to solving problems in which some elementary entities interact with each other. Game-playing agents learn by playing against opponent strategies. Machine learning algorithms generate hypotheses and test them on examples. Evolutionary algorithms simulate the evolved designs in various environmental conditions. What these scenarios have in common is the underlying concept of an elementary *interaction* between a *candidate solution* (strategy, hypothesis, design) and a *test* (opponent strategy, training example, environmental conditions, respectively). In each of them, an interaction has an *outcome*, which can be typically expressed by a scalar. Another feature they share is the fact that the number of tests can be large or even infinite, which often rules out the possibility of confronting candidate solutions with all tests. Therefore, an outcome of a single interaction provides only limited information about the underlying phenomenon.

These characteristics clearly delineate a class of *test-based problems* [17]. To solving a test-based problem one has to find the candidate solutions that are, in some sense, superior to other candidate solutions with respect to the outcomes they receive when interacting with tests. As determining the outcomes of interactions between all candidate solutions and all tests is computationally intractable, a practical algorithm for solving test-based problems has to decide which interactions are worth computing and which are

not. Although there exist many algorithms tailored to specific forms of such problems, e.g., statistical methods for machine learning or minimax tree search for game-playing, *competitive coevolutionary algorithms* [4, 54, 119, 120] constitute the most generic framework for solving test-based problems. The essence of coevolution is that the results of interactions between candidate solutions and tests propel the search process in the space of candidate solutions and the space of tests, and substitute for an objective, external fitness measure found in traditional evolutionary algorithms. Thus, the driving force of coevolutionary algorithms is the constant *arms race* between individuals playing the roles of candidate solutions and individuals that represent tests. Apart from this difference, coevolutionary algorithms typically use neo-Darwinian mechanisms similar to the ones used in evolutionary algorithms like mutation, crossover and selection.

Since competitive interaction is common in nature and mankind history, the elegance of virtual arms race attracted many researchers to apply coevolutionary algorithms to mathematical and engineering problems. As a result, coevolutionary methods have been applied to many test-based problems such as design of sorting networks [54], learning strategies for board and strategic games [121], or evolving rules for cellular automata [70].

Despite initial enthusiasm in the field, coevolutionary algorithms turned out to have hard to predict dynamics meant as the way in which they navigate a search space [68]. Analyzing, modeling, and understanding that dynamics turned out to be very hard [116]. In particular, many coevolutionary algorithms applied to nontrivial problems tend to fail to sustain progress during a run [96] and to converge towards solutions that have desired properties (corresponding to basins of global optima in conventional learning and optimization problems, where the objective fitness measure is known). This raises justified questions about reasonable stopping criteria. Moreover, many undesired patterns in coevolutionary algorithms' runs have been observed. Consequently, in many cases of test-based problems, coevolutionary algorithms have been unable to obtain high-quality solutions.

It has been recently shown that one of the reasons why coevolutionary algorithms often fail to succeed (and exhibit, so-called, *pathologies* described in this thesis) is the *aggregation* of outcomes of interactions between individuals representing candidate solutions and tests [15]. Such aggregation typically leads to characterizing the performance of an individual during a run by a single scalar value. Thus, there is a need for new methods in coevolutionary algorithms which would avoid aggregation. The quest for such methods is the main motivation for the research described in this thesis.

1.2. Aims and Scope

In the context of discussion outlined in the previous section, the overall goal of this thesis is *to analyze test-based problems, their properties, and to design novel coevolutionary methods that solve them while avoiding the problem of aggregation.* It is emphasized that the proposed methods should have strong theoretical foundations.

This goal should be satisfied by fulfilling the following objectives:

- To design a coevolutionary algorithm that does not aggregate results of interactions during the evaluation phase.
- To theoretically analyze its dynamics with reference to traditional evolutionary algorithms.
- To theoretically analyze the internal structure of test-based problem based on the concept of Pareto-coevolution.
- To design an effective algorithm for extraction of internal structure from test-based problems.
- To design an algorithm for test-based problems using the idea of internal structure extraction.
- To experimentally verify the proposed concepts and algorithms on artificial problems of known internal structure as well as on real-world problems of unknown internal structure.

The dissertation is organized as follows. **Chapter 2** provides the preliminary mathematical definitions, background information about test-based problems and coevolution, and points out related work in those fields. In **Chapter 3** we introduce *Fitnessless Coevolution*, a novel method of comparative one-population coevolution, which avoids aggregation into explicit fitness measure by using a single-elimination tournament to a randomly drawn group of individuals. We prove that, under a condition of transitivity of the payoff matrix, the dynamics of Fitnessless Coevolution is identical to that of the traditional evolutionary algorithm. The experimental results, obtained on a diversified group of problems, demonstrate that Fitnessless Coevolution is able to produce solutions that are not worse than solutions obtained using other methods. This is theme is continued in **Chapter 4**, where, in a case study, we apply Fitnessless Coevolution to a certain game of imperfect information using genetic programming. We provide the

complete record of the employed methodology as well as an in-depth evaluation of the obtained solutions.

In **Chapter 5** we move to more advanced topics and investigate the formalism of coordinate system for test-based problems and its properties. We relate them to properties of partially ordered sets, and design an exact algorithm for finding a minimal coordinate system. We also prove that the problem of finding the minimal coordinate system for a given problem is NP-hard and come up with a heuristic that is superior to the best algorithm proposed so far. Finally, we apply the algorithms to three abstract problems and demonstrate that the the number of axes in such minimal coordinate system called the dimension of the problem is typically much lower than the number of tests and for a random problem it seems to follow the logarithmic curve.

In **Chapter 6**, we propose a novel coevolutionary archive method, called Coordinate System Archive (COSA) that is based on the concepts introduced in the Chapter **5**. In the experimental part, we compare COSA to two state-of-the-art archive methods, IPCA and LAPCA, for two different problems.

Finally, **Chapter 7** sums up the work and points to future directions of work.

2. Background

In this chapter, after defining the underlying mathematical concepts, we formalize the notion of test-based problem and discuss its properties. We also review different solution concepts, which implicitly characterize the 'target' for any algorithm that attempts to solve a test-based problem. Then we introduce coevolutionary algorithms, review the difficulties related to their design, and show how they relate to test-based problems.

2.1. Mathematical Preliminaries

In this thesis (particularly, in Chapters 5 and 6) we use some basic discrete mathematics concepts concerning partially ordered sets, which we briefly introduce in this section. The definitions below follow Trotter's book [139] unless stated otherwise.

Definition 1. A *preordered set* is a pair (X, P), where X is a set and P is a reflexive and transitive binary relation on X. Then P is a *preorder* on X.

Definition 2. A *partially ordered set* (*poset* for short) is a pair (X, P), where X is a set and P is a reflexive, antisymmetric, and transitive binary relation on X. We call X the *ground set* while P is a *partial order* on X.

We write $x \leq y$ in P when $(x, y) \in P$ and $x \geq y$ in P when $(y, x) \in P$. The notations $x < y$ in P and $y > x$ in P mean $x \leq y$ in P and $x \neq y$. When the context is obvious, we will abbreviate $x < y$ in P by just writing $x < y$.

Definition 3. For a poset (X, P), $x, y \in X$ are *comparable* ($x \perp y$) when either $x \leq y$ or $x \geq y$; otherwise, x and y are *incomparable* ($x \parallel y$).

Definition 4. A poset (X, P) is called a *chain* if every pair of elements from X is comparable. When (X, P) is a chain, we call P a *linear order* on X. Similarly, we call a poset an *antichain* if every pair of distinct elements from X is incomparable. A chain (respectively, antichain) (X', P') is a *maximum chain* (respectively, *maximum antichain*) in (X, P), $X' \subseteq X, P' \subseteq P$, if no other chain (respectively, *antichain*) in (X, P) has more elements than it.

Definition 5. An element $x \in X$ is called a *maximal element* (respectively, *minimal element*) in (X, P), if there is no $y \in X$ such that $x < y$ (respectively, $x > y$). We denote the set of all maximal elements of a poset (X, P) by $\max(X, P)$, and the set of all minimal elements by $\min(X, P)$.

Definition 6. The *width* of a poset (X, P), denoted as $\text{width}(X, P)$, is the number of elements in its maximum antichain.

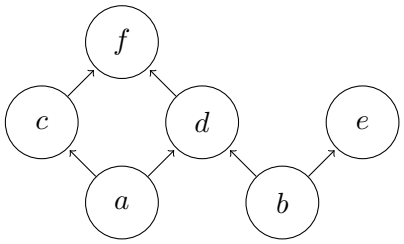
Theorem 7. (Dilworth theorem [36]) If (X, P) is a poset and $\text{width}(X, P) = n$, then there exists a partition of $X = C_1 \cup C_2 \cup \cdots \cup C_n$, where C_i is a chain for $i = 1, 2, \ldots, n$. We call it *minimum chain partition*, as it comprises the smallest possible number of chains.

Note that an important consequence of Dilworth theorem is that each C_i contains exactly one element of the maximum antichain.

Definition 8. Given a poset (X, P), a *linear extension* of P is any superset of P that is a linear order on X.

Definition 9. The *dimension* of a poset (X, P), denoted $\dim(X, P)$, is the smallest cardinal number[1] t for which there exists a family $\mathcal{R} = \{L_1, L_2, \ldots, L_t\}$ of linear extensions of P so that $P = \bigcap \mathcal{R} = \bigcap_{i=1}^{t} L_i$.

Example 10. Consider a poset (X, P) with $X = \{a, b, c, d, e, f\}$ and $P = \{(a, c), (a, d), (b, d), (b, e), (c, f), (d, f), (a, f), (b, f)\} \cup \{(x, x) : x \in X\}$ shown below in the form of the Hasse diagram.



In (X, P) some of the elements are comparable, e.g. $a \leq f$, $b \leq e$; others are incomparable, e.g., $c \parallel d$, $a \parallel e$, thus it is not a linear order. Examples of chains in (X, P) are $a < c$ and $b < d < f$. The latter is a maximum chain in (X, P). $\{a, e\}$ is an example of antichain, and $\{c, d, e\}$ is a maximum antichain of this poset, thus $\text{width}(X, P) = 3$.

[1] Here we follow the original paper by [37], since [139] requires the dimension of a poset to be a positive integer.

The set of maximal elements consists of e and f , while a and b are the minimal elements of the poset. $\{a, c, f\} \cup \{b, d\} \cup \{e\}$ is an example of a minimum chain partition. $L = a < c < b < d < e < f$ is an example of linear extension of P. $\dim(\mathrm{X}, \mathrm{P}) = 2$, because there exists a family of two linear extensions L_1 and L_2, such that $P = L_1 \cap L_2$, namely $L_1 = a < c < b < d < f < e$ and $L_2 = b < e < a < d < c < f$, and no smaller family with this property exists.

2.2. Test-Based Problems

2.2.1. Definition

We formulate an optimization problem by defining its domain, i.e., a set of candidate solutions and a function defined on the domain. In a classical optimization problem one seeks an element from the domain which maximizes (or minimizes) a given objective function. The function is usually a real one. Such a formulation allows for modeling and solving many practical tasks.

However, there exist problems for which the objective function does not exist at all or is itself so complex that computing it becomes impossible in practice. For example, consider searching for a best strategy for the first player for the game of Go, which is considered as one of the grand challenges in game related artificial intelligence [14, 11]. We could define the objective function of this game as the expected game result over games with all possible second player strategies, assuming that they are equally probable. Thus, in order to compute the objective function for a given candidate solution, one needs to play games with all possible Go strategies. Unfortunately, even for very simple games, the number of different strategies is huge. For example, the first player in the tic-tac-toe game has around $3,47 \times 10^{162}$ different strategies (see Section 5.9.4 for details) and Go is a much more complicated game than tic-tac-toe. For this reason, in practice, one is not able to compute the value of objective function for any candidate solution (a first-player strategy) for Go.

Problems for which the quality of a candidate solution is determined by its performance when interacting with a (usually large) set of tests are recently most often called test-based problems [17, 26, 29, 147, 60, 115]; other terms used in literature for similar types of problems are: coevolutionary domain [43], competitive fitness environment [3, 87], interactive domain [145, 115], adversarial problem [72], solution-test problem [54] or game problem [120]. Although authors agree about the general nature of test-based problems, they often do not define them formally. In this thesis we will use the following formal definition.

Definition 11. A *test-based problem* is a formal object $\mathcal{G} = (S, T, G, \mathcal{P}, \mathcal{P}^+)$ that consists of

- a set S of *candidate solutions* (a.k.a. candidates [17], learners [26], hosts [122] or components [115]),
- a set T of *tests* (a.k.a. teachers, parasites),
- an *interaction function* $G : S \times T \to O$, where O is a totally ordered set[2],
- a set of *potential solutions* $\mathcal{P}$ built from the set of candidate solutions, and
- a *solution concept*, which partitions the set of potential solutions $\mathcal{P}$ into *solutions* $\mathcal{P}^+$ and *non-solutions* $\mathcal{P}^-$.

To illustrate the above definition, consider the game of chess. Assume that we are seeking for the best strategy for the white player[3], so set S consists of all white player strategies while set T consists of all black player strategies. The interaction function G is then interpreted as the actual game and G's codomain O is an ordered set $\{lose < draw < win\}$. In this case, the set of potential solutions $\mathcal{P}$ may be simply the set of candidate solutions S, and one could search for a solution which maximizes games' outcomes on all tests from set T . Such a strategy exists and, as a result, $\mathcal{P}^+$ is non-empty at least.

Although the solution concept [40] univocally determines what solution one is seeking, it says very little about the relationship between two potential solutions to the problem to answer the question: none of two is the best, but which one is preffered? That is why, in order to have more fine-grained information, one may exchange the solution concept of the definition of the test-based problem to

- a *preference relation* $\preceq$ on $\mathcal{P}$, where $P_1 \preceq P_2$ is interpreted as 'P_1 not worse than P_2' with $P_1, P_2 \in \mathcal{P}$.

The relation $\preceq$ is a preorder and may be seen as a generalization of the solution concept. Indeed, the set of its maximal elements may be equal to the set of solutions $\mathcal{P}^+$. There are many possible ways to define preference relation. Let us give two examples of preference relation for chess:

[2] While O, in general, may be a partially ordered set, we are not aware of any instances of class of such problems, nor any research, in which O was only partially ordered.

[3] One may also look for just the best strategy for the game regardless of the color played; however, we use this example in order to emphasize the different roles played: white player strategies for candidate solutions and black player strategies for tests.

1. Assign scores -1 to *lose*, 0 to *draw*, and 1 to *win* in order to define an objective function $f : S \to \mathbb{R}$ as the average score received over all possible tests (black player strategies). Then we can define the preference relation in the following way: $s_1 \preceq s_2$ iff $f(s_1) \leq f(s_2)$. In this case, the preference relation determines the complete ordering of solutions.

2. The scores assigned above may seem too arbitrary, thus here is an alternative: $s_1 \preceq s_2$ iff for every test t, s_2 is not worse on t than s_1. In this case, $\preceq$ is just a preorder.

For a given pair of candidate solutions, $s_1 \preceq s_2$ may be hold or not depending on which definition is employed. Moreover, only in case of the first one, an objective function exists inducing a total order; it is, however, infeasible to compute in practice.

The researcher is not required to define a preference relation for a given problem. Referring to the terminology of optimization theory, problems with just the solution concept defined are called *co-search test-based problems*; and when a preference relation is given, we talk about *co-optimization test-based problems* [115].

2.2.2. Extensions, Terminology and Assumptions

For the purpose of this thesis, the definition of test-based problem given in this chapter is sufficiently general. However, it is interesting to note that the definition may be also elegantly generalized in two possible directions.

First, in Def. 11 we assumed that there are just two entities involved in an interaction. Consider an abstract problem in which we have a set of candidate solutions S, a set of tests T and a set of environments E, in which the interaction between candidate solutions and tests takes place. In such a case, the interaction function G has the following form $G : S \times T \times E \to O$, where elements from S play the role of candidate solutions and elements from T and E are two different types of tests. Representatives from both these kinds are required for the interaction to take place. In general, any number of different kind of test sets is possible.

Second, if more than one set of entities (e.g., sets X_1 and X_2) play the role of candidate solutions, there are no tests, and the problem solution requires an assembly or a team of elements to be evaluated, we call it a *compositional problem.*

Finally, we would like to point out that some co-optimization (or co-search) problems are neither test-based nor compositional, however, to the best of our knowledge, no research in this area has been conducted. For an in-depth study of the classification of co-search and co-optimization problems, see [117, 115].

Notice that when $T = \{t\}$, the test-based problem boils down to an a optimization problem, for which the objective function $f(s) = G(s, t)$.

When $S = T$, then such a problem will be called an *symmetrical test-based problem*. In symmetrical test-based problems, there are still two roles: candidate solutions and tests, but they are played by the same entities. The game of rock-paper-scissors is an example of such problem. When a problem is not symmetrical, we call it a *asymmetric test-based problem* [107].

Notice that in the perspective of game theory [49], G can be interpreted as a payoff matrix, and S, T as sets of game strategies. Indeed, methods proposed in this thesis are most intensively exploited in the context of games, thus, here a test-based problem is, usually, a game and we will use game-related terminology. Let us emphasize, however, that the actual interpretation of such terms like 'win' or 'player' depends on the context of particular application and may be distant from the literal meanings of these words.

For simplicity, we assume that both S and T are finite ($\mathcal{G}$ is a *finite test-based problem*), however, in Chapter 5 we will also raise the issue of test-based problems that are not finite.

In this thesis, we restrict our attention to the case where the codomain of G is a finite set, which is often the case in practice. In particular, in Chapter 3, we assume that the codomain of G is an ordered set $\{lost < draw < win\}$. Starting from Chapter 5 we will further restrict this set to a binary set $\{0 < 1\}$.

2.2.3. Non-Deterministic Test-Based Problems

Notice that Def. 11 assumes that the interaction between a candidate solution and a test is a deterministic process. It is so, indeed, in many cases; however, it may also happen that the result of the interaction is not deterministic. There are several methods to cope with such a situation.

The first method may be applied if the result of the interaction is actually deterministic, but the alleged randomness comes from a variable that we did not take into consideration. For example, when we have an artificial world and two creatures competing for food, the interaction between the two creatures may depend on the environment they were put into. In such a case, we may extend the interaction function to take the environment into consideration by treating the environment as an entity playing another test role. Then, the interaction function may have the form of $G : S \times T \times E \to O$, as we pointed out in the previous section.

When the process of interaction is random by nature, we may accept it and develop an algorithm with this in mind, or sample the interaction result and pretend it is deter-

ministic. We do the former in Chapter 3, and the latter in Chapter 4. In Chapters 5 and 6 we consider only deterministic test-based problems.

2.2.4. Solution Concepts

In the example of chess described in Section 2.2.1, we assumed that the set of potential solutions $\mathcal{P}$ to the problem equals the set of candidate solutions S and, as a result, the solutions to the problem (set $\mathcal{P}^+$) are elements of set S. However, this is not the case for all problems. For example, instead of seeking for one candidate solution that maximizes a given function, one may seek at the same time for all candidate solutions that maximize it. Another example is when the preference relation induces only a partial order on candidate solutions. In this case, the set of potential solutions $\mathcal{P}$ is a power set of S.

Although a solution concept may be defined arbitrarily, it often useful to classify test-based problems based on the type of solution concept they involve. Following [115], below we review the common classes of solution concepts. For each solution concept we also propose a *natural* preference relation for it. Although a co-search problem (with a solution concept) we may generalize to a co-optimization problem (with a preference relation) in many ways, some of preference relations seem to be, intuitively, better than others. When no obvious choice exists, we comment on this fact accordingly.

Best Worst Case In Best Worst Case $\mathcal{P} = S$, and the solution concept defines a solution as a candidate solution which does best on the hardest test. Let $v(s) = \min_{t \in T} G(s,t)$, where $s \in S$; then the solution is concept is unambiguously determined by defined the set $\mathcal{P}+$ in the following way:

$$\mathcal{P}^+ = \text{argmax}_{s \in S} v(s).$$

Analogously, we could define Worst Best Case. A natural preference relation $\preceq$ for this scenario, which induces a total order, is defined as

$$s_1 \preceq s_2 \iff v(s_1) \leq v(s_2),$$

where $s_1, s_2 \in S$.

Maximization of Expected Utility Similarly to the previous solution concept, also here, $\mathcal{P} = S$ holds. In this solution concept, one seeks a candidate solution which

maximizes the expected score against a randomly selected opponent. Thus

$$\mathcal{P}^+ = \text{argmax}_{s \in S} \text{E}[G(s,t)],$$

where E is the expectation operator and t is randomly drawn from T. A natural preference relation for this solution concept induces a total order on solutions:

$$s_1 \preceq s_2 \iff \text{E}[G(s_1,t)] \leq \text{E}[G(s_2,t)],$$

where $s_1, s_2 \in S$.

Pareto Optimal Set In the Pareto Optimal Set solution concept, every test is treated as a separate objective and the whole problem as a multi-objective optimization task. From the set $\mathcal{P} = 2^S$, we seek a Pareto-front

$$\mathcal{F} = \left\{ s \in S \mid \forall_{s' \in S} \left(\exists_{t \in T} G(s,t) \leq G(s',t) \implies G(s,t) = G(s',t) \right) \right\}.$$

The solution concept in this case contains just one solution: the Pareto-front, thus

$$\mathcal{P}^+ = \{\mathcal{F}\}.$$

The preference relation between Pareto-fronts generalizing this solution concept may be based on the dominance relation between them, e.g.,

$$F_1 \preceq F_2 \iff \forall_{t \in T} \left(\exists_{s_1 \in F_1} G(s_1,t) \implies \exists_{s_2 \in F_2} G(s_2,t) \right) \wedge |F_1| \leq |F_2|,$$

where $F_1, F_2 \in \mathcal{P}$ such that they exclusively consist of candidate solutions not strictly dominated by any other.

The second operand of this definition is required, since, according to the definition, the solution concept consists of the front containing *all* candidate solutions on the non-dominated Pareto-front. Notice, that this preference relation is very strict and many pairs of elements from $\mathcal{P}$ are incomparable, thus other preference relations are possible.

Pareto Optimal Equivalence Set This solution concept is similar to Pareto Optimal Set, since here also $\mathcal{P} = 2^S$. It may happen that the Pareto-front consists of candidate solutions that are indistinguishable according to the interaction results on the set of tests. In Pareto Optimal Set, all such candidate solutions must be included in the Pareto Optimal Set, while in the Pareto Optimal Equivalence Set only *at least* one such

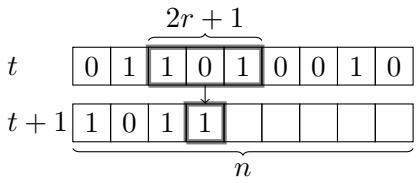


Figure 2.2.1.: A state transition in CA. Window by window the rule is applied to CA at step t in order to obtain CA at step $t + 1$.

candidate solution is required. If we further say that *exactly one* such candidate solution in Pareto-front is accepted, then we will have another solution concept termed *Pareto Optimal Minimal Equivalence Set.*

Simultaneous Maximization of All Outcomes In this scenario, again, $\mathcal{P} = S$ and one seeks a candidate solution, which is best on all test cases, i.e.,

$$\mathcal{P}^+ = \left\{ s \in S \mid \forall_{t \in T} \forall_{s' \in S} \left(G(s,t) \leq G(s',t) \implies G(s,t) = G(s',t) \right) \right\}.$$

For a given problem, the set of solutions $\mathcal{P}^+$ may be empty, thus it is unclear what a preference relation generalizing this solution concept should look like, in general.

2.2.5. Examples of Test-Based Problems

Many practical problems may be modeled as test-based problems, including those which traditionally were modeled as optimization problems. In the previous sections we analyzed on examples of games. Here, in order to show that games are not the only instances of test-based problems, we gave several other examples.

Problem 12. Density Classification Task

The density classification task (a.k.a. majority problem) is a problem in cellular automata (CA) research [48, 98]. Given a one-dimensional binary CA of size n, the objective is to find a state transition rule that makes the CA converge from an initial configuration (IC) to a required state within a given number of steps t. The required state is all zeros if IC contains more zeros than ones, and all ones if IC contains more ones than zeros (n is assumed to be odd to prevent ties, which would render the required behavior undefined). Given a radius r, a rule determines the value of the i^{th} bit of CA state based on the states of bits $[i - r, r + i]$ (cyclic) and can be represented as a lookup table with 2^{2r+1} entries describing its behavior in all possible states. See Figure 2.2.1 for an illustration.

We formulate this task as a test-based problem in the following way. The set of candidate solutions S contains all $2^{2^{2r+1}}$ possible rules, the set of tests T comprises all possible 2^n ICs, and the interaction function is defined as

$$G(s,t) = \begin{cases} 1 & \text{if } s \text{ correctly classifies } t; \\ 0 & \text{otherwise.} \end{cases}$$

The original task formulates the goal as finding the rule that correctly classifies the maximal number of ICs (Maximization of Expected Utility solution concept). Although it has been proved that for sufficiently large n, the rule correctly classifying all ICs does not exist [81], the maximal possible rule performance is unknown. For the most popular settings, i.e., $n = 149$, $r = 3$ and $t = 320$, the best known rule has been found using coevolutionary algorithms and has a success rate of 86.3% [69, 70].

Problem 13. Symbolic Regression

In the symbolic regression problem [95] one searches for a symbolic expression of a function that fits best to a given set of points P. If modeled as a test-based problem, the set of candidate solutions S consists of all functions of allowed form, while the set of tests T may comprise all points from P (or all subsets of P). The value of the interaction function G may be, for example, a square root error. Maximization of Expected Utility is one candidate of a right solution concept for this problem.

Problem 14. Optimization of Algorithm Parameters

Let H be a heuristic, S a set of all vectors of parameters for an optimization problem P and T a set of instances of problem P. What is the best, on average, vector of parameters from S over all instances from set T?

Other examples of the test-based problem, considered in the past, include intertwined spiral problem [67], neural network design [84] and sorting network design [54].

One may have noticed that all above defined problems involve the Maximization of Expected Utility solution concept. Indeed, Maximization of Expected Utility is a wise choice since a practitioner is usually looking for a solution in the form of one self-contained entity (e.g., game strategy, algorithm, creature, function) instead of a solution in a form of a large set of (e.g., non-dominated) entities. All research involving the Pareto-based solution concepts we are aware of concern only theoretic results or results on artificial problems. This may be caused by a practically infeasible size of the set of Pareto non-dominated front, similarly to the one spotted in the class of many-objective optimization problems [57].

Having that in mind, Pareto dominance may also play a role in problems involving the Maximization of Expected Utility solution concept, as all solution concepts introduced here respect the Pareto dominance relation between candidate solutions, i.e., no Pareto dominated candidate solution is ever element of the solution of the problem or itself the solution of the problem. This opens doors for treating the Pareto dominance between candidate solutions as a concept common to most of test-based problems, which is used in the *Pareto-coevolution* line of research (also in Chapter 5).

2.3. Solving Test-Based Problems using Coevolutionary Algorithms

2.3.1. Coevolutionary Algorithms

Coevolutionary algorithms are variants of evolutionary computation in which an individual's fitness depends on other individuals. An individual's evaluation takes place in the context of at least one other individual, and may be of *cooperative* or *competitive* nature. In the former case, individuals share benefits of the fitness they have jointly elaborated, and such algorithms are well suited for compositional problems, whereas in the latter one, a gain for one individual means a loss for the other, which fits test-based problems. Since in this thesis we concentrate on test-based problems, in the following, by 'coevolutionary algorithm' we will mean its competitive form.

A coevolutionary algorithm is similar to evolutionary one [7] because both families of methods maintain a number of individuals and use mechanisms that mimic the natural evolution, that is, selection and variation operators such as mutation and crossover. The driving force of coevolutionary algorithms is the continuous Darwinian *arms race* taking place between (usually) two competing populations [104]. The difference between coevolutionary and evolutionary methods lies in the evaluation phase, when the fitness of individuals is assessed. Evolutionary algorithms that solve optimization problems have access to the objective function of a problem, thus individuals' fitness is directly computed. In coevolutionary methods, individuals' fitness is typically only estimated by aggregating results of multiple interactions of individuals from one population with individuals from the other. Thus, in coevolution, the interaction between individuals substitutes the objective external fitness measure present in evolutionary algorithms. Various methods of evaluation in coevolution will be presented in Chapter 3.

Coevolutionary algorithms are natural methods for solving test-based problems due to several reasons. First, they do not require an objective fitness function, but only interac-

tions between pairs of individuals. Second, they are naturally suited to involve multiple populations, appropriately to the roles present in a test-based problem (candidate solutions and tests). Finally, they are general. While specialized methods for some classes of test-based problems exist (e.g., for machine learning and game-playing), coevolutionary algorithms, as a general metaheuristics, may work for any test-based problem, provided evolutionary operators involved are properly adapted to given individual representation.

2.3.2. Applications of Coevolutionary Algorithms

Coevolutionary algorithms have been applied to many disciplines and problems. First experiments with coevolution-like self-learning methods were performed by Samuel in 1959 on the problem of learning strategies of the game of checkers [125]. However, the discipline developed rapidly and grew in popularity much later, in early 1990s, after the work by Axelrod on iterated prisoner's dilemma [4], Hillis'es experiments on coevolving sorting networks with number arrays to be sorted [54], and Sim's work on coevolving competing virtual creatures [129].

Probably the most popular applications of coevolutionary methods are games, which is not surprising since they have always been a popular test-bed for artificial and computational intelligence [85]. Reynolds's experiments with coevolution for the game of tag [119] is a classical example of such attempts. One of the most successful approaches to games is Blondie 24, a master-level checker player coevolved without explicit encoding of human game expertise by Chellapilla and Fogel [21, 45, 20]. Recently, Fogel and colleagues used a similar method to coevolve a neural network chess player, Blondie 25, which beat Pocket Fritz 2.0 demonstrating a performance rank of 2650 [46] and won a game with Fritz 8.0 [47], one of the strongest chess programs. Some research shows the potential of coevolutionary algorithms or hybridized coevolutionary algorithms for Othello [22, 86, 136, 93, 94] or for a small-board version of Go [84, 123, 80]; however, particularly for the latter game, coevolutionary methods were unable to produce a player exhibiting a master level of play. Coevolutionary algorithms were also applied to non-deterministic games such as backgammon [114, 5, 130], Texas holdem poker [102], Blackjack [19], and robotic soccer [89, 87]. Other game applications include the pong game [100], nim [73], pursuit-evasion problems [97, 137], and real-time neuroevolution in the NERO game [134].

Applications of coevolutionary algorithms are not limited to games. Paredis had coevolved neural network classifiers [110] with life-time learning, and then used a similar technique to solve constraint satisfaction problems [109, 111]. Other interesting applications of coevolutionary algorithms include the traveling salesman problem [135], cellular

automata density classification task [70, 105, 112], Boolean network design [131], intertwined spirals problem [67], job shop scheduling [56, 52, 143], and protein sequence classification [106]. Recently, the coevolutionary methods have been used in the domain of multi-objective optimization [50].

2.3.3. Coevolutionary Pathologies

Coevolutionary algorithms exhibit complex, hard to understand dynamics [68, 116] and are prone to *coevolutionary pathologies*, which hamper the progress of search process and are a major challenge in coevolutionary algorithms design [42]. Examples of such undesired phenomena include:

- *Collusion* possibility, noticed by Blair and Pollack [9] in competitive environments. Collusion may be observed on the level of *meta-game of learning* [114] as a tendency to converge to suboptimal equilibria. It has been demonstrated that while in the competitive environment the most obvious way for a candidate solution to increase its chance of survival is to improve its ability to perform certain task ("performing"), there is another way called "infiltrating". Infiltrators are good at passing strategic attributes or genes to restrict the performers' ability to perform their task by, for instance, guiding the search process towards a region of a space where a local minimum is located. As a result, collusion can seriously stifle coevolutionary learning.

- *Red Queen effect* (a.k.a. *Red Queen dynamics* or *mediocre stable states*) [112] is related to collusion [16]. The test-based problems are inherently intransitive [104, 27], thus it can happen that coevolving parties increase their fitness with respect to each other, but in absolute terms they do not improve (see also *cycling* [140]).

- *Disengagement* [18, 10] occurs when an arms race is no longer possible due to *loss of gradient*. For example, it may happen that all maintained tests are solved by all candidate solutions; in effect, tests could not effectively discriminate candidate solutions, and the other way round.

- *Forgetting* [41] is an effect of other pathologies, such as those described above, and it manifests itself by a fact that certain traits which are being acquired during the coevolutionary search later disappear from population because of lack of selective pressure on them.

- *Over-specialization* or *focusing* [140] can be observed when a coevolutionary algorithm tends to produce solutions that focus on some aspects of the problem while ignoring the other ones. An example of this pathology can be a chess player who demonstrates a master level of play at opening and middlegame but is novice at endgame. Over-specialization can be caused by collusion.

Pathologies are detrimental for coevolutionary algorithms because they distract them from the task the algorithms are supposed to perform.

2.3.4. Coevolutionary Archives

The goal of *coevolutionary archives* is to sustain progress during a coevolutionary run, i.e., to counteract some of the coevolutionary pathologies. A typical archive is a (usually limited in size, yet diversified) sample of well-performing individuals found so far. New individuals submitted to the archive are forced to interact with its members, who may be replaced when no more useful. Archives in coevolution may be seen as a counterpart of elitism in standard evolution. Historically, one of the oldest and simplest archives was Hall of Fame [120] that stores all the best-of-generation individuals encountered so far. Modern examples of archives include DECA [32], EPCA [144], Nash Memory [44], DELPHI [34], IPCA [26], and LAPCA [28]. The last two will be described in Sections 6.3.1 and 6.3.2.

If an archive is present, we may talk about *generator-archive scheme* [30] for solving test-based problems by coevolutionary methods. The role of generator may be played by any algorithm that is able to produce tests and candidate solutions, no matter what are the internal principles it employs to perform this task.

A generator-archive scheme for coevolutionary algorithms solving a test-based problem is presented in Alg. 2.1. The listing consists of the initialization part (lines 2-4) and the main loop (lines 5-13). In the loop, first the new candidate solutions and tests are generated, typically via mutating or crossing over individuals from the current populations S' and T' (lines 6 and 7). In lines 8 and 9 the current populations are temporarily expanded with the new elements. After all individuals are submitted to the archive, which usually accepts only some of them, both populations are subject to evaluation and selection. Notice that the archive, apart from being updated in line 10, may also be used in lines 6 and 7 in order to support generating new good individuals.

While most authors recently concentrate on the archive component of the generator-archive scheme, we have to note that coevolutionary archive is not desired in all cases. The problem is that adding a candidate solution or a test to an archive requires many

Algorithm 2.1 Generator-archive scheme for solving test-based problems by coevolutionary algorithms.

```
1:  procedure Coevolution
2:      S', T' ← Initialize populations
3:      S_arch ← ∅
4:      T_arch ← ∅
5:      while ¬stopped do
6:          S_new ← GenerateNewSolutions(S', S_arch)
7:          T_new ← GenerateNewTests(T', T_arch)
8:          S' ← S' ∪ S_new
9:          T' ← T' ∪ T_new
10:         Archive.Submit(S', T')              ▷ Updates S_arch and T_arch
11:         Evaluate(S', T')
12:         S', T' ← Select(S', T')
13:     end while
14: end procedure
```

interactions (Hall of Fame archive is an exception here). In modern archives such as DECA, IPCA, LAPCA or COSA, where the archive consists of a set of tests T_{arch} and a set of candidate solutions S_{arch}, checking if a new test should be added to (accepted by) the archive requires $|T_{arch}|$ additional interactions, and checking if a new candidate solution should be added to the achieve requires additional $|S_{arch}|$ interactions. Thus, adding new elements to the archive requires additional computational power and may become costly with the growth of the archive. That is why some theoretical properties of archives, which, e.g., guarantee progress towards a given solution concept, are often traded for a reasonably small size of the archive (e.g., in MaxSolve). Because of this trade-off, and despite the fact that some research shows that using archives is profitable (e.g., [136]), using archives for practical applications may be disputable and conventional non-archive coevolutionary algorithms may still be better or not worse at least [100].

2.4. Discussion

There is a clear intersection between test-based problems and optimization problems. In particular, when the set of tests consists of only one element, the problem may be handled with classical optimization techniques. From a practical perspective, we may, informally, distinguish three categories of test-based problems according to the following distinction:

1. **The size of T is small**. In such a case, there is no need to approach the problem

with coevolutionary algorithms and, as no arms races could be achieved, no clear benefits from using such methods are expected. The exact objective performance of a solution can be calculated as an outcome of its interactions with all tests in T, so such problems can be usually handled by evolutionary algorithms or any other variant of heuristic search (not mentioning exact search methods like, e.g., branch-and-bound). Although some coevolutionary methods for classical function optimization problems were proposed [126], motivation for them is disputable and evidence that they may give any advances over optimization methods is unconvincing.

2. **The size of T is moderate**. For some of problems in this category, coevolutionary approaches may provide better results than optimization methods [111], however it is still possible (i.e., computationally viable) to use (single- or multi-criteria) optimization methods and to evaluate a candidate solution on all tests from T.

3. **The size of T is large.** For some problems in this category, it is still possible to use optimization methods by (randomly) sampling set T (e.g., in density classification task), or using other more problem-specific methods (e.g., in games). In general, however, this is the class of problems, in which the application of coevolutionary methods has the greatest potential [3].

We should emphasize, however, that the above classification should be taken with a grain of salt, as its practical usage requires taking into account the actual cost of interaction for a given problem. For problems with high cost of interaction (e.g., consisting in simulating robots or players in a virtual environment), handling even a small number of tests can be challenging, making the use of coevolutionary methods justified.

3. Fitnessless Coevolution

In this chapter, we present a one-population coevolutionary method for solving symmetrical test-based problems, called *Fitnessless Coevolution.* Fitnessless Coevolution is an example of a conventional coevolutionary methods, which does not use archives to sustain progress, thus it is prone to coevolutionary pathologies. On the other hand Fitnessless Coevolution has one unique feature: under certain assumptions it behaves as a standard evolutionary algorithm. As a result, it may sustain progress *per se.*

Fitnessless Coevolution assumes that the codomain of the interaction function G is a set $\{lose < draw < win\}$. That is why, in this chapter, we will make use of game-related terminology.

3.1. Introduction

In biology, coevolution typically refers to an interaction between two or more species. By analogy, in evolutionary computation coevolution usually implies using multiple populations. The main reason for having more than one population is the inherent asymmetry of most test-based problems, in which candidate solutions are fundamentally different from tests (e.g., they use different representation). Thus, in general, in order to solve a test-based problem, coevolution evolves at least two populations containing candidate solutions and tests [15], respectively.

However, for symmetrical test-based problems, one may use *one-population coevolution* [91], in which, unless some archiving mechanism is used, the individuals in the current population are the only data available to enforce the selection pressure on the evolutionary process.

One of the most important decisions to make when designing a coevolutionary algorithm is the choice of evaluation method. When no coevolutionary archive is present, the evaluation must rely on the current population only. Several such evaluation methods were designed. One of them is the *round-robin tournament* (a.k.a *complete mixing* [90]) that involves *all* the individuals from the population and defines fitness as the average payoff of the played games. A round-robin tournament requires $n(n-1)/2$ games to

be played in each generation, where n is the population size; therefore, it is computationally demanding even for a moderately sized population. As a remedy, Angeline and Pollack [3] proposed the *single-elimination tournament* that requires only $n-1$ games. Starting from the entire population, the players/individuals are paired, play a game, and the winners pass to the next round. The last round produces the final winner of the tournament, and the fitness of each individual is the number of games it won. Finally, the *k-random opponents* method [119] lets an individual play with k opponents drawn at random from the current population and defines fitness as the average payoff of games played, requiring kn games to be played per generation. This evaluation scheme was applied, for instance, by Fogel to evolve neural nets that play checkers [45].

All the aforementioned methods follow the evaluation-selection-recombination mantra. Games played in the evaluation phase determine individuals' fitnesses that are subsequently used in the selection phase. Obvious as it seems, this scheme is essentially redundant. Playing games is selective by nature, so why not use them directly for selection?

This observation led us to propose an approach called *Fitnessless Coevolution*. Fitnessless Coevolution uses game outcomes to settle tournaments in the selection phase, skipping therefore the evaluation. Technically, we skip the evaluation and proceed directly to selection, which works like a single-elimination tournament played between k individuals randomly drawn from the population. The winner of this tournament becomes the outcome of the selection process. The only factor that determines the winner is the specific sequence of wins and losses, so that no explicit fitness measure is involved in this process. As a result, Fitnessless Coevolution does not require any aggregation of results of interaction into a scalar fitness value. This makes our approach significantly different from most of the methods presented in literature. The only related contribution known to us is [138], where Tettamanzi describes *competitive selection* — a form of stochastic binary tournament selection.

In the experimental part of this chapter, we demonstrate that, despite being conceptually simpler than standard fitness-based coevolution, Fitnessless Coevolution is able to produce excellent players without an externally provided yardstick, like a human-made strategy. We also present a theoretical result: provided the payoff matrix of the game induces the same linear order of individuals as the fitness function, the dynamics of the Fitnessless Coevolution is identical to that of a traditional evolutionary algorithm. This makes it possible to study Fitnessless Coevolution using the same research apparatus as for the standard evolutionary methods.

3.2. Fitnessless Coevolution

In the traditional evolutionary algorithm, all individuals are tested in the environment and receive an objective fitness value during the evaluation phase. Afterwards, the fitness values are used in the selection phase in order to breed the new generation. In the single-population coevolutionary algorithm, there is no objective fitness function, and individuals have to be compared (pairwise or in larger groups) to state which one is better. Despite this fact, the scheme of a coevolutionary algorithm is similar to the evolutionary one. Typically, an individual receives a numerical fitness value that is based on the results of games played with some other individuals. Then, the selection procedure follows, most commonly a tournament selection that takes into account only the *ordering* of individuals' fitnesses, not their specific *values*. Thus, the outcomes of the games (relations *per se*) are converted into numerical fitness values which in turn determine the relations between individuals in the selection process. In this light, assigning fitness values to individuals seems redundant, because, in the end, only relations between them matter. Nonetheless, this is the common proceeding used in past work [3, 108, 45], except for the preliminary considerations in [138].

The redundancy of the explicit numerical fitness in one-population coevolution inspired us to get rid of it in an approach termed *Fitnessless Coevolution*. In fitnessless coevolution, there is no explicit evaluation phase, and the selection pressure is implemented in the *fitnessless selection*. Fitnessless selection may be considered a variant of a single-elimination tournament applied to a randomly drawn set K of individuals of size k, which is the only parameter of the method. The selection process advances in rounds. In each round, individuals from K are paired, play a game, and the winners pass to the next round (compare description of the single-elimination tournament in Section 3.1). For odd-sized tournaments, the odd individual plays a game with one of the winners of the round. In case of a game ending with a draw, the game winner is selected at random. This process continues until the last round produces the final winner of the tournament, which becomes also the result of selection. In particular, for $k = 2$, the winner of the only game is selected. The fitnessless selection operator is applied n times to produce the new population of size n, so the total number of games per generation amounts to a reasonable $(k-1)n$.

It should be emphasized that the term 'fitnessless' is not meant to suggest the absence of selection pressure in Fitnessless Coevolution. The selection pressure emerges as a side-effect of interactions between individuals, but is not expressed by explicit fitness function.

3.3. Equivalence to Evolutionary Algorithms

Fitnessless Coevolution, as any type of coevolution, makes investigation of the dynamics of the evolutionary process difficult. Without an objective fitness, individuals stand on each others shoulders rather than climb a single 'Mount Improbable'. In particular, notice that if the game is intransitive (beating a player P does not imply the ability of beating all those beaten by P), the winner of fitnessless selection does not have to be superior to all tournament participants. To cope with problems like that, Luke and Wiegand [91] defined the conditions that a single-population coevolutionary algorithm must fulfill to be *dynamically equivalent* to an evolutionary algorithm, i.e., to produce the same run, including the same contents of all generations. In the following, we first shortly summarize their work, then we determine when our Fitnessless Coevolution approach is dynamically equivalent to evolutionary algorithm and comment on how our result compare with Luke and Wiegand's.

Following [91], we define the payoff matrix and utility.

Definition 15. $G = [g_{ij}]$ is a *payoff matrix*, in which g_{ij} specifies the score awarded to strategy $\#i$ when playing against strategy $\#j$.

Definition 16. Assuming an infinite population size and complete mixing (i.e., each individual is paired with every other individual in the population including itself), aggregate subjective values for genotypes (their utility) can be obtained as follows:

$$\overrightarrow{u} = G\overrightarrow{x},$$

where $\vec{x}$ represents proportions of genotypes in an infinite population.

Definition 17. Given a linear transformation $g_{ij} = \alpha f_i + \beta f_j + \gamma$, the internal subjective utility u is linearly related to an objective function f, $u \sim_L f$, if the transitive payoff matrix G is produced using this transformation.

Luke and Wiegand proved the following theorem, which says when a single-population coevolutionary algorithm exhibits evolutionary dynamics.

Theorem 18. *A single-population coevolutionary algorithm under complete mixing and the assumption that population sizes are infinite employing a non-parametric selection method using the internal subjective utility $\overrightarrow{u} = Gx$ is dynamically equivalent to an evolutionary algorithm with the same selection method, using the objective function f, if $u \sim_L f$ as long as $\alpha > 0$ [91].*

In order to guarantee this dynamic equivalence, Luke and Wiegand had to make several assumptions about the evolutionary algorithm and the payoff matrix G: infinite populations, complete mixing, and $u \sim_L f$. In the following, we prove that Fitnessless Coevolution is equivalent to an evolutionary algorithm employing tournament selection under the only condition that f has to induce the same linear order of individuals as the payoff matrix G.

Theorem 19. *A single-population coevolutionary algorithm employing fitnessless selection (i.e., Fitnessless Coevolution) is dynamically equivalent to an evolutionary algorithm with tournament selection using the objective function f, if*

$$\forall_{i,j} f_i > f_j \iff g_{ij} > g_{ji}. \tag{3.3.1}$$

Proof. We need to show that, given (3.3.1), for any set of individuals Q, each act of selection out of S based on f in the evolutionary algorithm produces the same individual as the fitnessless selection applied to the same set Q. Let us assume, without loss of generality, that f is being maximized. As for an arithmetic objective function f,

$$f_i \geq f_j \wedge f_j \geq f_k \Rightarrow f_i \geq f_k,$$

it is easy to show that, under (3.3.1), a similar expression must be true for G:

$$g_{ij} \geq g_{ji} \wedge g_{jk} \geq g_{kj} \Rightarrow g_{ik} \geq g_{ki}.$$

In the Fitnessless Coevolution, the outcome of selection is the winner of the last game of a single-elimination tournament; let w be the index of that individual. The winner's important property is that it won or drew all games that it played in the tournament; since the payoff matrix G is transitive, the winner is in fact superior to *all* individuals in Q. Therefore, $\forall_{i \in S}\, g_{wi} \geq g_{iw}$, and this, together with (3.3.1), implies that $\forall_{i \in S}\, f_w \geq f_i$. Thus, the winner of fitnessless selection has the maximal objective fitness among the individuals in Q and would also win the tournament selection in the traditional evolutionary algorithm. As a result, under (3.3.1), both selection methods produce the same individual, and the course of both algorithms is identical. □

The consequence of the above condition is following. If the payoff matrix A is transitive, there always exists an objective function f, so that the evolutionary algorithm using f as a fitness function is dynamically equivalent to Fitnessless Coevolution using G. Thus, we refer to condition (3.3.1) as to *transitivity condition.*

Note that Fitnessless Coevolution does not need to know f explicitly. To make it behave as a standard evolutionary algorithm, it is enough to know that such objective f exists. One can argue that if there exists such a function f that the transitivity condition holds, it would be better to construct it explicitly, and run a traditional evolutionary algorithm using f as a fitness function, instead of running the Fitnessless Coevolution. One could even avoid the explicit function f and sort the entire population using the game outcomes as a sorting criterion (comparator), and then apply a non-parametric selection (like tournament selection) using that order. In both cases, however, fulfilling condition (3.3.1) is the necessary prerequisite. As we will show in the following experiment, Fitnessless Coevolution performs well even if it does not hold. Besides, even if f exists, it may be hard to define it explicitly.

We also claim that, where possible, one should get rid of numerical fitness because of Occam's razor principle: if it is superfluous, why use it? Note also that numerical fitness may be accidentally over-interpreted by attributing to it more meaning than it actually has. For instance, one could evaluate individuals using a single-elimination tournament, which produces fitness defined on an ordinal scale, and then apply a fitness-proportional selection. As the fitness-proportional selection assumes that the fitness is defined on the metric scale, its outcomes would be flawed.

3.4. Experiments

In order to assess the effectiveness of our Fitnessless Coevolution with fitnessless selection (FLS), we compared it to the fitness-based coevolution with two selection methods: single-elimination tournament (SET) and k-random opponents (kRO). In total, we considered twelve setups (FLS, SET, and kRO for $k = 1, ..., 10$), called *architectures* in the following.

We apply each architecture to four test-based problems. This includes two games: the tic-tac-toe (a.k.a. noughts and crosses) and a variant of the Nim game. As demonstrated in the following, both of them are intransitive so no objective fitness function exists that linearly orders their strategies. Following [108], we also apply the architectures to standard optimization benchmarks of minimizing Rosenbrock and Rastrigin functions, by casting them into symmetrical test-based problems. Of course, for this kind of task the objective fitness exists by definition (it is the function value itself) and the game is transitive. Normally, this kind of task is solved using an ordinary fitness-based evolutionary algorithm, but casting this problem into the test-based problem domain serves here the purpose of exploring the dynamics of the fitnessless one-population coevolution.

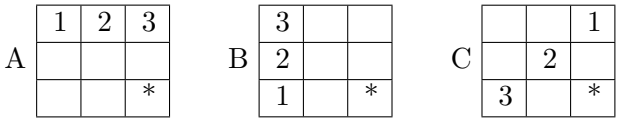


Figure 3.4.1.: Three simple tic-tac-toe strategies that violate condition (3.3.1).

Otherwise, as shown below for tic-tac-toe and Nim, no such problem casting is needed to apply the Fitnessless Coevolution to any two-player game.

Instead of designing our own genetic encoding, we followed the experimental setups from [3] (tic-tac-toe) and [108] (the rest). All three reference architectures used tournament selection of size 2. Note that we did not limit the number of generations; rather than that, each evolutionary run stops after reaching the total of 100,000 of interactions (games played). It is a fair approach, as some selection methods need more interactions per generation than the others, and time of an interaction is usually the core component of computational cost. We performed 50 independent runs for each architecture to obtain statistically significant results.

Our experiments were implemented with ECJ framework [88].

3.4.1. Tic-Tac-Toe

In this well-known game, two players take turns to mark the fields in a 3x3 grid with two markers. The player who succeeds in placing three marks in line wins the game.

Tic-tac-toe does not fulfill the transitivity condition (3.3.1), which is easy to demonstrate by an example. Let us consider a triple of trivial strategies A, B, C, shown in Fig. 3.4.1. Each of them consists in placing the marks in locations and in an order shown by the numbers when the grid cell is free, or placing the mark in the asterisk cell if the numbered cell is already occupied by the opponent. Clearly, no matter who makes the first move, strategy A beats B, as already its first move prevents B from having three marks in the leftmost column. By the same principle, B wins with C. According to transitivity condition, these two facts require the existence of f_A, f_B, f_C such that $f_A > f_B$ and $f_B > f_C$. This, in turn, implies $f_A > f_C$. However, Fig. 3.4.1 clearly shows that C beats A, which contradicts $f_A > f_C$. There is a cycle: none of these strategies outperforms the two remaining ones and their utilities cannot be mapped onto an ordinal (or numerical, in particular) scale.

Each individual-player in this experiment has the form of a single genetic programming tree (GP, [75]), built using a function set of nine terminals and seven functions. The terminals represent the nine positions on the board (pos00, pos01, ..., pos22). All func-

tions process and return board positions or a special value *NIL*. The binary function *And* returns the second argument if neither argument is *NIL*, and *NIL* otherwise. *Or* returns the first not-*NIL* argument or *NIL* value if both are *NIL*. *If* returns the value returned by the second argument if the first (conditional) argument is not *NIL*, otherwise the third argument. The *Mine*, *Your*s and *Open* operators test the state of a given field. They take a board position as an argument and return it if it has an appropriate state, otherwise they return *NIL*. The one-argument operator *Play-at* places player's mark on the position given by the argument if the field is empty and, importantly, stops the processing of the GP tree. If the field is taken, *Play-at* returns its argument and the processing continues.

As this function set does not guarantee making *any* move, we promote players which make some moves. Player's final score is, therefore, the number of moves made plus an additional 5 points bonus for a draw or 20 points for winning. The player with more points wins. As the game in its original form is not a symmetrical test-based problem, we let the players play *double-games*. A double-game consists of a pair of games, each starting with a different player. The player that wins both games in the double-game is declared the winner, otherwise there is a draw.

This experiment used maximal tree depth of 15, population size 256, crossover probability of 0.9, and replication probability of 0.1.

Following [108], we determined the best-of-run solution by running the SET on all best-of-generation[1] individuals. After carrying out 50 independent runs, we got 50 representative individuals from each architecture. To compare our twelve architectures, we let all $12 \times 50 = 600$ representative individuals play a round-robin tournament. The final evaluation of each individual was the average score against the other individuals in the tournament. The mean of these evaluations was the final architecture's score.

3.4.2. Nim Game

In general, the game of Nim involves several piles of stones. Following [108], we used only one pile of 200 stones. Players take in turn one, two, or three stones. The player who takes the last stone wins.

The Nim game individual is encoded as a linear genome of 199 bits (the 200^{th} bit is not needed because 200 stones is the initial state). The i^{th} gene (bit) says whether i stones in the pile is a *desirable game state* for the player (value 1) or not (value 0). A player can take one, two, or three stones in its turn. It takes three stones if it leads

[1] In fitnessless approach appointing the best-of-generation individual is not obvious, so we simply choose it randomly.

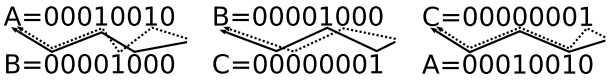


Figure 3.4.2.: Three games played between three players *A, B*, and *C* demonstrate Nim's intransitivity. The bitstrings encode the strategies of the players. The dotted line shows the advancement of the game when the upper player makes the first move. The solid line shows the advancement of the game when the lower player makes the first move. The upper players win in all three cases, no matter who makes the first move, so none of the strategies is better that the remaining two.

to a desirable game state (i.e., if the corresponding bit is 1). Then, it tests in the same way taking two and one stone. If all considered states are not desirable, the player takes three stones.

The outcome of the Nim game may depend on who moves first. For instance, let us consider a simplified Nim starting with just 7 stones and two strategies encoded in the way discussed above: $A = 001000$ (meaning three stones is a desirable state, while 1, 2, 4, 5, and 6 stones are not) and $B = 001001$ (only 3 and 6 stones are desirable). If A moves first, it takes 3 stones (as all three considered genes are 0), then B takes 1 stone (according to the third bit in its strategy), and finally A takes the last three stones and wins. However, if B moves first, it takes only one stone (due to the rightmost '1' in its genotype), A takes three stones, thus B is left with three stones to be taken and wins. Due to this property of Nim, in order to have a symmetrical test-based problem, we make our individuals face each other in a double-game, similarly to tic-tac-toe.

Despite its simplicity, Nim is intransitive too. Let us consider three 9-stone Nim strategies $A = 00010010$, $B = 00001000$, and $C = 00000001$ (as it turns out, nine stones is the minimum number required to demonstrate intransitivity). The double-game between A and B results in A's win (see Fig. 3.4.2). Thus, according to condition (3.3.1), A should have better fitness than B: $f_A > f_B$. As B beats C, also $f_B > f_C$ should hold. However, C wins against A, requiring $f_C > f_A$. No numerical (or even ordinal) fitness can model the mutual relationships between A, B, and C.

Our experiments involved population size of 128, a 1-point crossover with probability 0.97, and mutation with probability 0.03. The architectures were compared in the same way as in tic-tac-toe.

3.4.3. Rosenbrock

The Rosenbrock function has the following form for the N-dimensional case:

$$Rosenbrock(X) = \sum_{i=1}^{N-1} \left[(1 - x_i)^2 + 100 \left(x_{i+1} - x_i^2 \right)^2 \right].$$

We converted the problem of minimizing this function to a competitive counterpart by defining an interaction function

$$G(s,t) = \frac{Rosenbrock(s) - Rosenbrock(t)}{\max(Rosenbrock) - \min(Rosenbrock)},$$

where $\max(Rosenbrock)$ and $\min(Rosenbrock)$ are the maximum and minimum values of Rosenbrock function in the considered domain; $G(s,t)$ determines the result of an interaction (in the range $[-1,1]$) between player s and its opponent t. In this symmetrical test-based problem, $G(s,t) = -G(t,s)$.

In this experiment, we used genomes of $N = 100$ real values between -5.12 and 5.12 (function domain), population size of 32, a 1-point crossover, and mutation of a single gene with probability 0.005.

In the Rosenbrock problem, unlike in tic-tac-toe and Nim games, there exists an objective and external (i.e., not used during the evolution) individual's fitness — the Rosenbrock function itself. Therefore, as the best-of-run we chose the individual that maximizes the external fitness value, defined as

$$1 - \frac{Rosenbrock(X) - min(Rosenbrock)}{max(Rosenbrock) - min(Rosenbrock)}. \tag{3.4.1}$$

For the same reason, in the Rosenbrock problem, to compare the architectures we also used this external fitness. It should be emphasized, however, that the fitnessless run has no access to the external fitness function, which is used only for the purpose of the best-of-run selection and comparison of best-of-runs between particular runs.

3.4.4. Rastrigin

As the last problem, we considered minimizing the Rastrigin function, defined as:

$$Rastrigin(X) = A \cdot N + \sum_{i=1}^{N} \left[x_i^2 - A \cdot \cos(2\pi x_i) \right],$$

	Tic-tac-toe				Nim				Rosenbrock				Rastrigin			
FLS vs.	kRO			SET	kRO			SET	kRO			SET	kRO			SET
Noise	<	=	>		<	=	>		<	=	>		<	=	>	
0%		2	8	=			10	<			10	>			10	>
30%		10		=	2	5	3	=			10	>		4	6	>
40%	4	6		>	3	6	1	=	6		4	>	6	1	3	>
Total	4	18	8		5	11	14		6		24		6	5	19	

Table 3.1.: The outcomes of pairwise statistical comparison of FLS vs. kRO and SET (significance level 0.01). Symbols <, =, and > denote respectively FLS being worse, equally good, and better than the other method. For kRO, figures tell how many times FLS was in particular relation to kRO.

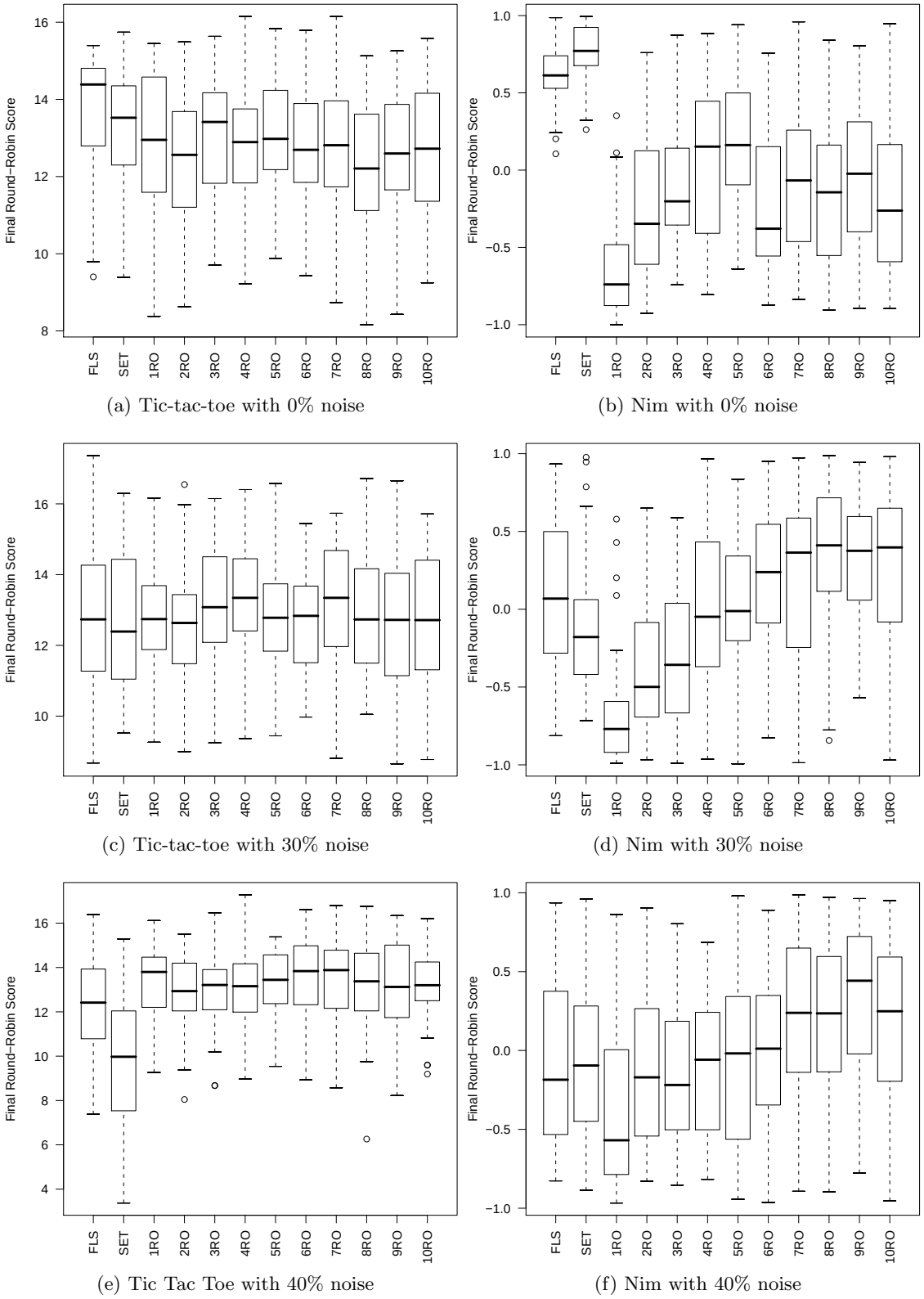
where $A = 10$ and $N = 100$. The Rastrigin minimization problem was converted to a test-based problem in the same way as the Rosenbrock function. Also, the setup of the experiment and comparison between architectures was identical to Rosenbrock's.

3.5. Results

Figures 3.5.1 and 3.5.2 compare the architectures of FLS, SET and kRO for k ranging from 1 to 10. These charts present the average external fitness of the best-of-run individuals from each architecture.

As we can see in Fig. 3.5.1a, FLS was not much better than the other architectures at playing tic-tac-toe and slightly worse than SET at evolving the Nim player (Fig. 3.5.1b). On the other hand, in problems that fulfill the transitivity condition (Fig. 3.5.2a and 3.5.2b), the FLS architecture was clearly better than SET and kRO, which is especially visible in case of Rastrigin function. More precisely, FLS is statistically better than kRO for all values of k on Nim, Rosenbrock, and Rastrigin; for tic-tac-toe, it beats kRO for 8 out of 10 values of k (t-Student, $p = 0.01$). Also, FLS is significantly better than SET on Rosenbrock and Rastrigin and worse on Nim; for tic-tac-toe, the test is inconclusive. Table 3.1 summarizes the outcomes of the statistical comparison of FLS to kRO and SET.

Following [90], we also tested how the noisy data influences evolution. We introduced noise by reversing the game outcome (thus swapping players' rewards) with a given probability. For instance, adding 100% noise would aim at evolving the worst possible player. Figures 3.5.1, 3.5.2, and Table 3.1 show the effect of adding 30% and 40% noise. Note that the presence of noise renders *all* four problems intransitive.



(a) Tic-tac-toe with 0% noise

(b) Nim with 0% noise

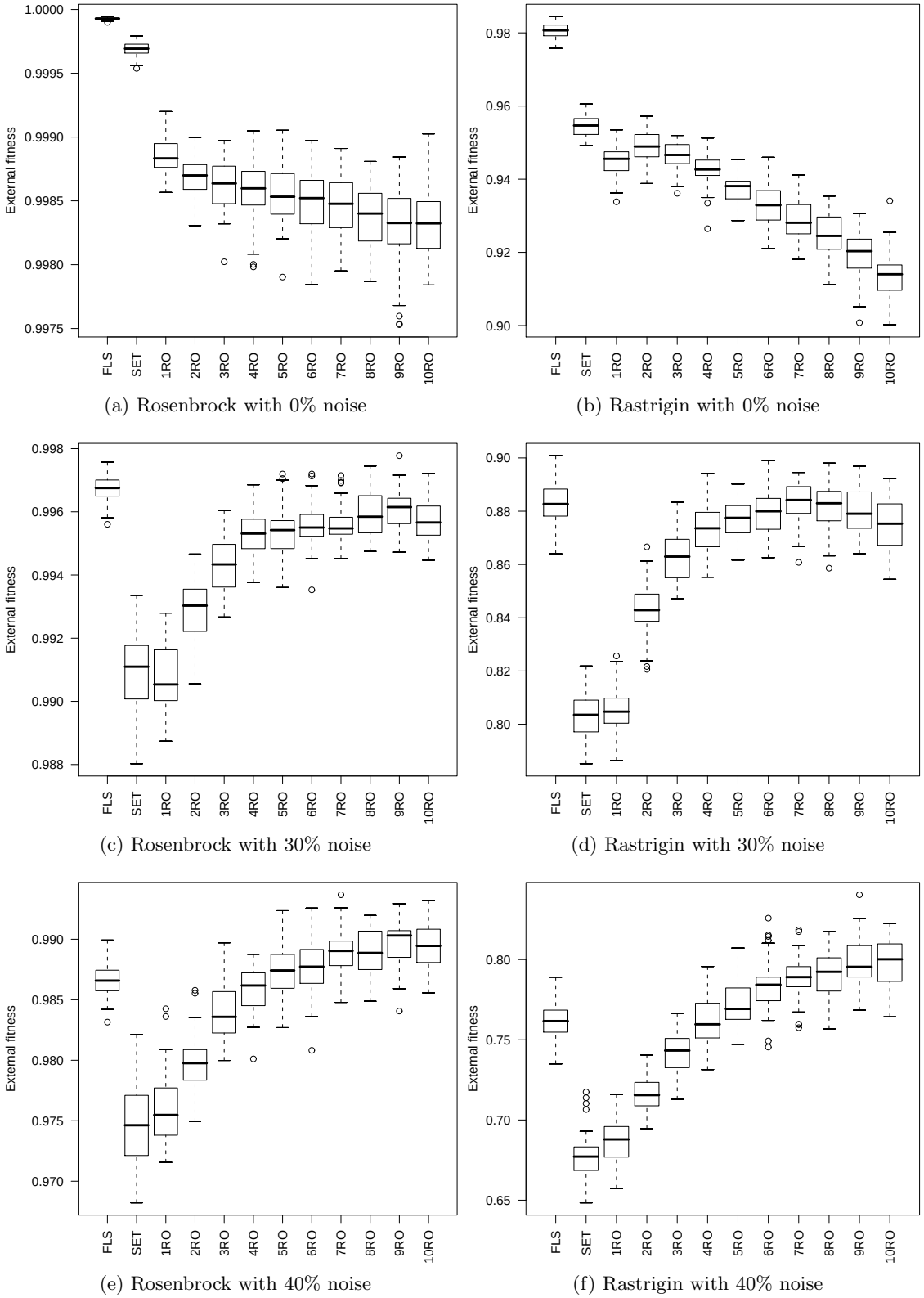
(c) Tic-tac-toe with 30% noise

(d) Nim with 30% noise

(e) Tic Tac Toe with 40% noise

(f) Nim with 40% noise

Figure 3.5.1.: Results for tic-tac-toe and Nim.



(a) Rosenbrock with 0% noise

(b) Rastrigin with 0% noise

(c) Rosenbrock with 30% noise

(d) Rastrigin with 30% noise

(e) Rosenbrock with 40% noise

(f) Rastrigin with 40% noise

Figure 3.5.2.: Results for Rosenbrock and Rastrigin.

It seems that FLS is less affected by noise than SET. In the hierarchical process of SET, each distorted game impacts the subsequent rounds. Even the (objectively) best-of-generation individual may be dropped behind due to noise. FLS turns out to be more resistant to noise as the random reversal of game outcome influences only one selection act.

In the overall picture, kRO shows the ability to attain the best resistance to noise among all the considered architectures: it performs at least as well or better than FLS and SET for some values of k, especially for the highest noise level considered (40%). However, the optimal value of k varies across the noise levels and problems, and it is difficult to determine the optimal k in advance. In general, higher values of k compensate for the presence of noise, but also shorten the evolutionary run by increasing the required number of games in each generation. FLS almost always offers a statistically equivalent or better performance than kRO and thus may be considered as an attractive option.

3.6. Discussion and Conclusions

In this chapter we proposed Fitnessless Coevolution, a new one-population coevolutionary method dedicated to solving symmetrical test-based problems. We proved that an evolutionary process employing Fitnessless Coevolution is equivalent to the fitness-based coevolution provided the fulfillment of transitivity condition (3.3.1).

The presented experimental results demonstrate that Fitnessless Coevolution is an appealing alternative to single-elimination tournament and the k-random opponents method, especially when the task fulfills the transitivity condition. Though this constraint is unlikely to hold for the majority of test-based problems, we hypothesize that the effectiveness of Fitnessless Coevolution increases with the *extent* of transitivity (meant as, e.g., the probability that transitivity holds for a pair of individuals randomly drawn from a population). However, this phenomena may be more complex and depend, among others, on the *structure* of transitivity as well, so this supposition requires verification in a separate study.

The mechanism of Fitnessless Coevolution is elegant and simple in at least two ways: in getting rid of the numerical fitness and in combining the evaluation and selection phase. Despite this simplicity, it produces effective solutions and is immune to noise to an extent that is comparable to kRO. The downside of the method is the extra effort required to appoint the best-of-run individual.

One could argue that, no matter whether the objective function exists, does not exist, or is difficult to define, there is always *some* way of estimating the numerical fitness, so

there is no need to apply fitnessless approach. Indeed, SET and kRO are examples of such ways. Note, however, how arbitrary they are; in particular, they have to assume a specific way of 'translating' the game outcome, usually defined only on an ordinal scale, into a numerical value, and some method to aggregate such quantities into fitness. Fitnessless selection, on the contrary, is conceptually simpler and requires few assumptions.

Another attractive property of Fitnessless Coevolution is its locality with respect to the population. SET requires simultaneous access to all individuals in the population. A single act of fitnessless selection, on the contrary, engages only a few individuals. This may have positive impact on the performance in case of parallel implementation, and may be nicely combined with other evolutionary techniques that involve locality, like the island model or spatially distributed populations.

While this can hardly be acknowledged to be a real virtue from the scientific perspective, it is interesting to note that Fitnessless Coevolution is natural. Similarly to biological evolution, the success of an individual depends here *directly* on its competition with other individuals. Also, the fitness function used in a standard evolutionary algorithm is essentially a mere technical means to impose selective pressure on the evolving population, whereas its biological counterpart (fitness) is defined *a posteriori* as probability of survival. By eliminating the numerical fitness, we avoid subjectivity that its definition is prone to.

In Chapter 4, we will demonstrate the ability of Fitnessless Coevolution to evolve human-competitive players for a game with partially observable states.

4. Application of Fitnessless Coevolution

In Chapter 3 we introduced Fitnessless Coevolution algorithm for symmetrical test-based problems and verified it experimentally on a set of simple test-based problems. Here we extend the experimental study by providing a complete record of application of Fitnessless Coevolution to a more sophisticated test-based problem: the game of Ant Wars. We will demonstrate that, although Ant Wars are unlikely to fulfill the requirement of transitivity condition (3.3.1), Fitnessless Coevolution allows to obtain for this problem a very good strategy in absolute terms.

4.1. Introduction

The *Ant Wars* contest was organized as a part of the 2007 edition of Genetic and Evolutionary Computation Conference (GECCO, London, July 7–12, 2007), the largest international scientific event pertaining to evolutionary algorithms. The contest aimed at evolving a controller for a virtual ant that collects food in a square toroidal grid environment in the presence of a competing ant. In a sense, this game extends the Artificial Ant problem [77], a popular genetic programming benchmark, into the framework of a two-player game.

Ant Wars is a probabilistic two-person board game with imperfect information and partial observability. The game starts with 15 pieces of food randomly distributed over an 11×11 toroidal board and two players, called Ant 1 and Ant 2, placed at predetermined locations, $(5, 2)$ for Ant 1 and $(5, 8)$ for Ant 2. No piece of food can be located in the ants' starting cells. An ant's field of view is limited to a square 5×5 neighborhood centered at its current location. An ant receives the complete information about the states (empty, food, enemy) of all cells within its field of view.

The game lasts for 35 turns per player. In each turn an ant moves into one of the 8 neighboring cells. Ant 1 moves first. Moving into an empty cell has no effect. If an ant moves into a cell with food, it scores 1 point and the cell is emptied. Moving into the cell occupied by the opponent kills it: no points are scored, but only the survivor can go on collecting food until the end of the game. A game is won by the ant that reaches

the higher score. In case of a tie, Ant 1 is the winner.

As the outcome of the game depends on spatial distribution of food pieces, the proper choice of a better one of two players requires grouping multiple games into *matches* played on different boards. A match consists of $2 \times k$ games played on k random boards generated independently for each match. To provide for fair play, the contestants play two games on the same board, in the first game taking roles of Ant 1 and Ant 2, and then exchanging these roles. We refer to such a pair of games as a *double-game*. To win a $2 \times k$-games match, an ant has to win $k + 1$ or more games. In the case of tie, the total score determines the match outcome. If there is still a tie, a randomly selected contestant wins.

The contest rules required an ant's controller to be encoded as an ANSI-C function *Move*(*grid*, *row*, *column*), where *grid* is a two-dimensional array representing the board state, and (*row*, *column*) represents the ant's position. The function indicates the ant's next move by returning the direction encoded as an integer from the interval $[0, 7]$. The source code of the function was not allowed to exceed 5kB in length.

In this chapter, we tell the story of *BrilliAnt*, the Ant Wars winner. BrilliAnt has been evolved through competitive one-population coevolution using genetic programming and Fitnessless Coevolution described in Chapter 3. We assess BrilliAnt's human-competitiveness in both direct terms (playing against a human opponent) and indirect terms (playing against a human-devised strategy), and analyze its behavioral patterns.

4.2. Genetic Programming and Game Strategies

Genetic programming (GP, [77]) is an evolutionary method of finding computer programs that perform a given task. It is known for human-competitive results in many areas including, among others, electronic design [79], quantum algorithms [133], bioinformatics [78], and game-playing. As to the latter, Koza was the first who used it to evolve strategies [76] for a simple discrete two-person game. Since then, it has been demonstrated many times that the symbolic nature of GP is suitable for game strategy learning. Past studies on the topic include both trivial games such as tic-tac-toe [3] or Spoof [142], as well as more complicated and computationally-demanding games, like poker [132]. Core Wars, a game in which two or more programs compete for the control of a virtual computer, is a popular benchmark problem for evolutionary computation and one of the best evolved players was created using GP [23]. Luke's work [89, 87] on evolving soccer softball team for RoboCup97 competition belongs to the most ambitious applications of GP to game playing since it involved a complicated environment

and teamwork. Also some fundamental problems from game theory such as the prisoner's dilemma have been approached with GP [24]. Recently, Sipper and his coworkers demonstrated [130] human-competitive GP-based solutions in three areas: backgammon [6], RoboCode (tank-fight simulator, [128]) and chess endgames [53].

4.3. Strategy Encoding

An ant's field of view (FOV) contains 25 cells and occupies 20.7% of board area, so, assuming an unbiased random distribution of food pieces, the expected number of visible food pieces is 3.02 when the game begins. The probability of having n food pieces within the FOV drops quickly as n increases; for instance, for $n = 8$ it amounts to less than 0.5%. Thus, most of the food to be collected is usually beyond the FOV. Also, a reasonable strategy should obviously take into account the rotational invariance and symmetry of FOV; e.g., for two mirrored FOV states, an ant should behave in a mirrored way. These facts let us conclude that the number of distinct and likely FOV states is relatively low, and that a strategy based only on the *observed* FOV state cannot be competitive in the long run. It seems reasonable to virtually extend the FOV by keeping track of the past board states. Thus, we equip our ants with *memory*, implemented by three arrays that are overlaid over the board:

- *Food memory* F, which keeps track of food locations observed in the past,
- *Belief table* B, which describes the ant's belief in the current board state,
- *Track table* V, which stores the cells already visited by the ant.

After each move, we copy food locations from the ant's FOV into F. Within the FOV, old states of F are overridden by the new ones, while F cells outside the current FOV remain intact. As the board state may change subject to the opponent's actions and make the memory state obsolete, we also simulate a memory decay in the belief table B. Initially, the belief for all cells is set to 0. Belief for the cells within the FOV is always 1, while outside the FOV it fades exponentially by 10% with each move. Table V, initially filled with zeros, stores the ant's 'pheromone track' by setting the visited elements to 1.

To represent our ants we use the tree-based strongly-typed genetic programming [101]. A GP tree is expected to evaluate the utility of the move in a particular direction: the more attractive the move, the greater the tree's output. To provide for rotational invariance, we evaluate multiple orientations using the same tree. However, as the ants are allowed to move both straight and diagonally, we store *two* trees in each individual,

Table 4.1.: The terminals used by evolving strategies.

Terminal	Interpretation
Const()	An ephemeral random constant (ERC) for type F ($[-1; 1]$)
ConstInt()	An integer-valued ERC for type F (0..5)
Rect()	An ERC for type A
TimeLeft()	The number of moves remaining to the end of the game
Points()	The number of food pieces collected so far by the ant
PointsLeft()	Returns 15−Points()

one for handling the straight directions (N, E, S, W) and one to handle the diagonal directions (NE, NW, SE, SW)[1]. Given a particular FOV state, we present it to the trees by appropriately rotating the FOV, the remaining part of the board, and the memory, by a multiple of 90 degrees, and querying both trees each time. Among the eight obtained values, the maximum response indicates the most desired direction and determines the ant's next move; ties are resolved by preferring the earlier maximum.

We define three data types: *float* (F), *boolean* (B), and *area* (A). The area type represents a rectangle stored as a quadruple of numbers: dimensions and midpoint coordinates (relative to ant's current position, modulo board dimensions). To avoid considering exceedingly large areas, we constrain the sum of area dimensions to 6 by an appropriate genotype-to-phenotype mapping. For instance, the dimensions encoded as (2, 5) in the genotype are effectively mapped to (1, 4) during tree execution.

The GP function set and the terminals are presented in Tables 4.1 and 4.2. Note that some functions calculate their return values not only from the actual state of the board, but also from the food memory table F and the belief table B. For example, NFood(A) returns the scalar product of table F (food pieces) and table B (belief), constrained to area A.

It is worth emphasizing that all GP functions used here are straightforward. Even the most complex of them boil down to counting matrix elements in designated rectangular areas. Though one could easily come up with more sophisticated functions, this would contradict the rules of the contest, which promote the evolved rather than the designed intelligence.

[1] We considered using a single tree and mapping the diagonal board views into the straight ones; however, this leads to significant topological distortions which could deteriorate the ant's perception.

Table 4.2.: The non-terminals.

Non-terminal	Interpretation
IsFood(A)	Returns *true* iff A contains at least one piece of food.
IsEnemy(A)	Returns *true* iff A contains the opponent.
And(B, B)	Logic functions
Or(B, B)	
Not(B)	
IsSmaller(F, F)	Arithmetic comparators
IsEqual(F, F)	
Add(F, F)	Scalar arithmetics
Sub(F, F)	
Mul(F, F)	
If(B, F, F)	The conditional statement
NFood(A)	The number of food pieces in the area A
NEmpty(A)	The number of empty cells in the area A
NVisited(A)	The number of cells already visited in the area A
FoodHope()	Returns the maximal number of food pieces that may be reached by the ant within two moves (assuming the first move is made straight ahead, and the next one in an arbitrary direction)

4.4. The Experiment

To limit human intervention, our ants undergo competitive evaluation, i.e., face each other, rather than an external selection pressure. To this aim, we used Fitnessless Coevolution described in Chapter 3.

To make the initial decisions about the experimental setup and parameter settings, we ran some preliminary experiments. As a result, we decided to set the run length to around 1500 generations, and, in order to effectively utilize the two-core processor, to employ the island model [141] with two populations, each of approximately 2000 individuals. In all experiments, we used probabilities of crossover, mutation, and ERC mutation, equal to 0.8, 0.1, and 0.1, respectively. GP trees were initialized using ramped half-and-half method and were not allowed to exceed the depth of 8. The experiment was implemented in the ECJ [88] framework and for the remaining parameters we used the ECJ's defaults.

We rely on the default implementation of GP mutation and crossover available in ECJ, while providing specialized ERC mutation operators for particular ERC nodes. For Const() we perturb the ERC by a random, normally distributed value with mean

0.0 and standard deviation 1/3. For ConstInt(), we perturb the ERC by a random, uniformly distributed integer value from interval $[-1; 1]$. For Rect(), we perturb each rectangle coordinate or dimension by a random, uniformly distributed integer value from interval $[-1; 1]$. In all cases, we trim the resulting values to domain intervals.

To speed up the selection process and to meet the contest rules that required the ant code to be provided in C programming language (ECJ is written in Java), in each generation we serialize the entire population into one large text file, encoding each individual as a separate C function. The resulting file is then compiled and linked with the game engine, also written in C. The resulting executable is subsequently launched and carries out the selection, returning the identifiers of the selected individuals to ECJ. The compilation overhead is reasonably small, and it is paid off by the speedup provided by using C language. This approach allows us also to monitor the actual size of C code, constrained by the contest rules to 5kB per individual.

The best ant emerged in an experiment with the population of 2250 individuals evolving for 1350 generations, using the fitnessless selection with tournament size $k = 5$ (thus 4 matches per single-elimination tournament), and with 2×6 games played in each match. We named it *BrilliAnt,* submitted it to the Ant Wars competition, and won it. BrilliAnt not only evolved, but was also selected in a completely autonomous way, by running a round-robin tournament involving all 2250 individuals from the last generation of the evolutionary run[2]. This process was computationally demanding: having only one double-game per match, the total number of games needed was more than 5,000,000, an equivalent of about 47 generations of evolution.

In order to determine BrilliAnt's ability to play Ant Wars, we assessed its human-competitiveness, which is a notion introduced by Koza [79] within Genetic Programing. In experiments, we analyzed two variants of this notion: *direct competitiveness*, i.e., the performance of the evolved solution playing against a human, and *indirect competitiveness*, meant as the performance of the evolved solution playing against a *program* designed by a human. For the former purpose, we implemented a software simulator that allows humans to play games against an evolved ant. Using this tool, an experienced human player played 150 games against BrilliAnt, winning only 64 (43%) of them and losing the remaining 86 games (57%). We also developed an online version of this tool that allows everybody to play with BrilliAnt. At the time of writing, the collected statistics confirm the above result: 514 games won by BrilliAnt vs. 188 won by humans, and one draw. Even if we assume that inexperienced beginners account for a great part of this statistics, these figures clearly indicate that the strategy elaborated by our ap-

[2]Selecting the best individual in a noisy environment is not a trivial task, see, e.g., [58]

Table 4.3.: The results of a round-robin tournament involving the evolved ants (in bold) and humants (plain font). Maximum possible score is 21,000,000.

Player	Matches won	Games won	Total score
ExpertAnt	6	760,669	10,598,317
HyperHumant	6	754,303	10,390,659
BrilliAnt	6	753,212	10,714,050
EvolAnt3	3	736,862	10,621,773
SuperHumant	3	725,269	10,130,664
EvolAnt2	3	721,856	10,433,165
EvolAnt1	1	699,320	10,355,044
SmartHumant	0	448,509	9,198,296

proach is challenging for humans. The reader is encouraged to visit the Web page [62] and measure swords with BrilliAnt.

To analyze BrilliAnt's indirect competitiveness, we let it play against human-designed strategies — *humants*. We manually implemented several humants of increasing sophistication (*SmartHumant*, *SuperHumant*, and *HyperHumant*). HyperHumant, the best humant we could develop, memorizes the states of board cells observed in the past, plans 5 moves ahead, uses a probabilistic memory model, and implements several endgame rules (e.g., *when your score is 7, eat the food piece even if the opponent is next to it*).

Table 4.3 and Fig. 4.4.1 present the results of the round-robin tournament involving the three humants, BrilliAnt, and four other evolved ants (*ExpertAnt, EvolAnt1, EvolAnt2, EvolAnt3*). Each pair of strategies played a match of $100,000$ double-games. An arrow leading from a to b means that a turned out to be statistically better than b; no arrow means no statistical advantage (at 0.01 level[3]). Note that one of the evolved individuals, *ExpertAnt*, won 50.12% of games against HyperHumant. As BrilliAnt turned out to be worse than HyperHumant (loosing 52.02% of games), ExpertAnt could be considered a better pick for the Ant Wars contest. However, ExpertAnt has been selected by explicitly testing all ants from the last generation against the *manually designed* HyperHumant. BrilliAnt, on the contrary, evolved and was selected completely autonomously, so it has been appointed as our contestant.

Programs in C implementing strategies of all players from Table 4.3 are presented in Appendix on page 127. Notice that while human-designed strategies are easy to follow,

[3]We estimate the statistical significance of the outcome of a match from the tails of the binomial distribution assuming the probability of success of 0.5 (the zero hypothesis is that both players win the same number of games, i.e., 50,000 games in this context).

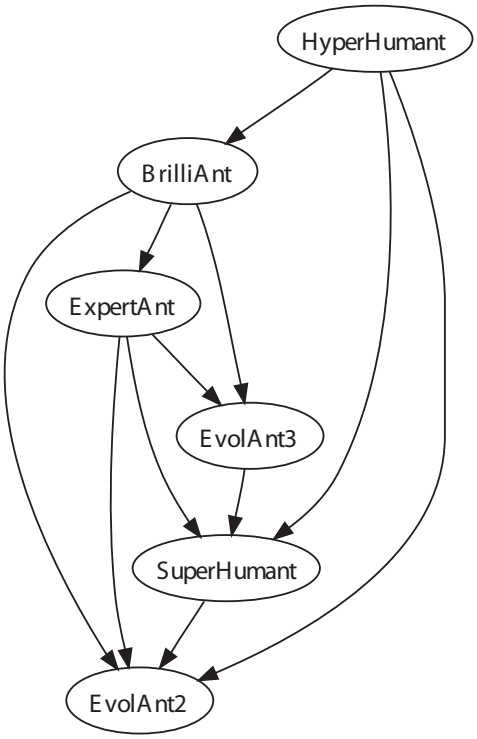


Figure 4.4.1.: Graph showing relations between players. If player a is statistically better than player b ($p = 0.01$), an arrow leads from a to b . If none of them is better, no arrow is drawn. EvolAnt1 and SmartHumant were not showed to improve graph's readability.

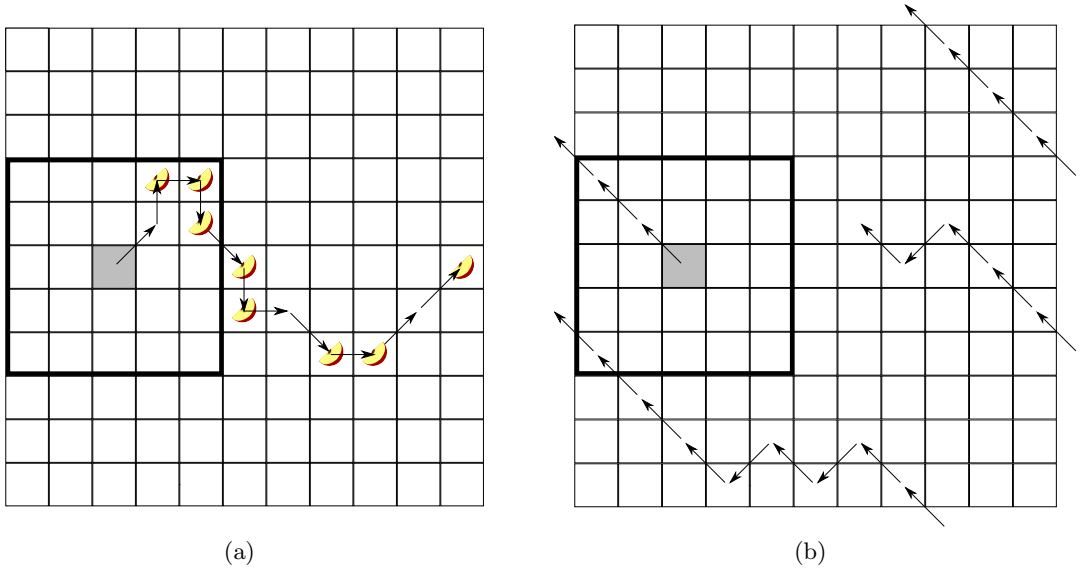
the evolved ones resemble programs processed by an obfuscator (see, e.g., expressions in lines 28–29 and 32–33 of BrilliAnt). Apparently, coevolution does not care about code readability.

4.5. Analysis of BrilliAnt's Strategy

BrilliAnt's genotype (a pair of GP trees) contains 237 nodes, and analysis of its code would be hard. However, we are able to analyze its phenotype, meant as its behavior in the course of game playing. BrilliAnt exhibits a surprisingly rich repertoire of behavioral patterns, ranging from obvious to quite sophisticated ones. Faced with the FOV with two corner areas occupied by food, BrilliAnt always selects the direction that gives a chance for more food pieces. It reasonably handles the trade-off between food amount and food proximity, measured using the chessboard (Chebyshev) distance (the minimal number of moves required to reach a cell). For instance, given a group of two pieces of food at distance 2 ($(2,2)$ for short), and a group of two pieces of food at distance 1, i.e. $(2,1)$, BrilliAnt chooses the latter option, i.e., $(2,2) \prec (2,1)$. Similarly, $(1,1) \prec (2,2)$, $(3,2) \prec (2,1)$, $(3,2) \prec (3,1)$, and $(2,2) \prec (3,2)$. If both food groups contain the same number of food pieces but one of them is accompanied by the opponent, BrilliAnt chooses the other group.

Food pieces sometimes happen to arrange into 'trails', similar to those found in the Artificial Ant benchmarks [77]. BrilliAnt perfectly follows such paths as long as the gaps between trail fragments are no longer than 2 cells (see Fig. 4.5.1a for an example). However, when faced with a large isolated group of food pieces, it does not always consume them in an optimal order, i.e., in a minimum number of moves.

If the FOV does not contain any food, BrilliAnt proceeds in the NW direction. However, as the board is toroidal, keeping moving in the same direction makes sense only to a certain point, because it brings the player back to the starting point after 11 moves, with a significant part of the board still unexplored. Apparently, evolution discovered this fact: after 7 steps in the NW direction (i.e., when FOV starts to intersect with the initial FOV), BrilliAnt changes its direction to SW, pursuing the following sequence: 7NW, 1SW, 1NW, 1SW, 6NW, 1SW, 1NW. A simple analysis reveals that 18 moves, shown in Fig. 4.5.1b, provide the complete coverage of the board. This behavior seems quite efficient, as the minimal number of moves that scan the entire board is 15. Note also that this contains only diagonal moves. In the absence of any other incentives, this is a locally optimal choice, as a diagonal move uncovers 9 board cells, while a non-diagonal one uncovers only 5 of them.



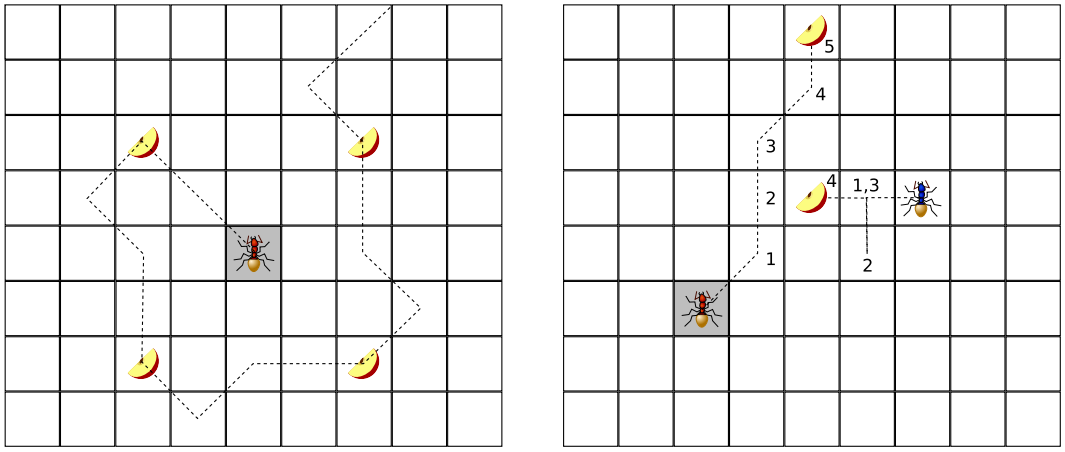
(a) (b)

Figure 4.5.1.: BrilliAnt's behaviors when following a trail of food pieces (a), and in the absence of food (b). Gray cell and large rectangle mark BrilliAnt's starting position and initial FOV, respectively.

Evolving this full-board scan is quite an achievement, as it manifests in the absence of food, a situation that is almost impossible in Ant Wars, except for the highly unlikely scenario of the opponent consuming all the food earlier. BrilliAnt exhibits variants of this behavioral pattern also after some food pieces have been eaten and its FOV is empty.

BrilliAnt also makes reasonable use of its memory. When confronted with multiple groups of food pieces, it chooses one of them and, after consuming it, returns to the other group(s), unless it has spotted some other food in the meantime. This behavior is demonstrated in Fig. 4.5.2a where BrilliAnt, faced with the initial board state with four food pieces visible in the corners of FOV, follows an almost optimal trajectory. Note that as soon as it makes the first NW move, three food pieces disappear from the FOV, so memory is indispensable here. After completing this task, BrilliAnt proceeds to the unexplored parts of the board.

BrilliAnt usually avoids the opponent, unless it comes together with food and no other food pieces are in view. In such a case, it approaches the food, maintaining at least distance 2 from the opponent. For an isolated food piece, this often ends in a deadlock: the players hesitatingly walk in the direct neighborhood of the food piece, keeping safe distance from each other. None of them can eat the piece, as the opponent immediately kills such a daredevil. This behavior is shown in Fig. 4.5.2b, where BrilliAnt is the ant



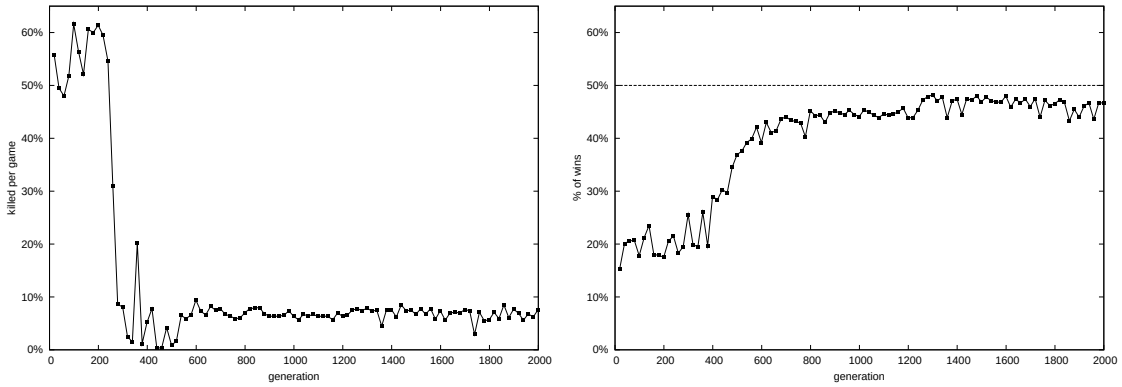
(a) (b)

Figure 4.5.2.: (a) BrilliAnt's memory at work. (b) The deadlock situation and its resolution.

with starting position marked by the gray cell, and the numbers reflect ants' positions in consecutive turns (BrilliAnt moves first). After making the first move, BrilliAnt spots the opponent on the other 'side' of food, so it does not eat the piece but walks along it (moves 2 and 3). In the meantime, the opponent behaves analogously. In the absence of any other incentives, this behavior could last till the end of the game. However, as soon as BrilliAnt spots another piece of food (after making move #3 in Fig. 4.5.2b), it changes its mind and starts heading towards it, leaving the disputed food piece to the opponent.

BrilliAnt also learned how to resolve such deadlocks. When the end of the game comes close and the likelihood of finding more food becomes low, it may pay off to sacrifice one's life in exchange for food — there will be not much time left for the opponent to gather more food. This in particular applies to the scenario when both players scored 7 and only the last food piece is left. This 'kamikaze' behavior also emerged in other evolutionary runs. Figure 4.5.3b illustrates this behavior in terms of the death rate statistic for one of the experiments. The ants from the several initial generations play poorly and are likely to be killed by the opponent. With time, they learn how to avoid the enemy and, usually at 200-300th generation, the best ants become perfect at escaping that threat (see Fig. 4.5.3b). Then, around 400-500th generation, the ants discover the benefit of the 'kamikaze' strategy, which results in a notable increase of death rate, but pays off in terms of the winning frequency.

BrilliAnt is also able to correctly estimate its chances of reaching a piece of food



(a) Percent of deaths per game (b) Winning frequency

Figure 4.5.3.: The dynamics of a typical evolutionary run. Each point corresponds to the best-of-generation ant chosen on the basis of 2×250 games against HyperHumant.

before the (visible) opponent, while taking into account the risk of being eaten (see Fig. 4.6.1; in all scenarios shown here BrilliAnt, located at the center, moves first). In Fig. 4.6.1a, BrilliAnt decides to approach the food because its first move effectively repels the opponent. In Fig. 4.6.1b, on the contrary, BrilliAnt walks away as it has no chance of reaching the food piece before the opponent (the shortest path traverses the cell controlled by the opponent). The situation depicted in Fig. 4.6.1c gives equal chances to both players to reach the food piece, so BrilliAnt approaches it, maintaining a safe distance from the opponent. If the opponent moves analogously to the north, this may end up in a deadlock described earlier. However, on the way to the food piece BrilliAnt or the opponent may spot another food piece and walk away, so this behavior seems reasonable. If there is a choice between an uncertain food piece and food piece that may be reached at full safety, BrilliAnt chooses the latter option, as shown in Fig. 4.6.1d.

4.6. Conclusions

We described an application of Fitnessless Coevolution to a complex test-based problem. From the computational intelligence perspective, it is important to notice that the coevolutionary process used relatively little domain knowledge to solve the problem. In particular, both the evolution and the selection of the best-of-run individual are completely autonomous and do not involve any external (e.g., human-made) strategies. The evolved players are human-competitive in both direct and indirect sense and

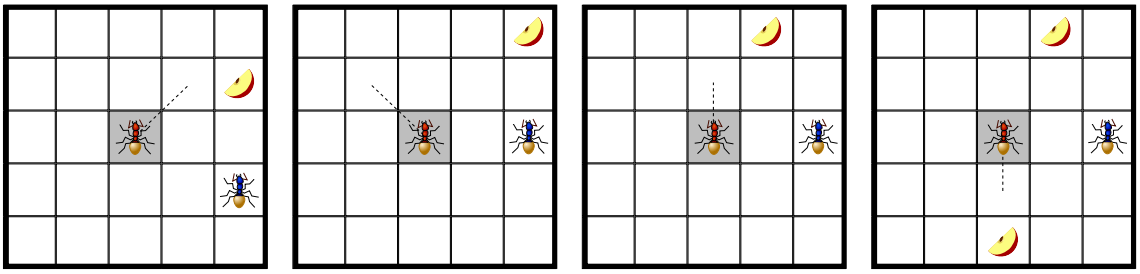


Figure 4.6.1.: BrilliAnt's behavior when faced with food and an opponent.

make reasonable choices based on the visible part of the board and memory state. In particular, BrilliAnt's traits include close-to-optimal board scanning in search of food, collecting of previously seen and memorized food pieces, ability to cancel the previous plans after discovering new food pieces, and rational estimation of chances of reaching the food before the opponent does. Also, BrilliAnt's strategy is dynamic, i.e., changing with the stage of the game, as demonstrated by its ability of sacrifice in exchange of food.

As we have shown, though unusual from the viewpoint of the core evolutionary computation research, selection without fitness has much rationale. The traditional fitness function is essentially a mere technical means to impose the selective pressure on the evolving population. It is often the case that, for a particular problem, the definition of fitness does not strictly conform to its biological counterpart, i.e., the *a posteriori* probability of the genotype survival. By eliminating the need for an arbitrary numeric fitness, we avoid the subjectivity that its definition is prone to.

In its pure form, Fitnessless Coevolution is a rather straightforward approach, so there are many conceivable ways in which it could be extended. In particular, as opposed to many modern coevolutionary algorithms, it lacks an *archive*, a form of memory that helps the algorithm to maintain progress during search process. In the following chapter we describe the concept of coordinate system for test based problems, a formal object that, as we demonstrate later in Chapter 6, can be used to construct a specific form of archive.

5. Coordinate Systems for Test-Based Problems

In this chapter, we put emphasis on theoretical research of test-based problems. To this aim, we exploit a concept of underlying structure of test-based problems in a form of coordinate systems, a formalism based on the principle of Pareto-coevolution. Our objective in this chapter is to better understand its properties and to prepare ground for a new algorithm based on this concept, which will follow in Chapter 6.

5.1. Introduction

In test-based problems, the outcome of a single interaction is usually not enough to provide an informative gradient and effectively guide the search process. Thus, in a typical coevolutionary algorithm multiple outcomes are aggregated, leading to an overall fitness measure, like probability of winning or accuracy of classification. In this way, however, the individual characteristics of particular candidate solutions and tests are inevitably lost, and the performance measure is prone to compensation — two candidate solutions with completely different outcomes of interactions with tests can obtain the same fitness. The search algorithm has no insight into the actual, complex interactions taking place between candidate solutions and tests.

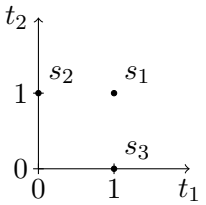


Figure 5.1.1.: Pareto coevolution. Each test (here t_1 and t_2) serves as a separate objective. Candidate solutions (here s_1, s_2 and s_3) are embedded in the space spanned by the objectives, i.e., placed according to their performance on tests. For instance, candidate solution s_3 solves test t_1, but fails test t_2.

The above-mentioned aggregation of interaction outcomes is one of the reasons for which coevolutionary algorithms often suffer from coevolutionary pathologies (cf. Section 2.3.3). In *Pareto coevolution* [43, 103] proposed to overcome these drawbacks, aggregation of interaction outcomes has been abandoned in favor of using each test as a separate objective. This transforms the test-based problem into a multi-objective optimization problem and allows resorting to the well-defined concept of dominance — candidate solution s_1 is not worse than candidate solution s_2 if and only if s_1 performs at least as good as s_2 on all tests (see Fig. 5.1.1). And, as we pointed out in Section 2.2.4, Pareto dominance is common ingredient of all solution concepts. This resembles the process of converting a single-objective problem into a multi-objective, which is termed *multi-objectivization* [74]. Unfortunately, in test-based problems the number of tests is usually prohibitively large or even infinite: recall, for instance, the number of strategies in tic-tac-toe. Therefore, also the dimension of search space in Pareto coevolution is enormous.

It was observed however, that the number of objectives in Pareto coevolution can often be reduced, since many test-based problems possess an *internal structure*. This structure manifests itself by the fact that sometimes it is possible to determine a group of tests that examine the same skill or aspect of solution performance, but with different intensity. Instead of defining different objectives, such tests can be ordered with respect to difficulty and placed on a common axis identified with a single new objective. Since such an objective is not known *a priori*, but must be revealed during exploration of the problem, it is referred to as an *underlying objective*, a term introduced by [34]. For instance, the underlying objectives in chess could include skills of controlling the center of the board, using knights, playing endgames, etc.

The above intuition about underlying objectives and internal structure of a problem was first formalized in the notion of *coordinate system* by Bucci et al. [17]. An important feature of coordinate system is that while *compressing* the initial set of objectives, it preserves the relations between candidate solutions and tests. Each candidate solution is embedded in the system and the outcome of its interaction with any test can be determined given its position on all axes. As stated by [17], *the structure space captures essential information about a problem in an efficient manner.*

Beyond the purely aesthetic appeal, the practical motivation for extracting the internal structure of a problem is twofold. First, there is evidence that such structure may be exploited for the benefit of coevolutionary algorithms, for instance, by accelerating convergence or guaranteeing progress, like in Dimension Extraction Coevolutionary Algorithm [32]. Second, by knowing the internal structure and underlying objectives, we

can learn important properties of the problem [33]. The answer to the question 'what are the underlying objectives of my problem' can give valuable insight into problem properties and help choose a method to solve it. The presumably most important problem property is its *dimension,* i.e., the number of underlying objectives. It is hypothesized that problem dimension is highly correlated with its difficulty.

We concentrate on the computational aspects of extracting the problem structure and try to answer the questions: how to extract the underlying objectives of a problem, and, even more importantly, how to do it efficiently, so that the underlying objectives can be updated during a coevolutionary run and exploited for the sake of improving search convergence, without potentially outweighing these benefits with enormous computational overhead. To this aim, we elaborate on the particular type of coordinate system defined in [17], formally introducing all the necessary concepts in Sections 5.2 and 5.3. In Section 5.5, we identify important new properties of coordinate systems and point out their relations to partially ordered sets. After proving the NP-hardness of the problem of determining the size of the minimal coordinate system in Section 5.7, we provide exact and approximate algorithms for building a minimal coordinate system (Section 5.8), and use them in computational experiments in Section 5.9.

5.2. Preliminaries

In this chapter, we assume that the codomain of the interaction function G is a binary set $\{0 < 1\}$. If $G(s,t) = 1$, we say that candidate solution s *solves* test t; if $G(s,t) = 0$, we say that s *fails* test t. Where convenient, we will treat G as a relation and denote the fact that a candidate solution s solves test t as $G(s,t)$ and the fact that it fails test t as $\neg G(s,t)$.

Definition 20. S*olutions failed set* $SF(t) \subseteq S$ is comprised of all candidate solutions that fail the test t. Analogically, *tests solved set* $TS(s) \subseteq T$ is comprised of all tests that are solved by candidate solution s.

Notice also that $t \in TS(s) \iff s \notin SF(t)$ for all $s \in S$, $t \in T$, since both sides hold if and only if s solves t.

Definition 21. Test t_1 is *weakly dominated* by test t_2, written $t_1 \leq t_2$, when $SF(t_1) \subseteq SF(t_2)$ for $t_1, t_2 \in T$. Analogically, candidate solution s_1 is *weakly dominated* by candidate solution s_2, written $s_1 \leq s_2$, when $TS(s_1) \subseteq TS(s_2)$ for $s_1, s_2 \in S$.

For brevity we use the same symbol $\leq$ for both relations, as they are univocally determined by the context. Since $\leq$ inherits transitivity and reflexivity from $\subseteq$, it is

a preorder in both S and T. To make $\leq$ a partial order we need to assume that no two elements of one set are indiscernible with respect to how they interact with the elements of the other set, precisely: $\nexists t_1, t_2 \in T,\ t_1 \neq t_2 : SF(t_1) = SF(t_2)$ and $\nexists s_1, s_2 \in S,\ s_1 \neq s_2 : TS(s_1) = TS(s_2)$. Under this assumption $s_1 = s_2 \iff TS(s_1) = TS(s_2)$ and, analogically, $t_1 = t_2 \iff SF(t_1) = SF(t_2)$; thus $(S, \leq)$ and $(T, \leq)$ are posets, which eases our further arguments. In case some indiscernible objects do exist (it can happen in practice), we can merge them into one object without losing any important features of $\mathcal{G}$.

In this chapter, we will not refer to solution concepts, but only to the dominance relation, which is, as we pointed out in Chapter 2, common for all solution concepts. Therefore, we will abuse the notation of test-based problems by writing $\mathcal{G} = (G, S, T)$, abstracting from any specific set of potential solutions $\mathcal{P}$ and the set of solutions $\mathcal{P}^+$, defined by a solution concept.

5.3. Coordinate System

In the context of test-based problems, a coordinate system is a formal concept revealing internal problem structure by enabling the candidate solutions and tests to be embedded into a multidimensional space. Of particular interests are such definitions of coordinate systems, in which the relations between candidate solutions and tests ($\leq$) are reflected in spatial arrangement of their locations in the coordinate system. Previous work suggests that this formalism can help design better coevolutionary algorithms [32] and examining properties of certain problems [33].

There is no unique definition of coordinate system for a test-based problem; currently we are aware of two formulations: by Bucci et al. [17] and de Jong and Bucci [32], further investigated in [33]. The difference between them lies in the way they define axes: the former defines an axis as a sequence of tests ordered by the domination relation, whereas the latter as a sequence of sets of candidate solutions ordered by the inclusion relation. In this chapter we analyze the coordinate system introduced in [17], so in the following by coordinate system we mean the one defined there. There are slight differences in our formulation, which, however, do not affect any important properties of the coordinate system. First, in our formulation, the positions of candidate solutions on an axis are shifted one test to the left, which is more convenient. Second, Bucci *et al.* worked with preordered sets, but we, as pointed out earlier, limit our discussion to posets. The reason for this simplification is merely technical, since some mathematical concepts we need were defined for posets and their extensions to preordered sets require additional

effort (see, e.g., the dimension of the preordered set in chapter 4.2.1 of [15]). Our results could be generalized to a situation where S and T are preordered sets, thus they are applicable to any test-based problem, however, in this thesis, we stick with posets to make our presentation more comprehensible.

For convenience, we introduce a formal element t_0 such that $G(s, t_0)$ for all $s \in S$. Also, we define an operator 'overline' that augments a set of tests with t_0, i.e., $\bar{X} = X \cup \{t_0\}$.

Definition 22. The *coordinate system* $\mathcal{C}$ for a test-based problem $\mathcal{G}$ is a set of axes $(A_i)_{i \in I}$, where each axis $A_i \subseteq T$ is linearly ordered by $<$. I is an index set and the *size of the coordinate system*, denoted by $|\mathcal{C}|$, is the cardinality of I.

We interpret an axis as an underlying objective of the problem. Tests on an axis are ordered with respect to increasing difficulty ($<$ relation), so that every candidate solution can be positioned on it according to the results of its interaction with these tests. The position of a candidate solution is precisely determined by the position function defined below.

Definition 23. *Position function* $p_i : S \to \bar{A}_i$ is a function that assigns a test from $\bar{A}_i$ to candidate solution $s \in S$ in the following way:

$$p_i(s) = \max\{t \in \bar{A}_i | G(s, t)\}, \tag{5.3.1}$$

where the maximum is taken with respect to the relation $<$. The test $p_i(s)$ is the *position* of s on the axis $\bar{A}_i$.

To give additional insight into the above definition, we show an important property of a coordinate system. Let $p_i(s) = t$. From the definition of position function p_i as the maximal test t for which $G(s, t)$, it follows immediately that $\neg G(s, t_1)$ for each $t_1 > t$. On the other hand, tests on the axis A_i are linearly ordered by the relation $<$, which means that for each $t_1, t_2 \in A_i$, $t_1 < t_2$ when $SF(t_1) \subset SF(t_2)$. Thus, according to the definition of $SF(t)$, $G(s, t_2)$ for each $t_2 < t$. Consequently, if $A_i = \{t_1 < t_2 < \cdots < t_{k_i}\}$ is an axis and $p_i(s) = t_j$, we can picture s's placement on A_i in the following way [17]:

$$\begin{array}{cccccccc} G(s,t) & 1 & 1 & \dots & 1 & 0 & \dots & 0 \\ \bar{A}_i & t_0 & t_1 & \dots & t_j & t_{j+1} & \dots & t_{k_i} \end{array}$$

As we can see, according to the position function, s is placed in such a way that for each axis it solves all tests on its left and fails all on its right.

Definition 24. The coordinate system $\mathcal{C}$ is *correct* for a test-based problem $\mathcal{G}$ iff for all $s_1, s_2 \in S$

$$s_1 \leq s_2 \iff \forall_{i \in I} p_i(s_1) \leq p_i(s_2).$$

Basically, this definition means that all relations between candidate solutions in set S have to be preserved by the coordinate system.

Notice also that in a correct coordinate system, $s_1 = s_2$ implies both $\forall_{i \in I} p_i(s_1) \leq p_i(s_2)$ and $\forall_{i \in I} p_i(s_2) \leq p_i(s_1)$, and consequently $\forall_{i \in I} p_i(s_1) = p_i(s_2)$. The proof of the converse implication is analogous. As a result, in the correct coordinate system for all $s_1, s_2 \in S$ we have

$$s_1 = s_2 \iff \forall_{i \in I} p_i(s_1) = p_i(s_2),$$

which means that two different candidate solutions never occupy the same position. Also

$$s_1 < s_2 \iff \forall_{i \in I} p_i(s_1) \leq p_i(s_2) \wedge \exists_{j \in I} p_j(s_1) < p_j(s_2)$$

and

$$s_1 \| s_2 \iff \exists_{i \in I} p_i(s_1) > p_i(s_2) \wedge \exists_{j \in I} p_j(s_1) < p_j(s_2).$$

Throughout this chapter, we will often ask about the relationship between two candidate solutions in a context of a test or some tests; thus the following two definitions will prove useful.

Definition 25. Test t *orders candidate solution* s_1 *before candidate solution* s_2, written $s_1 <_t s_2$, if $\neg G(s_1, t)$ and $G(s_2, t)$. Similarly, $s_1 =_t s_2$ when $G(s_1, t) = G(s_2, t)$, and $s_1 \leq_t s_2$ when $s_1 <_t s_2$ or $s_1 =_t s_2$.

This definition above resembles the Ficici's notion of *distinctions* [43]. Notice also that $s_1 <_t s_2$ implies $s_2 \not<_t s_1$. Obviously, $\forall_{t \in T} s_1 \leq_t s_2 \iff s_1 \leq s_2$.

We will also write $s_1 \leq_{\mathcal{C}} s_2$ to denote that $p_i(s_1) \leq p_i(s_2)$ holds for all $i \in I$ in $\mathcal{C}$. Similarly, we will use $s_1 <_{\mathcal{C}} s_2$, $s_1 =_{\mathcal{C}} s_2$ and $s_1 \|_{\mathcal{C}} s_2$.

The following simple proposition will allow us to rewrite the definition of correct coordinate system in an elegant way.

Proposition 26. *If* $\mathcal{C}$ *is a coordinate system, then for all* $s_1, s_2 \in S$

$$s_1 \leq_{\mathcal{C}} s_2 \iff \forall_{t \in \bigcup \mathcal{C}} s_1 \leq_t s_2.$$

Proof. ($\Rightarrow$) $s_1 \leq_{\mathcal{C}} s_2$ means $\forall_{i \in I} p_i(s_1) \leq p_i(s_2)$. Let $t_1 = p_i(s_1)$ and $t_2 = p_i(s_2)$; then $t_1 \leq t_2$. Def. (23) implies that

$$\forall_{t \leq t_1} G(s_1, t) \wedge \forall_{t > t_1} \neg G(s_1, t),$$

and

$$\forall_{t \leq t_2} G(s_2, t) \wedge \forall_{t > t_2} \neg G(s_2, t).$$

Consider three possible positions of t on axis $A_i \in \mathcal{C}$:

1. $t \leq t_1 \leq t_2$: $G(s_1, t) \wedge G(s_2, t)$,
2. $t_1 \leq t_2 < t$: $\neg G(s_1, t) \wedge \neg G(s_2, t)$,
3. $t_1 < t \leq t_2$: $\neg G(s_1, t) \wedge G(s_2, t)$.

Thus, for any $t \in \bigcup \mathcal{C}$, we have $s_1 \leq_t s_2$.

($\Leftarrow$) Suppose to the contrary that $\forall_{t \in \bigcup \mathcal{C}} s_1 \leq_t s_2 \wedge \neg(s_1 \leq_{\mathcal{C}} s_2)$. The right operand of this conjunction implies $\exists_{j \in I} p_j(s_1) > p_j(s_2)$. Let $t_1 = p_j(s_1)$ and $t_2 = p_j(s_2)$; thus $t_1 > t_2$. According to the definition of the position function (Def. 23), $G(s_1, t_1)$. Similarly $G(s_2, t_2)$, but we know that for any $t > t_2$, $\neg G(s_2, t)$. Since $t_1 > t_2$, we have $\neg G(s_2, t_1)$. It follows that $G(s_1, t_1) \wedge \neg G(s_2, t_1)$, thus $s_1 >_{t_1} s_2$; Since $t_1 \in \bigcup \mathcal{C}$, this contradicts our initial assumption that $\forall_{t \in \bigcup \mathcal{C}} s_1 \leq_t s_2$. □

The above proposition immediately leads to an alternative definition of correct coordinate system, which is equivalent to Def. 22.

Definition 27. The coordinate system $\mathcal{C}$ is correct for a test-based problem $\mathcal{G}$ iff for all $s_1, s_2 \in S$

$$s_1 \leq s_2 \iff \forall_{t \in \bigcup \mathcal{C}} s_1 \leq_t s_2. \tag{5.3.2}$$

Definition 28. A correct coordinate system $\mathcal{C}$ is a *minimal coordinate system* for $\mathcal{G}$ if there does not exist any correct coordinate system for $\mathcal{G}$ with smaller size.

Definition 29. The *dimension* $\dim(\mathcal{G})$ of a test-based problem $\mathcal{G}$ is the size of a minimal coordinate system for $\mathcal{G}$.

5.4. Example

Let us consider an exemplary test-based problem from [33], i.e., the misère version of game of Nim-1-3 with two piles of sticks: one containing a single stick and one containing

three sticks. The exact rules of this game are not important here, but an interested reader is referenced to [33].

Table 5.1.: The payoff matrix for Nim-1-3. An empty cell means 0.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
s_1	1		1	1		1	1		1
s_2									
s_3	1	1	1	1	1	1	1	1	1
s_4	1	1	1				1	1	1
s_5	1			1			1		
s_6							1	1	1

The payoff matrix of Nim-1-3 is shown in Table 5.1. There are a total of 144 strategies, but merging indiscernible strategies reduces the number of first player strategies to 6 (candidate solutions s_1-s_6) and second player strategies to 9 (tests t_1-t_9).

Figure 5.4.1 presents a minimal coordinate system for this game. We can see that the initial set of nine tests was "compressed" to only two underlying objectives represented by axes $A_1 = \{t_9 < t_8 < t_2\}$, $A_2 = \{t_4\}$. First, notice that tests on both axes are placed according to the definition of coordinate system, that is in the order of increasing difficulty (in A_1, t_9 is less difficult than t_8 that is, in turn, less difficult than t_2).

Second, the correctness of this coordinate system can be verified by checking whether all relations between pairs of candidate solutions are preserved (see conditions in Definition 24 or 27). For instance, consider a pair (s_1, s_3): $s_1 < s_3$ and s_1 is also dominated by s_3 in the 2D space; on the other hand, $s_1 \| s_6$ (since $s_1 <_{t_8} s_6$ and $s_6 <_{t_4} s_1$) and s_1, s_6 do not dominate each other also in the figure. Interestingly, only four tests out of nine were required to construct a coordinate system preserving all relations between candidate solutions from S.

Third, candidate solutions are placed in the example with accordance to the position function. Thus, for example, s_6 is placed so that it solves t_9 and t_8, but fails t_4 and t_2, which is consistent with the relations in the original payoff matrix.

With a little effort, one could also check that in this example $\text{width}(T, \leq) = 3$ and the minimum partition of $(T, \leq)$ consists of the following chains: $(t_9, t_3, t_6), (t_8, t_2, t_5), (t_7, t_1, t_4)$. Also, $\dim(S, \leq) = 2$, and $\dim(\mathcal{G}) = 2$

Now, let us consider removing t_9 from the horizontal axis. The resulting formal object is still a coordinate system, as the ordering of remaining tests, required by Def. 22, remains intact. However, it is incorrect, because s_1 shifts to the left and occupies the same position as s_5. This implies $s_1 = s_5$, which is inconsistent with the payoff matrix.

This helps understand the importance of correctness: an incorrect coordinate system does not reflect relations between candidate solutions and leads to essential information loss.

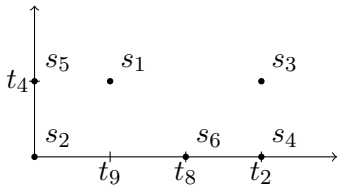


Figure 5.4.1.: A minimal coordinate system for Nim-1-3

In the context of this example, it is also worth emphasizing that the concept of coordinate system brings a new quality to test-based problems when compared to posets $(S, \leq)$ or $(T, \leq)$. Thanks to the position function, the coordinate system explicitly involves both candidate solutions and tests, while the poset describes the relations only within one object category (e.g., candidate solutions), with the other one (here: tests) hidden in relation $\leq$. By preserving all relations between candidate solutions and tests, a correct coordinate system unequivocally determines the result of interaction between every candidate solution and each test on any axis. This information cannot be restored from posets $(S, \leq)$ and $(T, \leq)$.

Finally, notice that if we had another candidate solution indiscernible with, for instance, s_6 (S is a preordered set), its position would be the same as s_6, and its existence would not change the coordinate system for the test-based problem in question. On the other hand, an additional test indiscernible with, for instance, t_8 is not needed in any axis and the coordinate system is correct without it. The latter will be further generalized in Proposition 34.

5.5. Properties of Coordinate Systems

In this section, we prove several facts about the coordinate system defined above. This will allow us to better understand this mathematical object, and, eventually, will help to design algorithms constructing a coordinate system for a given test-based problem.

Let us first observe that the definition of correct coordinate system (Def. 24) does not require all tests from set T to be used in the coordinate system.

Definition 30. Given a test-based problem $\mathcal{G} = (S, T, G)$, coordinate system $\mathcal{C}$ is *complete* if $\bigcup \mathcal{C} = T$.

As we have seen in Section 5.4, a correct system is not necessarily complete, however the inverse statement holds.

Proposition 31. *Every complete coordinate system is correct.*

Proof. If $\mathcal{C}$ is a complete coordinate system, then the condition (5.3.2) in Definition 27 is fulfilled, because $\forall_{t\in\bigcup\mathcal{C}} s_1 \leq_t s_2$ implies $s_1 \leq s_2$, as $\bigcup\mathcal{C} = T$. Therefore, by Definition 27, $\mathcal{C}$ is correct. □

Since not all tests from T are required to construct a minimal coordinate system, then it is natural to ask which tests are required and which are not. In the following we answer this question by proving that a test u can be safely removed from a correct coordinate system $\mathcal{C}$ if u orders only such pairs of candidate solutions that are also ordered by other tests from $\mathcal{C}$; and *vice versa, u* cannot be safely removed from $\mathcal{C}$ if it is the only test in $\mathcal{C}$ that orders a pair of candidate solutions. This is precisely expressed in the following theorem.

Theorem 32. *Let $\mathcal{C} = (A_i)_{i\in I}$ be a correct coordinate system. Let $\mathcal{C}'$ be a coordinate system resulting from removing a test u from some axis in $\mathcal{C}$. $\mathcal{C}'$ is a correct coordinate system iff*

$$\forall_{s_1,s_2\in S}\left(s_1 <_u s_2 \implies \exists_{t\in\bigcup\mathcal{C}'} s_1 <_t s_2\right). \tag{5.5.1}$$

Proof. First, observe that $\mathcal{C}'$ is a coordinate system since after removing u, all axes remain linearly ordered.

($\Rightarrow$) We will prove that if $\mathcal{C}'$ is correct then (5.5.1) is satisfied. Suppose to the contrary that $\mathcal{C}'$ is correct and (5.5.1) is not satisfied. This means that $\exists s_1, s_2 \in S$ such that $s_1 <_u s_2 \wedge \nexists_{t\in\bigcup\mathcal{C}'} s_1 <_t s_2$. It follows from $s_1 <_u s_2$ that $s_1 < s_2$ or $s_1 \parallel s_2$, thus $s_1 \not\geq s_2$. On the other hand, it follows from $\nexists_{t\in\bigcup\mathcal{C}'} s_1 <_t s_2$ that $s_1 \geq_{\mathcal{C}'} s_2$; subsequently, since $\mathcal{C}'$ is correct, $s_1 \geq s_2$, which contradicts the earlier statement.

($\Leftarrow$) Assume that (5.5.1) is satisfied. By contraposition, we have

$$\forall_{s_1,s_2\in S}\left(\neg\exists_{t\in\bigcup\mathcal{C}'} s_1 <_t s_2 \implies \neg s_1 <_u s_2\right),$$

which, after swapping s_1 with s_2, can be written as

$$\forall_{s_1,s_2\in S}\left(\forall_{t\in\bigcup\mathcal{C}'} s_1 \leq_t s_2 \implies s_1 \leq_u s_2\right). \tag{5.5.2}$$

Because $\mathcal{C}$ is correct, it follows from Definition 27 that for all $s_1, s_2 \in S$

$$s_1 \leq s_2 \iff \left(\forall_{t\in\bigcup\mathcal{C}'} s_1 \leq_t s_2\right) \wedge s_1 \leq_u s_2.$$

Whenever $s_1 \leq_u s_2$ is true, it can be ignored in the above condition. Otherwise, by (5.5.2), $\forall_{t \in \bigcup \mathcal{C}'} s_1 \leq_t s_2$ must be also false. Therefore, the above reduces to

$$s_1 \leq s_2 \iff \forall_{t \in \bigcup \mathcal{C}'} s_1 \leq_t s_2.$$

which, by Definition 27, implies that $\mathcal{C}'$ is correct. □

An obvious consequence of the above theorem is following.

Corollary 33. *The coordinate system $\mathcal{C}$ is correct for test-based problem $\mathcal{G} = (S, T, G)$ if and only if it preserves all relations between tests in T, i.e.,*

$$\forall_{s_1, s_2 \in S} \left(\exists_{t_1 \in T} s_1 <_{t_1} s_2 \implies \exists_{t_2 \in \bigcup \mathcal{C}} s_1 <_{t_2} s_2 \right). \tag{5.5.3}$$

Let us consider a special case of Theorem 32 when $u \in A_a$ and $u \in A_b$, $a \neq b$. If we remove u from one axis in $\mathcal{C}$, u will still remain in the other, thus $u \in \mathcal{C}'$ and the condition (5.5.1) will hold regardless of the existence of other tests. Thus, Theorem 32 implies the following proposition.

Proposition 34. *Let $\mathcal{C} = (A_i)_{i \in I}$ be a correct coordinate system. If a test t lies on two different axes, i.e., $t \in A_a$ and $t \in A_b$, $a \neq b$, we can remove it from one of the axes and the coordinate system will remain correct.*

A coordinate system that does not contain any test lying on two axes, will be called *non-redundant*. Notice that removing any test from a coordinate system does not increase its size, which leads to an obvious conclusion:

Corollary 35. *For any test-based problem, there exists a minimal coordinate system that is non-redundant. Thus, in order to find the dimension of $\mathcal{G}$, it is enough to search the space of correct non-redundant coordinate systems.*

Another important observation is that the size of a minimal coordinate system $\mathcal{C}$ is equal to the width of the partially ordered set consisting of all tests of $\mathcal{C}$. This is expressed formally as the following theorem

Theorem 36. *Let $\mathcal{C}$ be a minimal coordinate system for $\mathcal{G}$. Let $U = \bigcup \mathcal{C}$. Then*

$$\dim(\mathcal{G}) = \operatorname{width}(U, \leq). \tag{5.5.4}$$

Proof. By definition of axis, each $(A_i, \leq)$, where $A_i \in \mathcal{C}$, $i \in I$, is a chain. $\mathcal{C}$ is a minimal coordinate system, so by Proposition 34, we can assume without loss of generality that

each test occurs in at most one axis in $\mathcal{C}$. Hence, $\mathcal{C}$ is a chain partition of $(U, \leq)$. According to Dilworth (Theorem 7), width$(U, \leq)$ is the number of chains in the minimum chain partition of $(U, \leq)$, thus $|\mathcal{C}|$ is at least width$(U, \leq)$; therefore,

$$|\mathcal{C}| = \dim(\mathcal{G}) \geq \text{width}(U, \leq). \tag{5.5.5}$$

Now, let $\mathcal{D} = (D_i)_{i=1\ldots n}$ be a minimum partition of $(U, \leq)$ into chains, hence, by Def. 27, for all $s_1, s_2 \in S$

$$\forall_{t \in \bigcup \mathcal{D}} s_1 \leq_t s_2 \iff \forall_{t \in \bigcup \mathcal{C}} s_1 \leq_t s_2 \iff s_1 \leq s_2,$$

because $\bigcup \mathcal{D} = U = \bigcup \mathcal{C}$. Thus, again by Def. 27, $\mathcal{D}$ is a correct coordinate system and, as its size is width$(\mathrm{U}, \leq)$, we get

$$\dim(\mathcal{G}) \leq \text{width}(\mathrm{U}, \leq). \tag{5.5.6}$$

Combining inequalities (5.5.5) and (5.5.6) finishes the proof. □

Using the same reasoning as in the second part of the above proof, we could also show the correctness of a similar statement concerning correct, but not necessarily minimal, coordinate systems:

Proposition 37. *Let $\mathcal{C}'$ be a correct coordinate system for $\mathcal{G}$ and $U' = \bigcup \mathcal{C}'$. Then*

$$\dim(\mathcal{G}) \leq \text{width}(U', \leq). \tag{5.5.7}$$

For a brief demonstration, consider the following example. Let $\mathcal{G}$ be a test-based problem defined by the following matrix:

	t_1	t_2	t_3	t_4	t_5
s_1	1	1	0	0	1
s_2	1	0	1	0	0
s_3	0	0	1	1	1

A minimal coordinate system for this test-based problem is $\mathcal{C} = (\{t_1 < t_2\}, \{t_3 < t_4\})$, hence $\dim(\mathcal{G}) = \text{width}\left(\bigcup \mathcal{C}\right) = 2$. The coordinate system $\mathcal{C}' = (\{t_1 < t_2\}, \{t_3 < t_4\}, \{t_5\})$ is also correct, but the poset built from all its positions, i.e., $\{t_1 < t_2;\ t_3 < t_4;\ t_5\}$, has width 3.

Notice that the width of a poset is monotonic regarding its ground set. Thus, we can formulate an obvious remark.

Remark 38. Given a poset (X, P), for any $x \in X$,

$$\text{width}(X \setminus \{x\}, P) \leq \text{width}(X, P).$$

The above remark together with Theorem 36 lead. to the following statement.

Corollary 39. *In order to determine the dimension of $\mathcal{G} = (S, T, G)$ it is enough to find a poset of minimal width whose ground set is a minimal subset of T producing a correct coordinate system.*

It is interesting to determine the upper and lower bound for $\dim(\mathcal{G})$.

Theorem 40. *For every test-based problem $\mathcal{G} = (S, T, G)$,*

$$\dim(S, \leq) \leq \dim(\mathcal{G}) \leq \text{width}(T, \leq). \tag{5.5.8}$$

Proof. First we prove that $\dim(S, \leq) \leq \dim(\mathcal{G})$. Let $\mathcal{C} = (A_1, A_2, \ldots, A_n)$ be a minimal coordinate system for $\mathcal{G}$. We will construct a family $\mathcal{R} = (L_1, L_2 \ldots, L_n)$ of linear orders on S, such that $\bigcap \mathcal{R} = \leq$, where $\leq$ is the weak dominance relation between elements of S (see Def. 21). Let L_i, a linear extension of $\leq$, be defined as

$$L_i = \{(s_1, s_2) | p_i(s_1) \leq p_i(s_2)\}.$$

Consider an ordered pair of candidate solutions $(s_1, s_2) \in S \times S$ that is an element of $\bigcap \mathcal{R}$. By definition of $\mathcal{R}$ we have $(s_1, s_2) \in \bigcap \mathcal{R} \iff \forall_i (s_1, s_2) \in L_i$ and the latter is equivalent to $\forall_i p_i(s_1) \leq p_i(s_2)$. Coordinate system $\mathcal{C}$ is correct, thus $\forall_i p(s_1) \leq p(s_2)$ if and only if $s_1 \leq s_2$, which we can write as $(s_1, s_2) \in \leq$. Hence, $(s_1, s_2) \in \bigcap \mathcal{R}$ is equivalent to $(s_1, s_2) \in \leq$ and we finally get $\bigcap \mathcal{R} = \leq$. Therefore, since $\mathcal{C}$ is a minimal coordinate system, it must hold that $\dim(S, \leq) \leq \dim(\mathcal{G})$.

Next, we prove that $\dim(\mathcal{G}) \leq \text{width}(T, \leq)$. Let $\mathcal{C} = (A_i)_{i \in I}$ be a minimal coordinate system for $\mathcal{G}$. Let $U = \bigcup \mathcal{C}$. Obviously, $U \subseteq T$, thus $\text{width}(U, \leq) \leq \text{width}(T, \leq)$. By Theorem 36, $\dim(\mathcal{G}) = \text{width}(U, \leq)$, hence we have $\dim(\mathcal{G}) \leq \text{width}(T, \leq)$. □

Unfortunately, the problem of computing the lower bound $\dim(\text{S}, \leq)$ is NP-hard [146] and the question whether there exists any polynomial-time algorithm computing a reasonable lower bound of $\dim(\mathcal{G})$ remains open. On the other hand, the problem of determining $\text{width}(T, \leq)$ is easy (see Section 5.8.2), but, as we will show, better approximations of the upper bound exist (cf. Section 5.8.3).

5.6. Finite and Infinite Test-Based Problems

Until now, we considered only finite test-based problems, i.e., test-based problems where S and T were finite. Here we relax this restriction and consider how the notion of coordinate system behaves when S and T are infinite[1]. Let us emphasize that this distinction does not correlate with test-based problem complexity as perceived by humans: the game of chess is finite, although the number of strategies for both players is very large. On the other hand, the space of strategies in the conceptually trivial Numbers Games, such as these considered in Section 5.9, may be infinite (uncountable, in this case).

According to the definition of position function in Section 23, a correct coordinate system is well-defined as long as for each candidate solution in S, the max operator is well-defined. For all countable test-based problems, max will work, but it may not work for some uncountable test-based problems, when, for example, the set of tests solved by a candidate solution does not have the maximal element. Thus, the definition of coordinate system needs a generalization that will cope with such cases.

Bucci et al. [17] proved that every finite test-based problem has a dimension, thus, for every finite S and T, there exists a minimal coordinate system. It is not entirely naive to ask whether there exist highly-dimensional test-based problems. Could it possibly be the case that $\dim(\mathcal{G}) \leq 16$, for all test-based problems $\mathcal{G}$? The following example settles this issue.

Example 41. Consider a test-based problem $\mathcal{G}(n) = (S, T, G)$, defined in the following way:

$$S = (s_i)_{i=1\ldots n},$$
$$T = (t_j)_{j=1\ldots n},$$
$$G(s_i, t_j) \iff i = j.$$

Since all tests are mutually incomparable and each test orders a unique pair of candidate solutions, the dimension of $\mathcal{G}(n)$ is n.

Thus, for each n there exists a test-based problem of dimension n. Moreover, if we consider the same example, but assume that S and T are infinite, the dimension of $\mathcal{G}$ is not limited by any number, so it does not exist. We will denote such a situation as $\dim(\mathcal{G}) = \infty$.

We already know that when S and T are infinite, the test-based problem could have no dimension, but are all infinite test-based problems dimensionless? In the following example, we show that $\mathcal{G}$ can have a dimension even when S and T are infinite.

[1]Since we assumed that indiscernible elements do not exist in T and S, if a test-based problem is infinite, both T and S have to be infinite; otherwise, both have to be finite.

Example 42. Consider an infinite test-based problem $\mathcal{G}(n) = (S, T, G)$, such that

$$S = (s_i),$$
$$T = (t_j),$$
$$G(s_i, t_j) \iff i \geq j.$$

The dimension of this test-based problem is 1, because all tests can be placed on one axis $A_1 = (t_1 < t_2 < \dots)$. Then, $p_1(s_i) = t_i$.

Corollary 43. *When a test-based problem is finite, it always has a dimension; when a test-based problem is infinite, it may or may not have a dimension.*

5.7. Hardness of the Problem

Dimension is an important property of a test-based problem, thus it is natural to ask how hard it is to compute it for a given problem instance. Here we prove that this task is NP-hard. To this aim, let us formally define the decision version of the Dimension Problem.

Problem 44. (Dimension Problem) Given a test-based problem $\mathcal{G} = (S, T, G)$, where S and T are finite and a positive integer n, does a correct coordinate system $\mathcal{C}$ for test-based problem $\mathcal{G}$ of size n or less exist?

We will prove the NP-completeness of the above problem using *the Set Covering Problem* as the reference problem.

Problem 45. (Set Covering Problem) Given a universe $\mathcal{U} = (u_i)$, a family $\mathcal{R} = (R_j)$ of subsets of $\mathcal{U}$, a *cover* is a subfamily $\mathcal{V} \subseteq \mathcal{R}$ of sets whose union is $\mathcal{U}$. Given $\mathcal{U}$ and $\mathcal{R}$ and an integer m, the question is whether there exists a cover of size m or less.

Denote the size of $\mathcal{U}$ by u and the size of $\mathcal{R}$ by r.

The Set Covering Problem is NP-complete, which was proved by Karp [71]. Here we slightly narrow its domain by assuming that $\bigcup \mathcal{R} = \mathcal{U}$ and $u > 2$ and $r > 2$, calling it *Narrowed Set Covering Problem.* The Narrowed Set Covering Problem is also NP-complete. First, the limits on u and r do not ease the problem. Second, the answer to the Set Covering Problem with $\bigcup \mathcal{R} \subset \mathcal{U}$ is *no* regardless of the value of m. It is trivial to check whether $\bigcup \mathcal{R} = \mathcal{U}$, thus this modification does not change the hardness of the problem, either.

Theorem 46. *The Dimension Problem is NP-complete.*

Proof. Denote our decision problem by π_1. Let D_π denote the domain of problem π. First, notice that, according to Def. 27, in order to check whether the answer for a certain coordinate system $\mathcal{C}$ is *yes*, we need $O(|S|^2\,|T|)$ operations. Clearly then $\pi_1 \in NP$.

Second, we will show that π_2 is reducible to π_1 in a polynomial time, where π_2 is the Narrowed Set Covering Problem,

Given an instance $I_2 \in D_{\pi_2}$, we construct $I_1 \in D_{\pi_1}$ in the following way:

1. $n = m + u + r$;
2. $S = \{s_0\} \cup A \cup B \cup C \cup D$, where
   $A = (a_i)_{i=1\ldots u}$, $B = (b_i)_{i=1\ldots r}$, $C = (c_i)_{i=1\ldots u}$, and $D = (d_i)_{i=1\ldots r}$ are such sets that $|S| = 1 + 2r + 2u$;
3. $T = X \cup Y \cup Z$, where
   $X = (x_i)_{i=1\ldots r}$, $Y = (y_i)_{i=1\ldots u}$, and $Z = (z_i)_{i=1\ldots r}$, are such sets that $|T| = 2r + u$;
4. $\mathcal{G}$ is defined as follows:
   $$\begin{cases} G(s_0, y_i) \\ G(a_i, x_j) \iff u_i \in R_j \\ (G(a_i, y_j) \iff i \neq j) \text{ and } (G(b_i, z_j) \iff i \neq j) \\ (G(b_i, x_j) \iff i = j) \text{ and } (G(c_i, y_j) \iff i = j) \text{ and } (G(d_i, z_j) \iff i = j)\,. \end{cases}$$

It is easy to show that the construction of I_1 is limited from above by a polynomial function of the size of I_2. The following observations (see Example 48) should help comprehend the reasons for such design:

1. Notice that $G(a_i, x_j) \iff u_i \in R_j$ means that A corresponds to $\mathcal{U}$, X corresponds to $\mathcal{R}$ and the payoff submatrix $A \times X$ describes the elements of $\mathcal{R}$.
2. In instance I_1, all tests from T are mutually incomparable due to the 'diagonal' going through B, C, D and X, Y, Z; therefore any correct coordinate system $\mathcal{C}$ for test-based problem $\mathcal{G}$ will have each of its axes contain exactly one test, thus $n = |\mathcal{C}| = |\bigcup \mathcal{C}|$.
3. Any correct coordinate system for test-based problem $\mathcal{G}$ has to contain all elements from sets Y and Z, since they are indispensable to make some pairs from set S covered (see Example 48). Notice that if the submatrix $A \times X$ contained only zeros, then the coordinate system where $\bigcup \mathcal{C} = Y \cup Z$ would be correct, since all pairs from S would be ordered. On the other hand, thanks to the intricate construction of the payoff matrix, the only pairs of elements of S which must be ordered with

elements of set X, are pairs of the form (s_0, a_i), $i = 1 \dots u$. This will be formally shown later by analyzing all pairs of elements of S.

($I_2 \Rightarrow I_1$) Suppose that for $I_2 \in D_{\pi_2}$ the answer is *yes.* It means that there exists a cover $\mathcal{V} \subseteq \mathcal{R}$ of size m such that $\bigcup \mathcal{V} = \mathcal{U}$. Under this assumption, consider a coordinate system $\mathcal{C}$ such that

$$\bigcup \mathcal{C} = Y \cup Z \cup \{x_j \in X | R_j \in \mathcal{V}\}. \tag{5.7.1}$$

We will show that $\mathcal{C}$ is correct and its size $n = u + r + m$, thereby proving that the answer to I_1 is also *yes.*

First, notice that $|\bigcup \mathcal{C}| = u + r + m$. Thus $n = |\mathcal{C}| = |\bigcup \mathcal{C}| = u + r + m$.

Second, in order to show that $\mathcal{C}$ is correct, we will prove that condition (5.5.3) from Corollary (33) is satisfied for all pairs of candidate solutions $s_1, s_2 \in S$. We rewrite the condition below:

$$\exists_{t_1 \in T} s_1 <_{t_1} s_2 \implies \exists_{t_2 \in \bigcup \mathcal{C}} s_1 <_{t_2} s_2.$$

Let us consider all ordered pairs:

$s_0 <_t a_i$ does not hold for any $t \in Y \cup Z$, but it holds for all $t \in Q$ such that $Q = \{x_j | G(a_i, x_j)\} = \{x_j | u_i \in R_j\}$. From $\bigcup \mathcal{V} = \mathcal{U}$ it follows that

$$\forall_{u_p \in \mathcal{U}} \exists_{R_k \in \mathcal{R}} u_p \in R_k \wedge R_k \in \mathcal{V},$$

which in particular is true for our u_i, thus we have

$$\exists_{R_k \in \mathcal{R}} u_i \in R_k \wedge R_k \in \mathcal{V}.$$

And since the set of tests X corresponds to the set $\mathcal{R}$, we get

$$\exists_{x_k \in X} u_i \in R_k \wedge R_k \in \mathcal{V}.$$

The left operand of this conjunction implies that $x_k \in Q$. On the other hand, from the right operand, it follows that $x_k \in \bigcup \mathcal{C}$ (by (5.7.1)). Replacing the symbol x_k by t, we finally get

$$\exists_{t \in X} t \in Q \wedge t \in \bigcup \mathcal{C}.$$

Therefore, if $s_0 <_t a_i$ then $t \in \bigcup \mathcal{C}$, thus $\exists_{t_2 \in \bigcup \mathcal{C}} s_1 <_{t_2} s_2$ and (5.5.3) is satisfied.

$s_0 <_t b_i$ holds for $t = x_i$, but it is also true for $t \in \{z_k | k \neq i\}$, which is a non-empty set for $r > 1$; and since $Z \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$s_0 <_t c_i$ never holds, thus it can be ignored.

$s_0 <_t d_i$ holds only for $t = z_i$; and since $Z \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$a_i <_t s_0$ holds only for $t = y_i$; and since $Y \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$a_i <_t a_j$ can hold for $t = x_k$ for some k, but it is also true for $t = y_i$; and since $Y \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$a_i <_t b_j$ can hold for $t = x_k$ for some k, but it is also true for $t \in \{z_h | h \neq j\}$, which is a non-empty set for $r > 1$; and since $Z \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$a_i <_t c_j$ holds only for $i = j$ and $t = y_i$; and since $Y \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$a_i <_t d_j$ holds only for $t = z_j$; and since $Z \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$b_i <_t s_0$ holds for all $t \in Y$; and since $Y \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$b_i <_t a_j$ can hold for $t = x_k$ for some k, but it is also true for $t \in \{y_h | h \neq j\}$, which is a non-empty set for $u > 1$; and since $Y \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$b_i <_t b_j$ holds for $t = x_j$, but it is also true for $t = z_i$; and since $Z \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$b_i <_t c_j$ holds only for $t = y_j$; and since $Y \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$b_i <_t d_j$ holds only for $i = j$ and $t = z_i$; and since $Z \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$c_i <_t s_0$ never holds, thus it can be ignored.

$c_i <_t a_j$ can hold for $t = x_k$ for some k, but it also holds for $t \in \{y_h | h \neq i, h \neq j\}$, which is a non-empty set for $u > 2$; and since $Y \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$c_i <_t b_j$ holds for $t = x_j$, but it also holds for $t \in \{z_k | k \neq i\}$, which is a non-empty set for $r > 1$; and since $Z \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$c_i <_t c_j$ holds only for $t = y_j$; and since $Y \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$c_i <_t d_j$ holds only for $t = z_j$; and since $Z \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$d_i <_t s_0$ holds for $t \in Y$; and since $Y \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$d_i <_t a_j$ can hold for $t = x_k$ for some k, but it is also true for $t \in \{y_h | h \neq j\}$, which is a non-empty set for $u > 1$; and since $Y \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$d_i <_t b_j$ holds for $t = x_j$, but it also holds for $t \in \{z_h | h \neq i, h \neq j\}$, which is a non-empty set for $r > 2$; and since $Z \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$d_i <_t c_j$ holds only for $t = y_j$; and since $Y \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

$d_i <_t d_j$ holds only for $t \in \{z_k | k \neq j\}$; and since $Z \subseteq \bigcup \mathcal{C}$, (5.5.3) is satisfied.

It has been shown above that (5.5.3) is satisfied for all pairs of tests; therefore, according to Corollary (33), $\mathcal{C}$ is correct, thus the answer for instance I_1 is *yes*.

($I_1 \Rightarrow I_2$) Suppose that for $I_1 \in D_{\pi_1}$ the answer is *yes*. It means that there exists a correct coordinate system $\mathcal{C}$ of size n for test-based problem $\mathcal{G} = (S, T, G)$. Consider a cover $\mathcal{V} = \{R_j \in \mathcal{R} | x_j \in \bigcup \mathcal{C}\}$, whose size is $|\mathcal{V}| = |\{x \in X | x \in \bigcup \mathcal{C}\}|$. In order to prove that the answer to I_2 is *yes*, we will show that $\bigcup \mathcal{V} = \mathcal{U}$ and $|\mathcal{V}| \leq m$, where m fulfills the equation $n = m + r + u$.

$\mathcal{C}$ is correct so the condition (5.5.3) is satisfied, i.e.,

$$\forall_{s_1, s_2 \in S} \left(\exists_{t_1 \in T} s_1 <_{t_1} s_2 \implies \exists_{t_2 \in \bigcup \mathcal{C}} s_1 <_{t_2} s_2 \right).$$

In particular, it has to be true for a pair (s_0, a_i), where $a_i \in A$; thus we get

$$\forall_{a_i \in A} \left(\exists_{t_1 \in T} s_0 <_{t_1} a_i \implies \exists_{t_2 \in \bigcup \mathcal{C}} s_0 <_{t_2} a_i \right).$$

Recall that $s_0 <_t a_i$ may hold only when $t \in X$, that is

$$\forall_{a_i \in A} \left(\exists_{x_k \in X} s_0 <_{x_k} a_i \implies \exists_{x_j \in \bigcup \mathcal{C}} s_0 <_{x_j} a_i \right).$$

By definition, $s_0 <_x a_i$ means $\neg G(s_0, x) \wedge G(a_i, x)$ and since the left operand of this conjunction is true for all $x \in X$, the expression implies $G(a_i, x)$, thus

$$\forall_{a_i \in A} \left(\exists_{x_k \in X} G(a_i, x_k) \implies \exists_{x_j \in \bigcup \mathcal{C}} G(a_i, x_j) \right).$$

Since $G(a_i, x_j) \iff u_i \in R_j$ and the fact that the elements of A correspond to the elements of $\mathcal{U}$ and the elements of X correspond to the elements of $\mathcal{R}$, we can rewrite the above as

$$\forall_{u_i \in \mathcal{U}} \left(\exists_{R_k \in \mathcal{R}} u_i \in R_k \implies \exists_{x_j \in \bigcup \mathcal{C}} u_i \in R_j \right).$$

In the Narrowed Set Covering Problem, $\bigcup \mathcal{R} = \mathcal{U}$, so the left hand side of the implication always holds, thus

$$\forall_{u_i \in \mathcal{U}} \exists_{x_j \in \bigcup \mathcal{C}} u_i \in R_j.$$

From the definition of $\mathcal{V}$, it follows that $x_j \in \bigcup \mathcal{C} \iff R_j \in \mathcal{V}$ so we finally have

$$\forall_{u_i \in \mathcal{U}} \exists_{R_j \in \mathcal{V}} u_i \in R_j,$$

which is an equivalent to $\bigcup \mathcal{V} = \mathcal{U}$.

Now, in order to compare the size of cover $\mathcal{V}$ with m, we will use the following fact. Consider tests from sets Y and Z. Observe (cf. Example 48) that y_i is the only test for which $b_1 <_{y_i} c_i$, for $i = 1 \ldots u$. Similarly, $c_1 <_{z_i} d_i$ for $i = 1 \ldots u$ and z_i is the only test for which $c_1 <_{z_i} d_i$, for $i = 1 \ldots r$. Thus, according to condition (5.5.3), since $\mathcal{C}$ is correct, $\bigcup \mathcal{C}$ must contain all elements from Y and Z, i.e., $Y \cup Z \subseteq \bigcup \mathcal{C}$.

So, $n = u + r + |\{x \in X | x \in \bigcup \mathcal{C}\}|$, which boils down to $n = u + r + |\mathcal{V}|$; and since we know that $n = u + r + m$, we eventually have $|\mathcal{V}| = m$.

Since $\bigcup \mathcal{V} = \mathcal{U}$ and $|V| \leq m$, the answer to instance I_2 is *yes.* □

The Dimension Problem is NP-complete, thus:

Corollary 47. *Finding a minimal coordinate system for a test-based problem is NP-hard.*

Example 48. This example shows the construction of test-based problem $\mathcal{G}$. Let $\mathcal{U} = \{1, 2, 3, 4\}$, $\mathcal{R} = \{R_1, R_2, R_3\}$, $R_1 = \{1, 2\}$, $R_2 = \{1, 3\}$, $R_3 = \{1, 3, 4\}$. Then we can present the relation G of test-based problem $\mathcal{G}$ graphically (empty cells mean 0):

	x_1	x_2	x_3	y_1	y_2	y_3	y_4	z_1	z_2	z_3
s_0				1	1	1	1			
a_1	1	1	1		1	1	1			
a_2	1			1		1	1			
a_3		1	1	1	1		1			
a_4			1	1	1	1				
b_1	1								1	1
b_2		1						1		1
b_3			1					1	1	
c_1				1						
c_2					1					
c_3						1				
c_4							1			
d_1								1		
d_2									1	
d_3										1

Notice that most of the pairs to be ordered according to condition (5.5.3) from Corollary 33 can be ordered by elements from either Y or Z, for instance, $a_2 <_{y_2} a_3$, $a_4 <_{y_4} s_0$ or $b_1 <_{z_1} b_2$. The only pairs of elements of S that must be ordered by elements of set X are pairs (s_0, a_1), (s_0, a_2), (s_0, a_3), and (s_0, a_4), which correspond to the universe $\mathcal{U}$ of the set covering problem.

5.8. Algorithms

In this section, we show three algorithms that construct correct coordinate systems.

5.8.1. Simple Greedy Heuristic

The first algorithm (here called SimpleGreedy) for coordinate system extraction was given in [17]. SimpleGreedy finds a correct coordinate system, but it does not guarantee finding a minimal one.

The pseudocode of SimpleGreedy is shown as Algorithm 5.1. In the first stage, the algorithm removes from T the tests that are combinations of two other tests in the sense of candidate solutions they fail (lines 2-7). In the second stage, it greedily finds an axis to place a test on, ensuring that at every step the axes remain linearly ordered (line 12). To this aim, it first tries to place a test on an existing axis (lines 11-17); if this

Algorithm 5.1 SimpleGreedy heuristic for extracting a minimal coordinate system.

```
 1: procedure SimpleGreedy(S, T, G)
 2:     U ← T
 3:     for distinct t, t1, t2 ∈ U do
 4:         if SF(t) = SF(t1) ∪ SF(t2) then
 5:             T ← T \ {t}
 6:         end if
 7:     end for
 8:     C ← ∅
 9:     for t ∈ T sorted ascendingly by |SF(t)| do
10:         found ← false
11:         for A_i ∈ C do
12:             if max((A_i, ≤)) < t then
13:                 A_i ← A_i ∪ {t}                    ▷ Add t to existing axis
14:                 found ← true
15:                 break
16:             end if
17:         end for
18:         if !found then
19:             C ← C ∪ {{t}}                          ▷ Create a new axis
20:         end if
21:     end for
22:     return C
23: end procedure
```

is impossible, it creates a new axis (lines 18-20). Tests are considered in the ascending order with respect to the number of candidate solutions they fail (line 9), so that the "best-performing" tests are placed at the end. Note that the poset $(A_i, \leq)$ in line 12 is a non-empty chain, so $\max(A_i, \leq)$ contains exactly one test, which is being compared with t.

In [17] there was no formal proof that SimpleGreedy is correct, so we provide it here.

Proposition 49. *For a given test-based problem $\mathcal{G}$,* SimpleGreedy *algorithm produces a correct coordinate system $\mathcal{C}$.*

Proof. Observe that skipping the first stage of Algorithm 5.1 results in a complete coordinate system, which must be correct by Proposition 31. Thus, it is enough to show that removing a test t such that $SF(t) = SF(t_1) \cup SF(t_2)$, where t, t_1, t_2 are distinct and $t, t_1, t_2 \in \bigcup \mathcal{C}$, preserves the correctness of $\mathcal{C}$. From $SF(t) = SF(t_1) \cup SF(t_2)$ it follows

that

$$s \in SF(t) \implies s \in SF(t_1) \vee s \in SF(t_2) \tag{5.8.1}$$

and

$$s \notin SF(t) \implies s \notin SF(t_1) \wedge s \notin SF(t_2). \tag{5.8.2}$$

Now, consider a pair $s_1, s_2 \in S$ such that $s_1 <_t s_2$; therefore $s_1 \in SF(t)$ and $s_2 \notin SF(t)$. From the former, by (5.8.1) and without loss of generality, it follows that $s_1 \in SF(t_1)$. On the other hand, the latter implies $s_2 \notin SF(t_1)$ (by (5.8.2)). Therefore, $\neg G(s_1, t_1)$ and $G(s_2, t_1)$, which implies $s_1 <_{t_1} s_2$. Since $t_1 \neq t$, test t_1 orders s_1 before s_2. Thus, according to Theorem 32, we can safely remove t and $\mathcal{C}$ will remain correct.

To complete the proof, notice that the tests can be removed from $\mathcal{C}$ in a top-down order, i.e., we remove test t_1 from the coordinate system $\mathcal{C}$ only when there do not exist any t_2, t_3 such that $SF(t_1) \cup SF(t_2) = SF(t_3)$, where t_1, t_2, t_3 are distinct elements of $\bigcup \mathcal{C}$; otherwise we remove t_3 first. In this way we guarantee that all tests t such that $SF(t) = SF(t_1) \cup SF(t_2)$, where t, t_1, t_2 are distinct elements of T, will be removed, as it is the case in the algorithm. □

SimpleGreedy is fast; its worst case time complexity is $O(|T|^3|S|)$, because of the bottleneck in lines 3-7.

5.8.2. The Exact Algorithm

In the following we propose an exact algorithm (Exact for short) that constructs a minimal coordinate system for a given test-based problem and thus determines its dimension. As we have proved that, unless N = NP, there does not exist any exact, polynomial-time algorithm, Exact has exponential time complexity. Despite this, we wanted it to be as fast as possible, that is why we founded it on three results proved earlier in this chapter:

1. Corollary 35, which says that it is enough to consider the coordinate systems in which every test appears on at most one axis. We use that result to initialize the search in our algorithm.

2. Theorem 32, which determines which tests can be safely removed from T driving our algorithm.

3. Corollary 39, which implies that the minimal coordinate system for $\mathcal{G}$ must be among chain partitions of $(L, \leq)$, where L is such a subset of T producing a correct coordinate system that we cannot safely remove any test from L.

Algorithm 5.2 EXACT algorithm for extracting a minimal coordinate system.

```
1: procedure Exact(S, T, G)
2:     U ← AllMinimalSubsets(T, T)          ▷ Find all minimal correct subsets of T
3:     C ← ∞
4:     for U ∈ U do                          ▷ Find a poset of minimal width
5:         C' ← ChainPartition(U, ≤)
6:         if |C'| < |C| then
7:             C ← C'
8:         end if
9:     end for
10:    return C
11: end procedure

12: procedure AllMinimalSubsets(L, R)
13:    U ← ∅
14:    isLeaf ← true                         ▷ Is L a leaf in the recursion tree?
15:    for t ∈ R do                          ▷ We visit every subset L of set T at most once...
16:        R ← R \ {t}
17:        if CanBeRemoved(t, L) then        ▷ ...taking into account correct subsets only
18:            isLeaf ← false
19:            L ← L \ {t}
20:            AllMinimalSubsets(L, R)
21:            L ← L ∪ {t}
22:        end if
23:    end for
       ▷ L, a Leaf in the recursion tree is a correct subset, but not necessarily a minimal
       one, which we need still to check
24:    if isLeaf and ∄_{t∈L} CanBeRemoved(t, L) then
25:        U ← U ∪ {L}                       ▷ L is a minimal correct subset
26:    end if
27:    return U
28: end procedure
29:
30: procedure CanBeRemoved(u, Q)
31:    return ∀_{s1,s2∈S} (s1 <_u s2 ⟹ ∃_{t∈Q\u} s1 <_t s2)        ▷ Condition (5.5.1)
32: end procedure
33:
34: procedure ChainPartition(X, P)
35:    return a minimal partition of poset (X, P) into chains.
36: end procedure
```

Exact is shown in Algorithm 5.2 and it works as follows. In the first stage (line 2), the algorithm computes $\mathcal{U}$, the family of all *minimal correct subsets* of T, i.e., such sets $T' \subseteq T$ that there exists such a correct coordinate system $\mathcal{C}$ that $\bigcup \mathcal{C} = T'$ and no $T'' \subset T'$ with this property exists. The procedure AllMinimalSubsets recursively visits the subsets of T, and whenever it finds a minimal one, it appends it to $\mathcal{U}$ (line 25) and continues the search. Its recursion tree is consistent with subset inclusion, i.e., the recursive calls visit the subsets of the current set. Testing whether t can be removed from L (procedure CanBeRemoved) relies on Theorem 32. By maintaining the set R of tests not yet considered for removal at a given level of recurrence (lines 15-16), the procedure never visits any subset of T twice. Recursion returns when no more tests can be removed and some parts of the search tree are never considered. Note that reaching a leaf of the recursion tree (detected using the *isLeaf* flag) does not guarantee that L is a minimal correct subset: we still have to check individual elements of L for removal (line 24). This method of visiting all subsets satisfying certain requirements is similar to the one described by de la Banda et al. [35].

In the second stage, the algorithm computes a chain partition $\mathcal{C}$ for every element of $\mathcal{U}$, that is, the coordinate system (see the proof of Theorem 36), and returns the one with a minimal number of axes $|\mathcal{C}|$. Partitioning a poset (P, X), $|X| = n$ into chains can be solved in $O(n^3)$ [99, 39] using a max-flow computation on a bipartite network with unit capacities. This result can be further improved to $O(n^{5/2})$ with the algorithm by Hopcroft and Karp [55] and to $O(n^{5/2}/\sqrt{\log n})$ by a method introduced by Alt et al. [2]. Recognizing if the number of chains is at most k is even faster: $O(n^2k^2)$ [39].

As the first stage of the algorithm is exponential, we implemented ChainPartition using the simplest $O(n^3)$ algorithm. As a consequence, the overall worst-case time complexity of Exact algorithm is $O(2^{|T|}|T|^4|S|)$. The consoling fact is that the elements of $\mathcal{U}$ can be independently processed one by one, so it is not necessary to maintain all of them simultaneously, which results in polynomial space complexity.

5.8.3. Greedy Cover Heuristic

The proof of NP-hardness given in Section 5.7 used the Set Covering Problem as a reference problem. This inspired us to adopt an algorithm designed for this well-known problem, a classical greedy heuristic of polynomial complexity that has very good properties. Starting from an empty set $\mathcal{V}$, in each step the heuristic adds to $\mathcal{V}$ such an element from $\mathcal{R}$ that covers the maximal number of not yet covered elements from the universe $\mathcal{U}$. The procedure stops when all elements from universe $\mathcal{U}$ are covered. This heuristic

Algorithm 5.3 GreedyCover heuristic for extracting a minimal coordinate system.

```
1: procedure GreedyCover(S, T, G)
2:     V ← ∅                                          ▷ Working set of tests
3:     N ← {(s1, s2)|s1, s2 ∈ S ∧ ∃t∈T s1 <t s2}      ▷ Set of pairs not yet ordered
4:     while N ≠ ∅ do                  ▷ Are all pairs ordered by tests from V?
5:         u ← argmax_{t∈T\V} |{(s1, s2) ∈ N|s1 <t s2}|
6:         N ← N \ {(s1, s2) ∈ N|s1 <u s2}
7:         V ← V ∪ {u}
8:     end while
9:     return ChainPartition(V, ≤)
10: end procedure
```

was shown [66] to achieve an approximation ratio of

$$\sum_{k=1}^{s} \frac{1}{k} \leq \ln s + 1, \tag{5.8.3}$$

where s is the size of the largest set in $\mathcal{R}$. It was also proved [92, 118, 1] to be the best possible polynomial-time approximation algorithm for this problem.

To solve Dimension Problem, we need to search for a set of tests that preserves the relation for all pairs of candidate solutions, i.e., orders some pairs of candidate solutions. Thus, it is easy to spot similarities between the Set Covering Problem and Dimension Problem. This led us to designing the GreedyCover heuristic (Algorithm 5.3). The algorithm first uses the classical heuristic for Set Covering Problem to construct a set of tests V that order all pairs of candidate solutions from set S. Then it computes the minimal chain partition on $(V, \leq)$.

The correctness of this heuristic results directly from Theorem 32 and the fact that elements in a chain are ordered by $<$ relation. Assuming that minimal chain partition is computed using the simplest $O(n^3)$ algorithm (this can be improved, see discussion in 5.8.2), GreedyCover has a worst case polynomial time complexity of $O(|T|^2|S|^2+|T|^3)$, because the loop in line 4 executes maximally $|T|$ times and the cost of line 5 is $O(|T||S|^2)$.

Although the approximation ratio (if any) of GreedyCover is unknown, the fact that the heuristic is based on an algorithm for which the approximation ratio is low, makes us hypothesize that our algorithm will perform well in practice. This intuition will be verified in the experiments in the following section.

5.9. Experiments and Results

5.9.1. Compare-on-One

The goal of the first experiment is to verify how the dimension computed by EXACT algorithm corresponds to the intrinsic properties of COMPARE-ON-ONE, a variant of the abstract Numbers Game [140], proposed in [34] and widely used as a coevolutionary benchmark [28, 25, 34, 26, 29, 32, 30, 17, 127].

Problem 50. Compare-on-one game

In this test-based problem, strategies are represented as non-negative real-number vectors of length d, which we call here the *a priori dimension* of the game. The outcome of the interaction between candidate solution s and test t depends only on the dimension in which test t has the highest value. Formally, the index of this dimension is $m = \arg\max_{i=1\ldots d} t[i]$, where $t[i]$ denotes the i-th element of vector t. The interaction function is defined as follows:

$$G(s,t) \iff s[m] \geq t[m].$$

The rules of a two-dimensional version of COMPARE-ON-ONE are visualized in Figure 5.9.1. Candidate solution s_1 solves only tests from the shaded area.

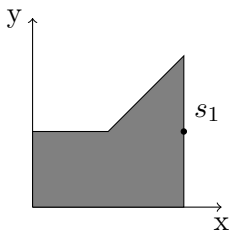


Figure 5.9.1.: A visualization of rules of two-dimensional COMPARE-ON-ONE ($d = 2$). Each strategy (a candidate solution or a test) is represented as a point in a 2D space. According to the game definition, a candidate solution s_1 solves all tests from the gray area, thus it solves t_1, but fails t_2.

Despite its straightforward formulation, COMPARE-ON-ONE is a challenging problem because it has been designed to induce over-specialization: a coevolving system of candidate solutions and tests can easily focus on some (or even a single one) underlying objectives (here: axes of the multi-dimensional space), while ignoring the remaining ones. To make steady progress on this problem, a coevolutionary algorithm has to carefully maintain the tests that support *all* underlying objectives from the very beginning of the run.

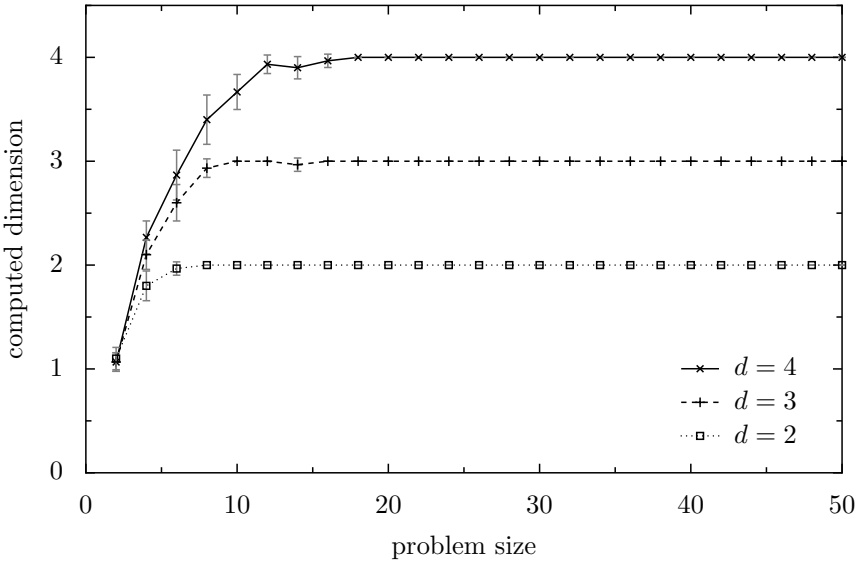


Figure 5.9.2.: The figure shows how the dimension for COMPARE-ON-ONE changes when increasing the number of strategies involved. It may be observed that the computed dimension converges to the *a priori* dimension of the game. Here shown for $d = 2, 3, 4$. Grey vertical whiskers denote 95% confidence intervals.

As COMPARE-ON-ONE is an artificial problem, we can objectively and precisely measure the progress of coevolution (e.g., by using the shaded area in Fig. 5.9.1), which is usually troublesome for many other test-based problems and real games [31].

In order to check how, for a given d, the game dimension changes for growing S and T, we randomly generated $n = |T|$ test strategies and $n = |S|$ candidate solution strategies from a fixed $[0.0, 10.0]$ interval and computed the dimension with EXACT for different values of problem size n. The results of this procedure for $d = 2, 3, 4$ are shown in Fig. 5.9.2. Each data point represents the computed dimension averaged over 30 random samples with 95% confidence intervals. The plot clearly indicates that the computed dimension of the game converges to the *a priori* dimension d of the game with growing problem size n. Also, for this game, its dimension may be reliably estimated already from a small number of interactions.

The above results lead to the question what the minimal coordinate system for this game looks like. It is easy to notice that $t[0, 1]$ is indiscernible from $t[0.5, 1]$ with respect to S, since only the highest value counts. In general, $t[a_1, \ldots, a_m, \ldots a_d]$ is indiscernible from $t[b_1, \ldots, b_m, \ldots, b_d]$, whenever $a_m = b_m$ and $\forall_{i \neq m} a_i \leq a_m \wedge b_i \leq b_m$. Thus, the subset of tests of the form $t[0, \ldots, a_m, \ldots, 0]$, where $a_m > 0$, is sufficient to construct the

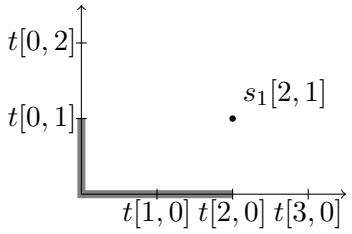


Figure 5.9.3.: Minimal coordinate system for two-dimensional COMPARE-ON-ONE.

minimal coordinate system. The observation that $t[0, \ldots, a_m, \ldots, 0] < t[0, \ldots, b_m, \ldots, 0]$ if and only if $a_m < b_m$ makes it possible to define a minimal correct coordinate system:

$$A_i = \{[0, \ldots, a_i \ldots, 0], a_i \in \mathbb{R}^+\} \tag{5.9.1}$$

for $i = 1 \ldots d$, and

$$p_i(s[s_1, \ldots, s_i, \ldots, s_d]) = t[0, \ldots, s_i, \ldots, 0]. \tag{5.9.2}$$

Thus, for this problem, the number of underlying objectives equals the a priori dimension of the game d.

Figure 5.9.3 shows the minimal coordinate system for $d = 2$, with the tests solved by an exemplary candidate solution s_1 marked by a gray line.

In most trials, the minimal coordinate system found by EXACT was coherent with the minimal coordinate system described above, so the algorithm correctly identified the d underlying objectives of the game. Only when the number of candidate solutions and tests n was small in proportion to d, EXACT produced axes that did not correspond to such objectives.

5.9.2. Compare-on-All

In the second experiment, we examine another abstract game, the COMPARE-ON-ALL [34] (a.k.a. TRANSITIVE [17]).

Problem 51. Compare-on-all

In COMPARE-ON-ALL, strategies are represented like in COMPARE-ON-ONE, but the interaction function is defined as weak dominance relation:

$$G(s, t) \iff \forall_i s[i] \geq t[i].$$

The rules of COMPARE-ON-ALL are visualized in Figure 5.9.4.

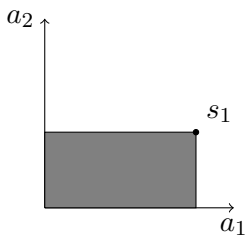


Figure 5.9.4.: A visualization of rules of two-dimensional COMPARE-ON-ALL ($d = 2$). Both tests and candidate solutions are points in the d-dimensional space. s_1 solves all tests from the gray area.

The results (computed with EXACT) for this problem for $d = 2, 3, 4$ are shown in Fig. 5.9.5. Although the computed dimension grows much slower than the problem size, this time the computed dimension clearly fails at approximating the *a priori* game dimension d. Even worse, it does not seem to saturate with growing n. To some extent, this result is similar to the results obtained by Bucci et al. in [17], who, using SIMPLE-GREEDY heuristic and a variant of Population Pareto Hill Climber (P-PHC) algorithm for generating candidate solutions and tests, found out that the computed dimension was overestimating the *a priori* dimension, especially for large values of the latter one.

Our results demonstrate that even using an exact algorithm, it is hard to arrive at the true dimension of this test-based problem. Let us also notice that in [17] the scale of overestimation was much lower than in our experiment (e.g., for $d = 10$, the game dimension was estimated to be only ca. 17). The two experiments are not easily comparable because of the way in which the candidate solutions and tests were generated in [17] (P-PHC); however, taking into the consideration the fact that the true dimension of COMPARE-ON-ALL equals its *a priori dimension* (see below), better estimates found in [17] indicate that properties of generators are of crucial importance when designing practical coevolutionary algorithms.

The minimal coordinate system for COMPARE-ON-ALL looks as the one for COMPARE-ON-ONE. Also here we need only the tests of the form $t[0, \ldots, a_m, \ldots, 0]$, where $a_m > 0$: any test having more than one non-zero element can be discarded because its solutions failed set can be constructed using the solutions failed sets of tests located on the axes (cf. the proof of Proposition 49). Therefore, since also here $t[0, \ldots, a_m, \ldots, 0] < t[0, \ldots, b_m, \ldots, 0]$ if and only if $a_m < b_m$, and this relation cannot be modeled in a lower number of axes, the minimal coordinate system is defined by (5.9.1) and (5.9.2) as for COMPARE-ON-ONE, and has d dimensions.

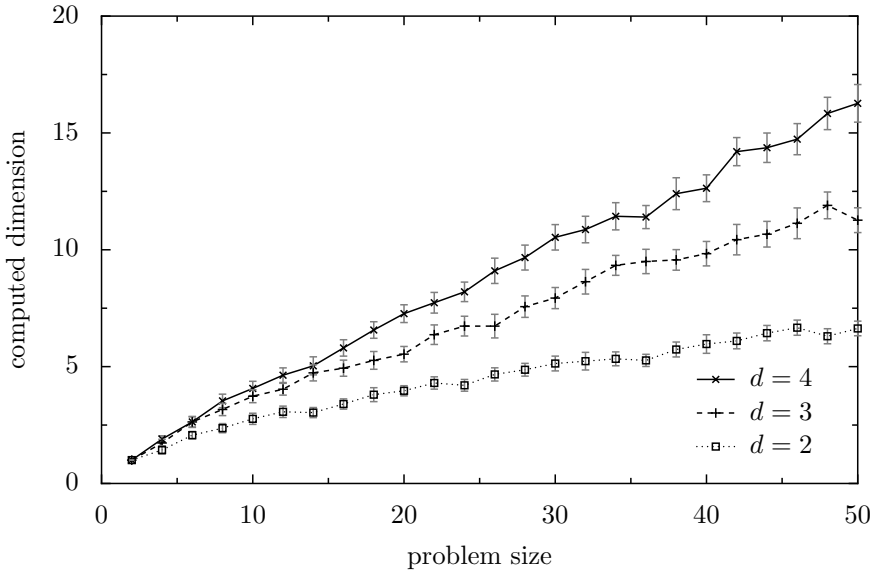


Figure 5.9.5.: The figure shows how the dimension for COMPARE-ON-ALL changes when increasing the number of strategies involved. Notice that for $d = 2, 3, 4$, the computed dimension does not converges to the *a priori* dimension of the game. Grey vertical whiskers denote 95% confidence intervals.

5.9.3. Dimension of Random Test-Based Problem

The goal of the next experiment was twofold: first, to compare Bucci's SIMPLEGREEDY [17] with our EXACT and GREEDYCOVER algorithms; second, to observe how the computed dimension changes with the problem size. To this aim, we considered random test-based problems of different sizes $n = 2 \dots 200$. A random test-based problem of size n is given by a random payoff matrix $n \times n$ (n tests and n candidate solutions), with each interaction outcome drawn independently at random with equal probability. The dimension computed by the algorithms are shown in Fig. 5.9.6. Each data point represents the computed dimension averaged over 30 random matrices, with 95% confidence intervals.

Figure 5.9.6 gives rise to several interesting observations. EXACT performs clearly much better than SIMPLEGREEDY, which hardly does any compression. The gap between the algorithms grows rapidly with n, so that SIMPLEGREEDY overestimates twice the true dimension computed by EXACT already for $n = 25$, and ten times for $n = 200$ (the latter case did not fit in the figure). The compression provided by EXACT is impressive, given that incomparability of almost all pairs of test becomes almost certain for large random matrices, since the probability that a test weakly dominates another

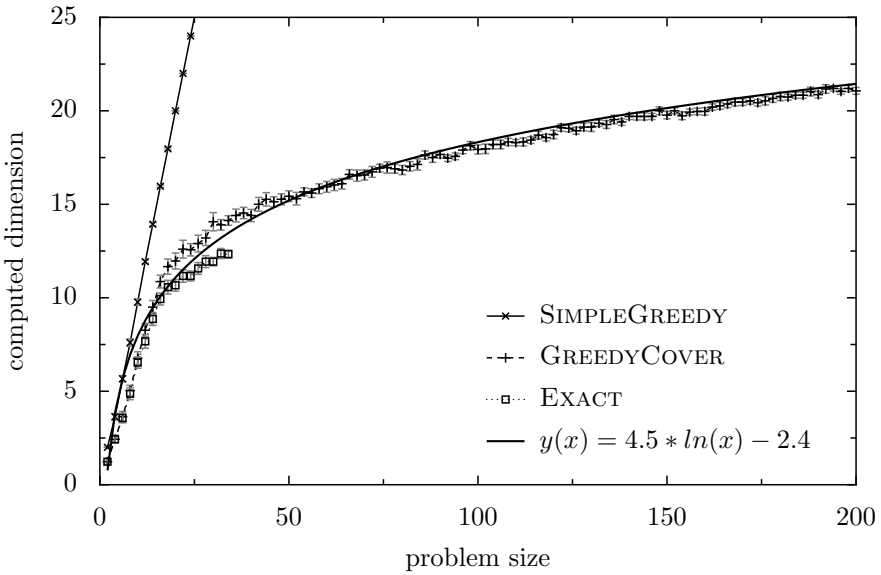


Figure 5.9.6.: Comparison of the dimension computed by three algorithms: SimpleGreedy, Exact, GreedyCover on a random problem in the function of the problem size. Clearly, SimpleGreedy is inferior to both Exact and GreedyCover. On the other hand, GreedyCover, which is only a heuristic, performs nearly as well as Exact. Grey vertical whiskers denote 95% confidence intervals.

test is $\left(\frac{3}{4}\right)^{n}$[2]. Note also that the logarithm-like shape of EXACT curve suggests that the compression could be even higher for larger n. On the other hand, EXACT is obviously much slower than SIMPLEGREEDY — that is why we could not produce results for $n > 34$.

However, in contrast to SIMPLEGREEDY, the GREEDYCOVER heuristic seems to approximate the true problem dimension very well. For small instances ($n \leq 14$), it is not significantly worse than EXACT. For larger instances, for which we have exact results, it overestimates the test-based problem dimension only slightly.

Moreover, the dimension computed by GREEDYCOVER seems to follow a logarithmic curve, similarly to the actual problem dimension (as computed using EXACT). Since the result of GREEDYCOVER is an upper bound of problem dimension, we may hypothesize that, on average, the dimension of a test-based problem is bounded from above by a logarithm of problem size. This result may indicate that the dimension of at least some test-based problems is possible to handle and that the compression obtained by extracting underlying objectives may be exponential with respect to the total number of tests.

5.9.4. Estimating Problem Dimension

In order to verify how the above result generalizes over more complex test-based problems, we performed experiments similar to the one in Section 5.9.3 on two other problems: tic-tac-toe and density classification task.

5.9.4.1. Problems

Tic-Tac-Toe has already been introduced in Chapter 3, but here we approach the game in a different way. In contrast to the setup described in Chapter 3, we separate the strategies for player X from strategies for player O. The former become the candidate solutions and the latter should be identified with tests. As a result, the interaction function involves one strategy of player X and one strategy of player O, and may end with a win or defeat of player X or with a draw. Moreover, in order to meet the assumptions made in this chapter about the interaction function's codomain ($G : S \times T \to \{0, 1\}$), we treat both a win and a draw of player X as passing a test (player's O strategy) and defeat as failing it.

In contrast to Genetic Programming encoding used in Chapter 3, which may bias the search of strategy space, here we encode strategies directly [8] in order to be able to

[2]The probability of the event $t_1 \leq_{s_1} t_2$ is $\frac{3}{4}$, because out of four possible cases, it does not occur only when $G(s, t_1) = 1$ and $G(s, t_2) = 0$.

consider all possible strategies for this game. The direct encoding explicitly defines the strategy's move in each possible board state. Using little computational power, we may check that tic-tac-toe has 765 different board states (not counting boards created by rotation or reflection). 138 of them are final (3 boards finishing with a draw, 91 boards with player X winning and 44 boards with player O winning), while in the remaining 627 boards moves are still possible. Among them, in 338 cases player X is to play and in 289 cases it is player's O turn. Thus, we encode player X with 338 genes and player O with 289 genes. A gene corresponds to one board state and it encodes the move to play in this state.

Tic-tac-toe is a simple problem easily solved by a minimax algorithm [124]. However when we abstract from the fact that an interaction between a candidate solution and a test involves only a few moves on a simple 3×3 board and we are just left with the results of interactions between strategies, the number of possible strategies makes tic-tac-toe a non-trivial task [8]. Assuming that two strategies are different when they play differently in at least one board state, the total number of strategies for player X (candidate solutions) is approximately 3.47×10^{162} and the number of strategies for player O (tests) is approximately 2.82×10^{142}.

Density Classification Task has been described in Section 2.2.5 on page 27. Here, we use an instance of density classification task in which the rule's radius $r = 3$, the size of cellular automata $n = 59$ and the number of time steps $t = 100$. Thus, in our problem, there are $2^{3^{2r+1}} = 2^{128}$ rules (candidate solutions) and $2^n = 2^{59}$ initial configurations (tests).

5.9.4.2. Results

The number of strategies in both tic-tac-toe and density classification task is too high to analyze the whole payoff matrix. That is why, we randomly and uniformly sampled the strategies from the whole strategy space. Given the constraints on computer resources, we were able to consider payoff matrices from 5×5 to 1885×1885 with step 5 for tic-tac-toe and 10×10 to 4500×4500 with step 10 for density classification task. For each such payoff matrix the number of underlying objectives was estimated by the GreedyCover algorithm.

Figure 5.9.7 shows the results of this procedure for tic-tac-toe. Two standard trend curves fit our data: a power function with equation $y = 4.7x^{0.357}$ with the coefficient of determination $R^2 = 0.86$; and a logarithmic function $y = 12.62\ln(x) - 31.6$ with $R^2 = 0.93$. The difference between the R^2 values is small, so there is not enough

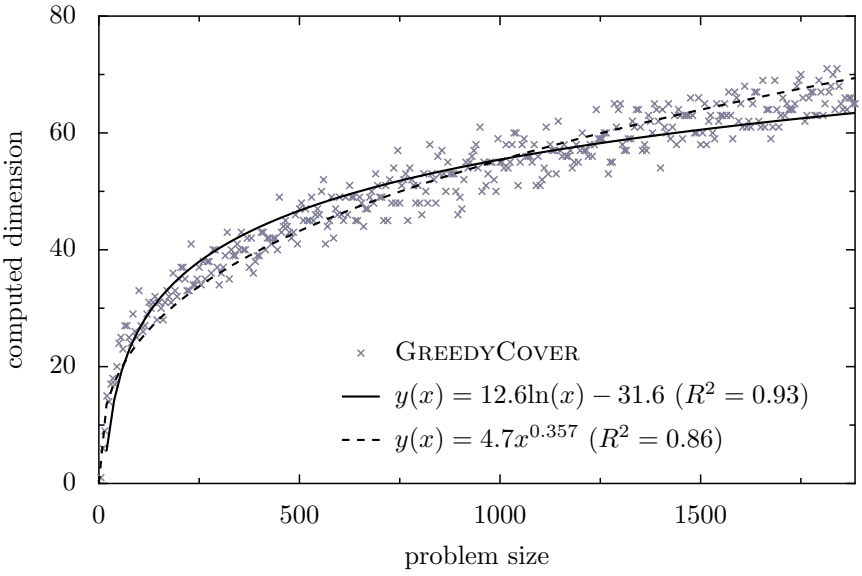


Figure 5.9.7.: Dimension trend for of tic-tac-toe.

evidence to confidently claim that the logarithmic curve describes the dimension trend better. Having said that, we have to recall that GreedyCover is a heuristic and as such it overestimates the dimension (cf. Fig. 5.9.6 on page 96); moreover, the difference between the problem dimension and its estimation by GreedyCover is likely to be higher for bigger payoff matrices. Although this observation does not prove anything, it is a plausible argument in favor of the logarithmic curve.

To model the trend more precisely, one should estimate dimensions for greater payoff matrices, but this, despite polynomial complexity, is computationally demanding. It has taken a week on a modern PC to run the algorithm for payoff matrices up to 1885×1885, and the greater the matrix, the greater the computational cost; GreedyCover is also memory demanding, thus the amount of data required to model the trend with greater confidence is not expected to substantially increase in the predictable future, unless a better algorithm than GreedyCover is designed.

The results for density classification task are presented in Fig 5.9.8. This time, the data are best modeled by a linear function $y(x) = 0.0074x - 0.57$ with $R^2 = 0.87$.

5.9.4.3. Discussion

When trying to estimate the dependency between the problem size and its dimension, we got different results for different problems. For the random problem considered in Section 5.9.3, the trend is clearly a logarithmic function. In Section 5.9.4, we found

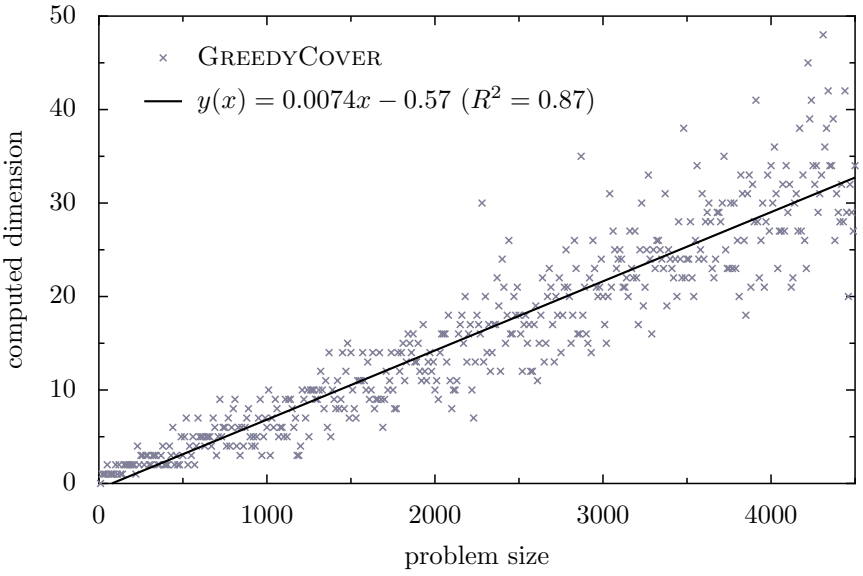


Figure 5.9.8.: Dimension trend for an instance of density classification task ($r = 3$, $n = 129$, $t = 100$).

that for tic-tac-toe, it may be a logarithmic or power function, while for our instance of density classification task the trend is linear. We may, thus, conclude that the relation between the dimension of a problem and its size is highly problem dependent.

Having the relation between the size of the problem sample and its dimension, we may try to extrapolate the modeled trend to estimate *the upper bound of the dimension* of a problem (i.e., technically, to substitute for n the actual problem size). For our instance of density classification task, the dimension is approximately $0.0074 \times 2^{129} - 0.57 \approxeq 5.04 \times 10^{36}$. Since for tic-tac-toe we are not able to unambiguously determine the trend function, for this problem we may make claims only conditionally. If we assume that the power function correctly determines the dimension trend, then extrapolating it to 3.47×10^{162} strategies, the dimension is approximately $4.7 \times \left(3.47 \times 10^{162}\right)^{0.357} \approxeq 5.0 \times 10^{58}$, but when we assume that the logarithmic function is the one which models the data better, the dimension equals only $12.6\ln(3.47 \times 10^{162})) - 31.6 \approxeq 2038$. If the latter is true, then tic-tac-toe could be perceived as an easy problem when compared with density classification task.

The table below summarizes the computed estimates. Although they should be treated with caution, since the numbers are subject to substantial uncertainty resulting from extrapolation, we may conclude that the compression rate obtained is significant for both problems considered in this section.

	Tic-tac-toe	Density classification task ($r = 3$, $n = 129$, $t = 100$)
Number of tests (original objectives)	2.82×10^{142}	6.81×10^{38}
Number of candidate solutions	3.47×10^{162}	3.40×10^{38}
Estimated upper bound of the problem dimension	5.0×10^{58} or 2038	5.04×10^{36}

5.10. Relation to Complete Evaluation Set

It is interesting to notice the relation between our findings in this chapter and the idea of *ideal evaluation* introduced by de Jong and Pollack [34], which was based on a concept of *complete evaluation set*. The complete evaluation set is defined as a subset of T that preserves all relations between candidate solutions from S. Notice that the set of all tests $\bigcup \mathcal{C}$ of a correct coordinate system $\mathcal{C}$ is a complete evaluation set, since the condition (33) holds. By transforming the problem of determining the complete evaluation set to the Set Covering Problem, it is easy to show that also the problem of determining the minimal complete evaluation set is NP-hard.

The complete evaluation set is used in the coevolutionary algorithm Delphi introduced in [34]. Although the authors state that Delphi does not need to compute the *minimal* complete evaluation set, it is reasonable to hypothesize that approximating it could lead to better performance of Delphi by decreasing the number of tests it has to maintain. Employing a variant of GreedyCover that computes just the complete evaluation set may thus be beneficial also for Delphi, opening the door to a practical application of results obtained in this chapter.

Let us also point reader's attention to the analogy between the above concepts and those of the rough set theory [113]. The starting point in rough set theory is an information system, i.e., a table containing objects (corresponding to rows) described by attributes (columns), where for each attribute an indiscernibility relation is also given. A reduct is any subset of attributes that induces the same equivalence classes in the set of objects as the set of all attributes. Thus, it plays a similar role there as the complete evaluation set here, i.e., it preserves the structure present in the original data (information system or test-based problem). It should not come as a surprise that finding

a minimal reduct is NP-complete. Obviously, the important difference is that standard rough set theory does not involve dominance relation.

5.11. Discussion and Conclusions

Test-based problems form an important and surprisingly common class of optimization problems which was not recognized as a separate branch of research until recently, and can be conveniently modeled and studied within the framework of coevolution. They are difficult by nature, and thus require a proper formal analysis that was the main aim of this chapter. We concentrated on the notion of coordinate system, a concept introduced in [17] that allows extracting the internal structure of test-based problems by means of underlying objectives, which can be potentially exploited to maintain the progress of search in coevolutionary algorithms.

A significant part of this chapter was devoted to revealing the properties of coordinate systems. Apart from determining the lower and upper bounds for problem dimension, we proved its equality to the width of the partially ordered set of tests. Moreover, we formally identified the tests that are redundant and can be safely discarded when constructing a coordinate system

These findings allowed us to answer the question about the complexity of the problem of extracting the minimal coordinate system. Despite the fact that the problem turned out to be NP-hard, we demonstrated that the problem can be solved at low computational cost by means of an appropriate heuristic. Our GreedyCover algorithm is clearly superior to SimpleGreedy, the best algorithm proposed so far in terms of approximating the true problem dimension, and similar to it in terms of computational complexity. Additionally, we carefully designed an exact algorithm which, though exponential, may be used for problems of moderate size.

In the experimental part, we have shown that application of these algorithms leads to significant compression of the objective space of test-based problems. We demonstrated that, on average, the number of underlying objectives for an abstract random test-based problem seems to be limited from above by a logarithm of the number of tests. This result has been also verified on tic-tac-toe and an instance of density classification task. Cautiously speaking, we could not rule out the possibility that the dimension of tic-tac-toe is also bounded by a logarithm of the number of tests. For the considered instance of density classification task, the number of underlying objectives grows linearly with the number of tests; however, the small slope (0.0074) of the trend function makes the number of underlying objectives still significantly lower than the number of tests. All

in all, our results indicate that the dimension of some test-based problems, including the well-known ones like tic-tac-toe or density classification task, is typically much lower than that resulting from the original idea of Pareto-coevolution that treats every test as a separate objective.

In multi-objective optimization, in general, the more objectives, the harder the problem [12]. If we assume that also for test-based problems, which may be interpreted in a multi-objective manner, problem dimension is a yardstick of problem difficulty[3], we can conclude that some test-based problems, though still hard, are likely to be less difficult than previously thought. Hence, random sampling used extensively in Section 5.9 as a method estimating the dimension trend may be a tool of practical interest. In this light, tic-tac-toe turns out to be much easier than our instance of density classification task.

In case of some problems (here: COMPARE-ON-ONE game), we have shown that the axes of extracted coordinate systems correctly identify the true underlying skills of the game. On another problem, COMPARE-ON-ALL, we have demonstrated that sometimes, even with an exact algorithm, the true dimension of a game may not be found if just a random sample of tests and candidate solutions are provided. This indicates the importance of design of effective generators for tests and candidate solutions in coevolutionary algorithms based on the idea of coordinate system.

This chapter focused on *modeling* the underlying structure of the problem; how to effectively *exploit* it will be the theme of Chapter 6. Polynomial-time complexity and good results provided by GREEDYCOVER make it particularly appealing for coevolutionary algorithms, enabling to update the coordinate system online (i.e., during the run) in a framework with a coevolutionary archive.

[3] A plausible argument for this assumption is the, now widely accepted, distinction of such problems into multi-objective (number of objectives < 4) and many-objective (number of objectives ≥ 4) [57]

6. Coordinate System Archive

In this chapter, we propose a coevolutionary archive, called Coordinate System Archive (COSA), which is based on the idea of extracting the underlying structure of test-based problems using coordinate systems introduced in Chapter 5. COSA is designed to work for asymmetric test-based problems. In this respect, on the one hand, it is more widely applicable than Fitnessless Coevolution presented in Chapter 3, which was designed for symmetrical test-based problems only. On the other hand, Fitness Coevolution has been applied to a complex problem, but COSA is not mature enough to have practical applicability yet. Instead, it should be treated as a proof-of-concept that the coordinate system defined in Chapter 5 may be employed in coevolutionary algorithms for solving test-based problems.

6.1. Introduction

The idea of extracting and using the underlying structure of test-based problems to maintain the progress in coevolution was first applied in Dimension Extraction Coevolutionary Algorithm (DECA) [32]. Here we propose another method based on this concept, called Coordinate System Archive (COSA). Our method differs substantially from the earlier attempt, since DECA uses a distinct definition of coordinate system (see Chapter 5) and exploits it in a different way.

6.2. Coordinate System Archive (COSA)

COSA is a coevolutionary archive that may be used as a component in the generator-archive scheme for solving test-based problems by coevolutionary methods (cf. Section 2.3.4 and Alg. 2.1 on page 33), and as such it can work with any generator. The idea of our archive algorithm is based on the concept of underlying problem structure and coordinate system described in Chapter 5. The archive extracts a coordinate system of the test-based problem consisting of currently available candidate solutions and tests, and uses it to retain the *base axis set*, containing one test from each axis of the extracted

Algorithm 6.1 Coordinate System Archive (COSA)

procedure SUBMIT(S_{new}, T_{new})
  $T_{tmp} \leftarrow T_{arch} \cup T_{new}$
  $S_{tmp} \leftarrow S_{arch} \cup S_{new}$
  $T_{tmp} \leftarrow$ GETUNIQUE(T_{tmp}, S_{tmp})
  $S_{tmp} \leftarrow$ GETUNIQUE(S_{tmp}, T_{tmp})
  $S_{Pareto} \leftarrow \{s \in S_{tmp} | \forall_{s' \in S_{tmp}} s' \leq_{T_{tmp}} s\}$
  $T_{base} \leftarrow$ FINDBASEAXISSET(T_{tmp}, S_{Pareto})
  $S_{req} \leftarrow$ PAIRSETCOVER($S_{Pareto}, S_{tmp}, T_{base}$)
  $T_{req} \leftarrow$ PAIRSETCOVER($T_{base}, T_{tmp}, S_{Pareto}$)
  $S_{arch} \leftarrow S_{req}$
  $T_{arch} \leftarrow T_{req}$
end procedure

procedure GETUNIQUE(A, B)
  $U \leftarrow \emptyset$
  for $a \in A$ **do**
    $I \leftarrow \{e \in A | \forall_{b \in B} G(a, b) = G(e, b)\}$
    $U \leftarrow U \cup$ oldest individual from I
    $A \leftarrow A \setminus I$
  end for
  return U
end procedure

procedure PAIRSETCOVER(A_{must}, A, B)
  $A \leftarrow A \setminus A_{must}$
  $\mathcal{N} \leftarrow \{(b_1, b_2) | b_1, b_2 \in B$ ▷ Pairs to be ordered
    $\wedge \exists_{a \in A} b_1 <_a b_2) \wedge \nexists_{a \in A_{must}} b_1 <_a b_2)\}$
  $V \leftarrow A_{must}$
  while $\mathcal{N} \neq \emptyset$ **do** ▷ Are all pairs ordered?
    $u \leftarrow \text{argmax}_{a \in A \setminus V} |\{(b_1, b_2) \in \mathcal{N} | b_1 <_a b_2\}|$
    $\mathcal{N} \leftarrow \mathcal{N} \setminus \{(b_1, b_2) \in \mathcal{N} | b_1 <_u b_2\}$
    $V \leftarrow V \cup \{u\}$
  end while
  return V
end procedure

Algorithm 6.2 A procedure that finds the tests that should be kept in the archive.

```
1: procedure FindBaseAxisSet(T_tmp, S_Pareto)
2:     C ← ChainPartition(T_tmp, ≤)
3:     n_dims = |C|
4:     S_list ← S_Pareto sorted descendingly by min(GetPos(s, C))
5:     for s ∈ S_list do
6:         T' ← {t ∈ T_tmp | G(s, t)}
7:         (A, found) ← the greatest antichain in poset (T', ≤)
8:         if found then
9:             return A
10:        end if
11:    end for
12:    return T_tmp
13: end procedure
14:
15: procedure ChainPartition(X, P)
16:    return minimal chain partition of poset (X, P)
17: end procedure
18:
19: procedure GetPos(s, C)
20:    n_dims ← |C|
21:    P ← array[1 ... n_dims]
22:    for i = 1 ... n_dims do
23:        P[i] ← |{c ∈ C[i] | G(s, c)}|
24:    end for
25:    return P
26: end procedure
```

coordinate system. The objective of the base axis set is to prevent the coevolution from cycling due to inherent interactivity in a multi-objective search space of test-based problems. Additionally, COSA maintains the *Pareto set,* consisting of non-dominated candidate solutions, which represents best candidate solutions found so far. COSA does not maintain unnecessary candidate solutions, thus if the problem is low-dimensional, the base axis set remains small. As a result, COSA can better utilize given processor time than algorithms that maintain many useless tests.

COSA maintains two separate archives, one for candidate solutions (S_{arch}) and one for tests (T_{arch}). Each time new candidate solutions and tests (S_{new}, T_{new}) are submitted to it (line 10 of Alg. 2.1), COSA merges them with the archives into temporary sets of individuals S_{tmp} and T_{tmp} (see Alg. 6.1, lines 2-3). Ultimately, only some of those individuals will be retained in the archives.

The algorithm first gets rid of any duplicates in both archives (lines 4-5): from every equivalence class (group of individuals that are indiscernible in terms of the payoff matrix) only the oldest one is retained. Preferring the older individuals prevents replacing the objectively better individuals by the worse ones, which otherwise would be likely due to the predominantly destructive character of mutation and crossover operators.

Then, in lines 6-7, COSA decides which individuals must be retained. For candidate solutions, it is the Pareto set S_{Pareto}, consisting of candidate solutions that are Pareto non-dominated with respect to T_{tmp}. For tests, it is the *base axis set* T_{base}*,* consisting of one test from each axis determined by the FindBaseAxisSet procedure that will be described later. S_{Pareto} and T_{base} are not sufficient to provide each other *stability*, i.e., to prevent changing the mutual relationship between individuals and, in consequence, to prevent being removed during subsequent submissions. Therefore, COSA selects some additional individuals and forms the supersets S_{req} and T_{req} of, respectively, S_{Pareto} and T_{base}, which become the new archives at the end of the submission phase (lines 10 and 11). S_{req} is a set of candidate solutions that preserves all relations between tests in T_{base}. Formally, if for a pair $t_1, t_2 \in T_{base}$ there exists $s \in S$ such that $t_1 \leq_s t_2$, then S_{req} must contain s' such that $t_1 \leq_{s'} t_2$. Analogically, T_{req} is a set of tests that preserves all relations between candidate solutions in S_{Pareto}.

6.2.1. Stabilizing the archives by PairSetCover

Both S_{req} and T_{req} are computed by PairSetCover (Alg. 6.1), which is a greedy heuristic working according to the same principle as GreedyCover described in Section 5.8.3. Similarly to GreedyCover, it may not produce the *minimal* set of required elements. PairSetCover accepts generic arguments A and B. A_{must} is the set of

individuals that have to be retained. $\mathcal{N}$ is the set of pairs that have to be ordered. Notice that pairs of elements of B are already ordered by individuals from A_{must} and are not included in $\mathcal{N}$ (line 27). Starting with the set V containing only individuals from A_{must}, PairSetCover in each iteration (lines 30-32) extends it by such an element from A that orders the maximal number of not yet ordered pairs from $\mathcal{N}$. The procedure stops when all pairs are ordered ($\mathcal{N}$ is empty).

6.2.2. Finding the base axis set by FindBaseAxisSet

The most important part of the archive algorithm is the FindBaseAxisSet procedure (Alg. 6.2), which determines the base axis set. FindBaseAxisSet builds a coordinate system using ChainPartition, which finds a minimal chain partition $\mathcal{C}$ of poset $(T, \leq)$ (line 2). Similarly to the implementation used in Exact algorithm in the previous chapter (Alg. 5.2), for ChainPartition we use the standard $O(|X|^3)$ method, which computes the max-flow on a bipartite network with unit capacities. Each chain of $\mathcal{C}$, being linearly ordered by relation $<$, corresponds to one axis of the coordinate system, thus $\mathcal{C}$ is a coordinate system (cf. Def. 22 on page 69). As we have shown in Corollary 39 on page 77, the coordinate system built in this way is a correct one (cf. Def. 24 on page 69).

The size of the extracted coordinate system n_{dims} (line 3) is a temporal estimation of the true dimension of our problem. It may, but it does not have to be, the true dimension of the test-based problem we are trying to solve, because (i) the extracted coordinate system is not necessarily minimal, and (ii) we operate on a test-based problem that involves samples of the sets of all possible candidate solutions and all possible tests.

Having the coordinate system $\mathcal{C}$, we can easily determine the coordinates of each candidate solution s in $\mathcal{C}$ by computing the number of tests that are solved by s on each axis (procedure GetPos, lines 19-26). The position is represented by a vector of n_{dims} cardinal numbers (cf. Def. 23).

In lines 4-11, FindBaseAxisSet tries to find an antichain in $(T_{tmp}, \leq)$ with three properties: (i) its size is n_{dims}, (ii) it is the *greatest antichain*, and (iii) there exists a candidate solution that solves all its elements. Considering the first property, according to the Dilworth theorem (Theorem 7), such antichain always exists, since $\text{width}\,(T_{tmp}, \leq) = n_{dims}$.

Let us explain the second property. Let $\mathcal{X}$ be the set of all maximum antichains of $(T_{tmp}, \leq)$ and $\mathcal{P}$ be a relation on $\mathcal{X}$ such that $X_1 \mathcal{P} X_2$ with $X_1, X_2 \in \mathcal{X}$ iff there exists a bijection $f : X_1 \to X_2$ such that $x_1 \leq f(x_1)$ for all $x_1 \in X_1$. Notice that the poset $(\mathcal{X}, \mathcal{P})$ has the maximum element, called here the *greatest antichain* of T_{tmp}, which is

computed by a polynomial-time algorithm in line 7.

The third property cannot be always fulfilled, since the maximum antichain of size n_{dims}, guaranteed to exist in $(T_{tmp}, \leq)$, does not necessarily exist in $(T', \leq)$, where $T' \subseteq T_{tmp}$ is a set of tests solved by a certain candidate solution s.

FindBaseAxisSet iterates over all candidate solutions from S_{Pareto} and returns the first encountered antichain with the three properties described above (line 9); if such antichain does not exist, it returns all tests from T_{tmp}. The candidate solutions from S_{Pareto} are considered in the order of decreasing minimal coordinate, which identifies the weakest element of a candidate solution. Thanks to that, the algorithm prefers the search direction that treats all the underlying objectives equally, which is intended to protect COSA from over-specialization.

6.3. Experiments

In order to validate COSA, we conducted an experiment in which we compared it with two state-of-the-art coevolutionary archives IPCA and LAPCA, which use the concept of Pareto domination but do not involve coordinate systems. As benchmark test-based problem we used two number games: compare-on-one and compare-on-all.

6.3.1. Iterated Pareto-Coevolutionary Archive (IPCA)

IPCA is a coevolutionary archive proposed in [26] and further investigated in [30]. It guarantees monotonic progress for the solution concept of Pareto-Optimal Set. This guarantee, however, comes at a cost: its test archive may grow infinitely.

IPCA maintains a set of tests and a set of candidate solutions. A newly generated candidate solution is accepted by the archive only if it is non-dominated with respect to the tests maintained in the archive. When a candidate solution is accepted, a newly generated test that is required to keep it non-dominated is also accepted to the archive. The candidate solutions in the archive that become dominated by newly accepted candidate solutions are removed.

6.3.2. Layered Pareto-Coevolutionary Archive (LAPCA)

LAPCA [28] maintains a set of *Pareto layers*. For a given set of candidate solutions and tests ($S' = S_{arch} \cup S_{new}$, $T' = T_{arch} \cup T_{new}$), the first Pareto layer consists of all non-dominated candidate solutions. Each subsequent layer is obtained in a similar way, after removing the candidate solutions from all previous layers. The solution archive

consists of candidate solutions of first l layers, where l is a parameter of the method. In theory, l makes it easy to trade-off the archive size (thus the computational power required) and reliability of the archive. However, in general, it is unknown which value of l is right for a particular problem.

LAPCA also maintains an archive T_{arch} of tests that separate the candidate solutions stored by S_{arch}. For any two candidate solutions $s_i, s_j \in S_{arch}$ in layers i and j respectively, where $|i - j| \leq 1$, if there exists a test $t \in T'$ that orders s_i before s_j, then such a test must also be retained in T_{arch}. (Notice that if $i = j$, s_i and s_j are in the same layer). Tests are processed in any order, and the first one that orders a not-yet-ordered pair of candidate solutions is selected to T_{arch}.

6.3.3. Objective Progress Measures

COMPARE-ON-ONE and COMPARE-ON-ALL problems are defined in Sections 5.9.1 on page 91 and 5.9.2 on page 93, respectively. Originally, those problems were defined as *open-ended*, which means that the values of genes (variables) of strategies are not limited from above and could increase infinitely. This raises problems with defining a reasonable solution concept for such problems, because, intuitively, the best solution for such problems does not exist. Thus, here we limit the strategy space by a non-negative value m in such a way that for any strategy x (candidate solution or test) $x[i] \leq m$ for $i = 1 \ldots d$.

In order to objectively monitor the progress of algorithms on these problems, we employ two performance measures: *lowest dimension* and *expected utility*.

A standard performance measure for these problems used by other authors (e.g., in [34, 29, 144]) is the lowest dimension, which was designed to detect over-specialization (cf. Section 5.9.1). For a given candidate solution s, it is defined as $\min_i s[i]$, thus it determines the 'weakest point' of the candidate solution.

Expected utility, on the other hand, takes into the account all 'strong points' of a candidate solution s and it is defined as the probability that s solves a test. However, as we do not want to assign any arbitrary value to m, we equal the expected utility with the hypervolume of the polyhedron that contains all the tests solved by s. For COMPARE-ON-ONE the hypervolume of such a polyhedron is given by

$$U(s) = \frac{1}{d} \sum_{i=1 \ldots d} s[i]^d,$$

where d is the a priori dimension of the problem. The hypervolume is visualized in Fig. 5.9.1 on page 91 (for $d = 2$) and in Fig. 6.3.1 (for $d = 3$).

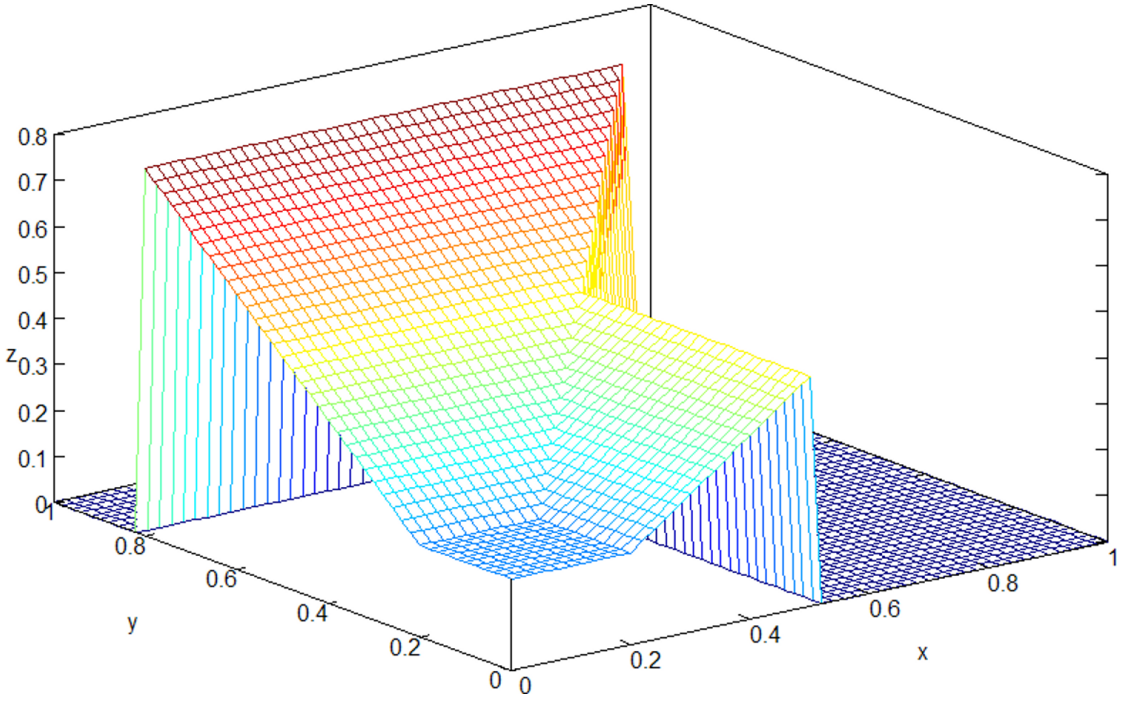


Figure 6.3.1.: Visualization of COMPARE-ON-ONE for $d = 3$. A candidate solution $s = [0.5, 0.8, 0.2]$ solves all tests bounded by the polyhedron.

For COMPARE-ON-ALL the polyhedron is a hyperrectangle (for $d = 2$ cf. Fig. 5.9.4 on page 94), thus for a given a priori dimension d, its hypervolume is given by

$$U(s) = \prod_{i=1...d} s[i].$$

The performance measures we use here could be interpreted as objective functions, based on which the preference relation for test-based problems could be easily constructed inducing a total order on potential solutions (here: candidate solutions). Notice that both preference relations correspond to Maximization of Expected Utility solution concept (cf. Section 2.2.4), which, in this case, consists of one element: the strategy $[m, m, \ldots, m]$.

6.3.4. Setup

In the experiment, we used the common generator-archive scheme for coevolution presented in Alg. 2.1 on page 33. The coevolutionary algorithm maintains n candidate

solutions and n tests (here, $n = 20$) and works as follows. At the beginning, both populations are initialized, with each gene drawn at random from the $[0, 1]$ range (line 2 of Alg. 2.1). Archives are initially empty (lines 3-4). Afterward, the algorithm iterates over consecutive generations in a way that resembles an ordinary evolutionary algorithm (cf. [51]). Each generation involves breeding of new candidate solutions and tests using the generator (lines 6 and 7), evaluation (line 11), and selection (line 12). The following paragraphs detail these stages.

The exploration of the search space is driven by a generator which is responsible for providing genetic variation. Technically, the generator of candidate solutions (GenerateNewSolutions in Alg. 2.1) and the generator of tests (GenerateNewTests) work in the same way, producing n individuals (offspring) in the following way. First, a generator decides randomly (with equal probability) whether the parent of the generated individual should come from the population or from the archive. Second, it mutates the parent and returns it as a new individual. The difference between generating candidate solutions and generating tests lies in handling the archive. The generator of tests uses all tests from T_{arch} as potential parents, whereas the generator of candidate solutions uses only the Pareto non-dominated candidate solutions from S_{arch} as potential parents (this increases the pressure towards better candidate solutions).

Mutation randomly perturbs two genes of an individual (candidate solution or test). Following [30], the mutation is uniform in interval $[-0.2, 0.1]$, thus it is negatively biased: it is more likely to decrease a gene than to increase it. Such asymmetry causes the problem to be harder and bear more resemblance to real problems, where variation is more likely to cause regress than progress [30]. Negative values of genes are clamped to 0.

For a similar reason our generators refrain from crossing over the parent strategies: for the considered number games and the way the strategies are encoded, crossover could easily produce very good candidate solutions and could alleviate to some extent the intentionally built-in tendency to focusing, rendering the game too easy[1].

The rationale behind mutating exactly *two* of vector elements is that such variation simulates epistatic interactions between the elements of the strategy or, in other words, a non-trivial genotype-phenotype mapping [28] — the offspring (generated strategy) differs from the parent on more than one dimension (skill). Mutating only one element would imply one-to-one correspondence between genes (strategy elements) and game skills, which would be simplistic and unrealistic from the real-world perspective. Thus,

[1] Consider, for example, two heavily over-specialized candidate solutions: [0,10] and [10,0] and a possible product of crossing-over them: [10,10]

modifying two elements is the minimal non-trivial perturbation. This and other aforementioned design choices make COMPARE-ON-ONE and COMPARE-ON-ALL, despite their simplicity, a realistic and scalable approximation of practical test-based problems.

After updating S_{arch} and T_{arch} , the coevolutionary algorithm proceeds to the evaluation phase, in which we use the round robin tournament: each candidate solution from S' is given a fitness value computed as the number of tests it solves from T'. Analogically, the fitness of a test t from T' equals the number of candidate solutions from S' that do not solve t. This fitness determines the odds for an individual to pass the subsequent selection, which is implemented as tournament [51] of size 2. The selection of candidate solutions proceeds independently from the selection of tests.

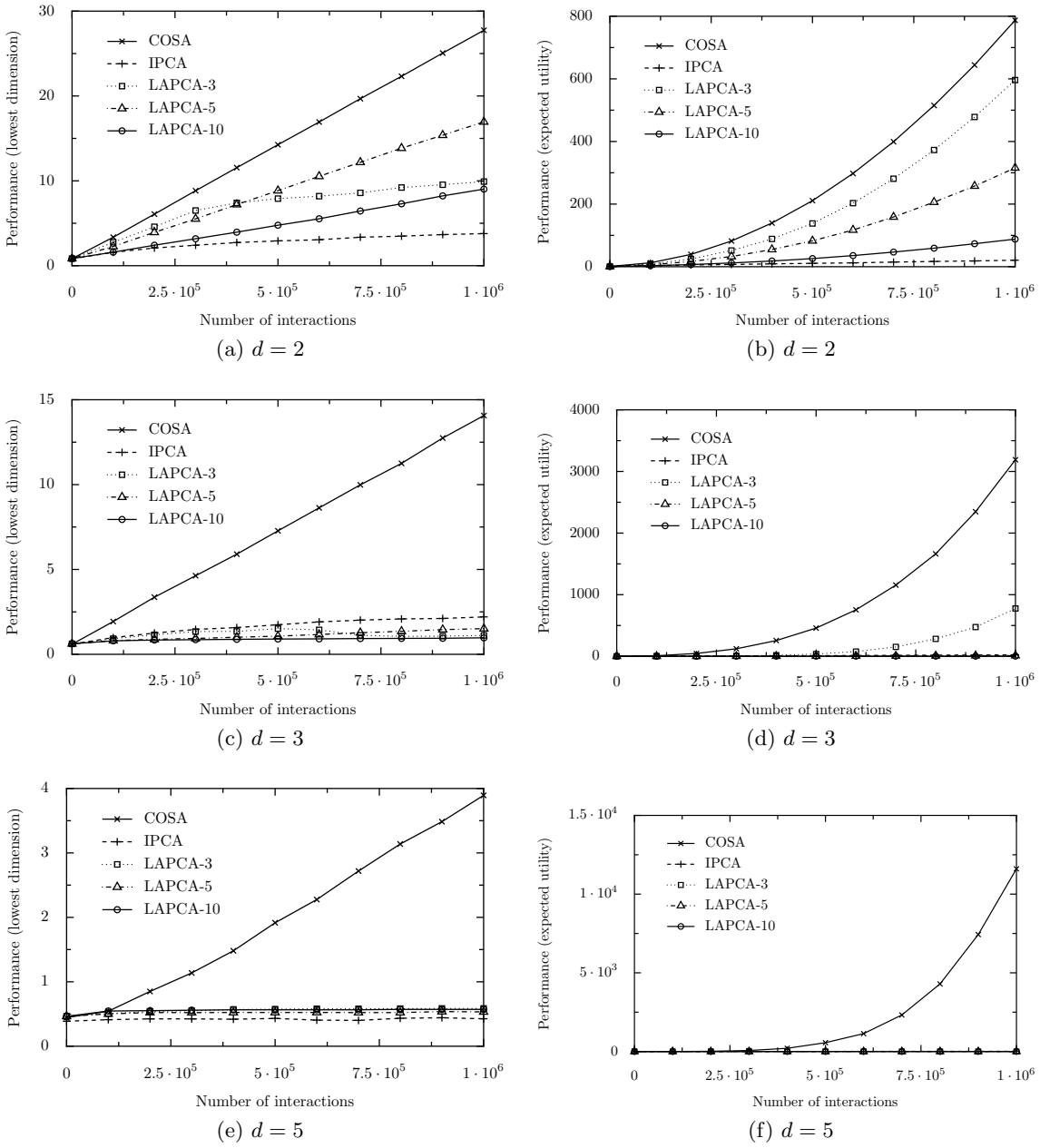
The coevolutionary algorithm terminates when the total number of interactions (games played) reaches $1,000,000$. Note that the actual number of elapsed generations can be, and usually is, different for particular algorithms and runs, because the number of interactions depends on the size of the archive, which varies over time.

6.4. Results

6.4.1. Compare-on-One

Charts presented in Figs. 6.4.1 and 6.4.2 summarize the results of the conducted experiment for COMPARE-ON-ONE for a priori dimensions $d = 2$, 3, and 5. LAPCA has been run for $l = 3, 5, 10$ layers, to give it a chance of attaining good performance. The left column of charts in Fig. 6.4.1 depicts the performance of the algorithm expressed by the lowest dimension of the best candidate solution in the archive, and the right one — the expected utility of the best candidate solution in the archive. In Fig. 6.4.2, all charts visualize the sizes of archives, left-hand charts for candidate solutions, right-hand charts for tests. All charts present the averages of 10 runs.

The superiority of COSA for this problem is evident and spans all considered problem instances and performance measures. In particular, COSA is able to make steady progress in terms of the lowest dimension of the candidate solutions. The other algorithms lag far behind, particularly for harder instances (larger d). LAPCA is quite good for $d = 2$, and the best version of LAPCA seems to be the one with 5 layers (LAPCA-3 is initially better, but then slows down). For $d = 2$, LAPCA-10 is slow, but still faster than IPCA. The things change for $d = 3$, when IPCA gets better than all LAPCAs. For $d = 5$ no substantial progress in the lowest dimension is observed either for IPCA or LAPCA. All in all, COSA copes with over-specialization better than the other considered methods.

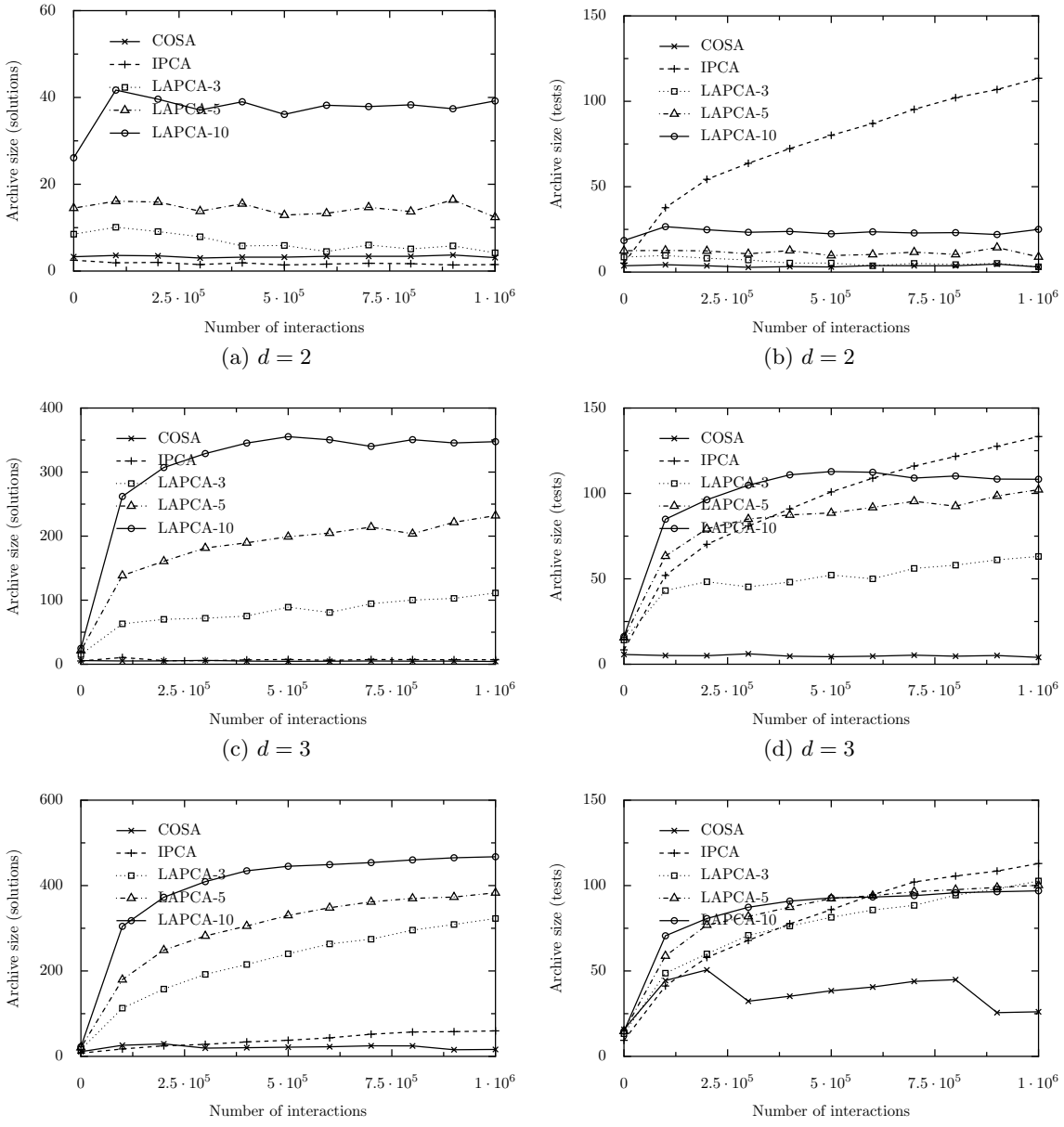


(a) $d = 2$ (b) $d = 2$

(c) $d = 3$ (d) $d = 3$

(e) $d = 5$ (f) $d = 5$

Figure 6.4.1.: Results for d-dimensional COMPARE-ON-ONE ($d = 2, 3, 5$) for two performance measures: lowest dimension (left column) and expected utility (right column).



(a) $d = 2$ (b) $d = 2$

(c) $d = 3$ (d) $d = 3$

(e) $d = 5$ (f) $d = 5$

Figure 6.4.2.: Archive sizes for d-dimensional COMPARE-ON-ONE ($d = 2, 3, 5$).

Dimension	COSA	IPCA	LAPCA-3	LAPCA-5	LAPCA-10
2	980.9±1.8	407.7±15.0	915.3±53.0	763.0±10.3	488.4±18.7
3	928.1±2.3	343.9±22.2	380.6±256.1	133.2±82.5	51.3±33.7
5	705.6±251.3	344.6±35.7	104.5±18.1	59.1±7.9	41.2±5.5

Table 6.1.: Average number of generations that the algorithms were able to process during 1,000,000 interactions for COMPARE-ON-ONE.

The expected utility of the best evolved candidate solution also votes in favor of COSA. The probability of a test being solved by the best candidate solution in the archive is highest for COSA. The progress on this criterion looks polynomial for all algorithms, because the hypervolume of the polyhedron defined in 6.3.3 for COMPARE-ON-ONE depends on the d^{th} power of the largest dimensions of a candidate solution. For the same reason, the absolute values of this measure get much higher when increasing the problem a priori dimension.

From the viewpoint of archive sizes, COSA is also among the leaders. For candidate solutions, it is neck and neck with IPCA, and together they beat LAPCA. For $d = 5$, COSA seems to noticeably subdue IPCA in the later phases of the search. Strikingly, for $d = 2$ and 3 the number of candidate solutions stored in the COSA archive is low and does not increase over time. For test archives, the differences between methods are even more evident: this time, even IPCA is unable to keep pace with COSA, which is not surprising as IPCA never discards any older tests, while COSA stores only one test per dimension and a few additional tests to separate candidate solutions in the Pareto layer. Our algorithm manages to maintain the lowest number of tests, which is usually only a small fraction of the archives of other methods. Most importantly, COSA's archive sizes remain virtually constant over time and there is no reason to doubt in further maintenance of this behavior. At the same time, other archives' sizes keep growing and do not seem to saturate, which at some stage may render them useless.

The fact that IPCA is better than LAPCA for $d = 3$ and that, in general, LAPCA performs so poorly is surprising, since it contradicts the findings of earlier research [30], where IPCA was found worse than LAPCA on the same problem. The charts in Fig. 6.4.2 suggest that the presumed reason for that is an excessive growth of LAPCA's archive, because the Pareto layers contain many non-dominated candidate solutions. This, in turn, could be caused by the absence of crossover in our setup (de Jong [30] used two-point crossover with 50% probability).

As mentioned earlier, the number of iterations processed by the coevolutionary loop

Dimension	COSA	IPCA	LAPCA-3	LAPCA-5	LAPCA-10
2	756.4±44.2	420.0±23.4	902.9±33.4	770.8±25.6	506.8±31
3	361.5±280.3	360.2±21.1	393.4±146.2	304.5±107.7	139.5±99.6
5	170.2±208.9	333.9±39.5	105.2±32.7	60.2±13.5	31.7±50

Table 6.2.: Average number of generations that the algorithms were able to process during 1,000,000 interactions for COMPARE-ON-ALL.

(Alg. 2.1) depends on the sizes of archives. We illustrate this dependency in Table 6.1, where we report the average number of generations that each algorithm went through. The presented numbers resonate with the charts: because COSA is able to maintain small and approximately constant-sized archives, its run length measured in generations is the highest and does not seem to be affected by the a priori dimension of the problem d, which is not the case for LAPCA. This is critical, as the number of iterations is also the number of generator invocations, which are the only source of variability for the search process.

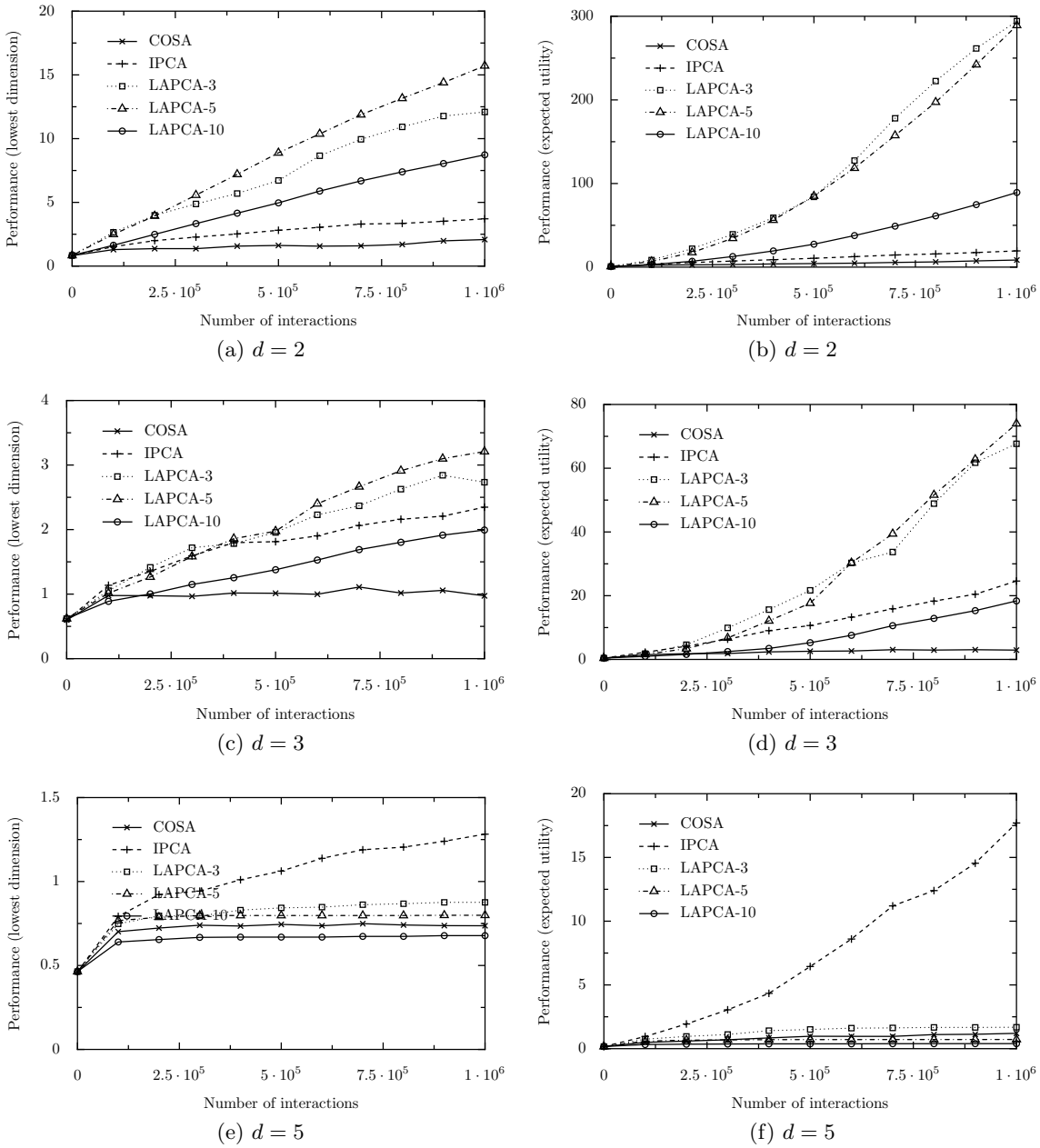
6.4.2. Compare-on-All

The results obtained for the COMPARE-ON-ALL problem are presented in Fig. 6.4.3. Contrary to the results presented for COMPARE-ON-ONE, the figures show that COSA performs worse than other methods on this problem. Virtually no progress is observed for any performance measure and any of the examined value of d. Other archive methods perform much better: LAPCA-3 and LAPCA-5 work best for the 2 and 3-dimensional problems, but they lose in favor of IPCA for a 5-dimensional game.

Fig. 6.4.4 shows that COSA maintains a much larger archive than it was in case of COMPARE-ON-ONE. The solution archive contains 10, 60 and 80 individuals for $d = 2, 3$ and 5, respectively, while it was always less than 10 for the COMPARE-ON-ONE problem. This is reflected in the total number of generations that COSA was able to process during 1,000,000 interactions (see Table 6.2), which is much lower than it was for COMPARE-ON-ONE (cf. Table 6.1).

6.5. Discussion and Summary

In this chapter we proposed Coordinate System Archive, a novel coevolutionary archive, based on the concepts of underlying problem structure and coordinate systems. We verified COSA on two popular benchmarks: COMPARE-ON-ONE and COMPARE-ON-ALL

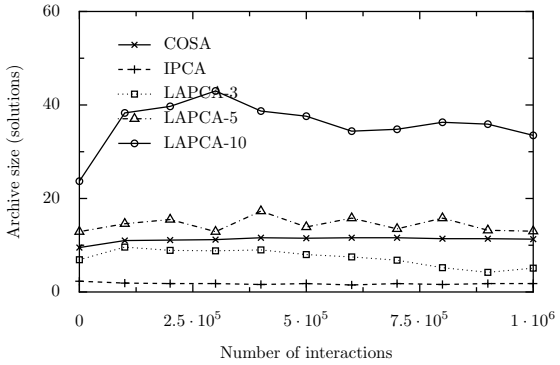


(a) $d = 2$ (b) $d = 2$

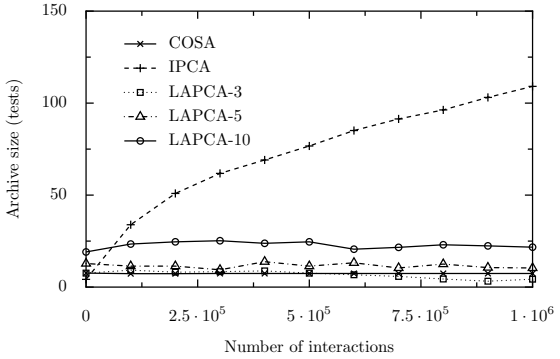
(c) $d = 3$ (d) $d = 3$

(e) $d = 5$ (f) $d = 5$

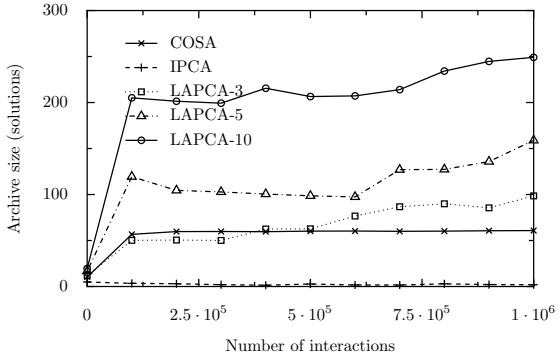
Figure 6.4.3.: Results for d-dimensional COMPARE-ON-ALL ($d = 2, 3, 5$) for two performance measures: lowest dimension (left column) and expected utility (right column).



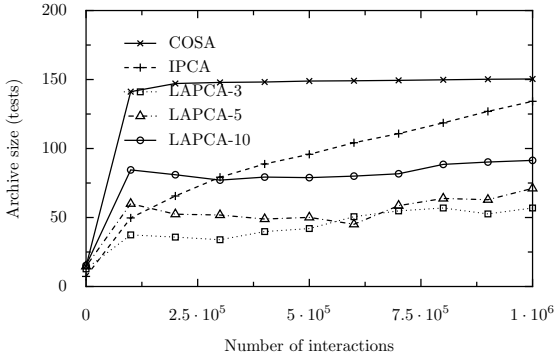
(a) $d = 2$



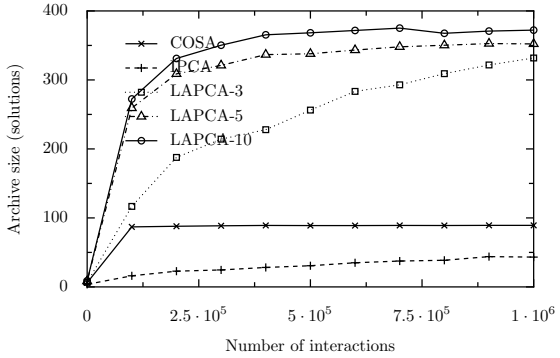
(b) $d = 2$



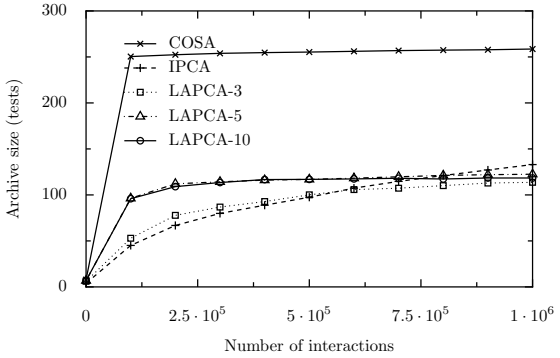
(c) $d = 3$



(d) $d = 3$



(e) $d = 5$



(f) $d = 5$

Figure 6.4.4.: Archive sizes for d-dimensional COMPARE-ON-ALL ($d = 2, 3, 5$).

number games with two progress measures corresponding to two different solution concepts. It is, generally, agreed that COMPARE-ON-ONE is a harder problem [34] than COMPARE-ON-ALL. Thus, it may seem surprising at the first glance that COSA performs the best on the former problem and the worst on the latter one. In order to explain this phenomenon, we have to refer to results of Chapter 5. There, we have demonstrated that algorithms for extracting the coordinate system defined by Bucci et al. properly approximate the *a priori* dimension of COMPARE-ON-ONE (Sections 5.9.1) but fail to find the *a priori* dimension of COMPARE-ON-ALL (Section 5.9.2). Since COSA is based on the same definition of coordinate system, its results on these two problems are understandable.

In this perspective, despite being an alternative for such methods as IPCA or LAPCA, and a step towards practical methods effectively solving test-based problems, COSA has serious drawbacks in being unable to succeed on COMPARE-ON-ALL. Future research should make an attempt to answer the question whether real-world test-based problems resemble rather COMPARE-ON-ONE or COMPARE-ON-ALL.

Moreover, COSA has not been tested on harder problems considered in Chapter 5: density classification task and tic-tac-toe. As we have seen in Chapter 5, the former problem has been found to have very large dimension, which grows linearly with respect to the number of tests involved to estimate it, and COSA, by design, is not a good method to cope with it. The dimension of tic-tac-toe, on the other hand, is reasonably large (several thousand), which, however, still is too computationally demanding for COSA. That is why the future work on COSA should answer the question how to determine the most important axes in an extracted coordinate system. As a result, in order to keep the archive small, only tests from those axes could be retained.

7. Conclusions

7.1. Summary

The class of test-based problems embraces tasks from many disciplines and fields. Competitive coevolution is a general approach for solving such tasks. The aggregation step, typically employed to compute individual's fitness in the evaluation phase of coevolutionary algorithm, makes the dynamics of search process complex and hard to predict. The results gathered in this dissertation demonstrate not only that it is possible to avoid such aggregation, but also that algorithms can be designed that perform well despite bypassing the aggregation step.

Studying the dynamics of bio-inspired algorithms is interesting from the perspective of artificial life discipline [82], which aims at imitating traditional biology and recreating the biological phenomena withing computers. In contrast, computational intelligence puts emphasis on *utilizing* computational principles observed in nature for solving problems. As coevolution, in the form as we know it from nature, exhibits many behaviors that are undesired from the viewpoint of problem solving, coevolutionary algorithms need to be armed with some, not necessarily bio-inspired, mechanisms to turn them into effective methods for learning or optimization. We hope that Fitnessless Coevolution and COSA presented in this thesis, contribute to this research direction.

7.2. Contribution

The main contribution of this work might be summarized as follows:

- Introduction of Fitnessless Coevolution for symmetrical test-based problems. Fitnessless Coevolution due to its fitnessless selection is a novel coevolutionary algorithm which does not use an explicit fitness measure. It was evaluated experimentally on a set of simple problems, with the outcomes indicating that it is at least as good as other, state-of-the-art methods in coevolution.
  [Chapter 3]

- The proof that, under a condition of transitivity of the payoff matrix, Fitnessless Coevolution is dynamically equivalent to the traditional evolutionary algorithm using tournament selection. Luke and Wiegand showed that a single-population coevolutionary algorithm is dynamically equivalent to an evolutionary algorithm [91], but they did it under a different set of conditions and did not demonstrate any practical algorithm meeting those conditions.
  [Theorem 19 in Chapter 3]

- A case-study of an application of Fitnessless Coevolution to a game of imperfect information. An exhaustive evaluation of the results obtained by Fitnessless Coevolution for the Ant Wars game and an in-depth analysis of interesting behavioral patterns of the best obtained solution was provided.
  [Chapter 4]

- An in-depth analysis of coordinate system for test-based problems. The formal definition of coordinate system was provided by Bucci *et al.* [17], and here it was analyzed in detail. This includes (Chapter 5):

  - Proposing an alternative definition of the coordinate system, equivalent to the original one.
    [Definition 27]
  - Proving several useful and interesting properties of coordinate systems.
    [Theorem 32; Propositions 31 and 34; Corollaries 33, 35 and 39]
  - Showing the link between dimension of a test-based problem and the width of an underlying poset.
    [Theorem 36; Proposition 37]
  - Estimating the lower and upper bounds for the dimension of a test-based problem.
    [Theorem 40]
  - Proving that finding the minimal coordinate system for a test-based problem is NP-hard, which was earlier conjectured [15], but never shown formally.
    [Theorem 46]

- Design, analysis and experimental evaluation of three algorithms for constructing correct coordinate systems for test-based problems in Section 5.8. This includes:

  - Proving that the SimpleGreedy algorithm proposed in [17] is correct.
    [Proposition 49]

  - The first exact algorithm for finding the minimal coordinate system and a GreedyCover heuristic. GreedyCover is as quick as SimpleGreedy, but produces much better, near-optimal results.
    [Algorithms 5.2 and 5.3]

- Demonstration that problem dimension is typically much lower than the number of tests
  [Section 5.9]

- Proposal of the COSA algorithm for test-based problems, which serves as a proof-of-concept that the definition of the coordinate systems analyzed in Chapter 5 may have practical applications. It was demonstrated that COSA performs well for problems for which it is easy to approximate their a priori dimension (compare-on-one) and does not perform well for others (compare-on-all).
  [Chapter 6]

7.3. Future Work

The work presented in this thesis may be extended in many directions, some of which have been already discussed in chapter summaries. Here we would like to add the following interesting ideas:

- Fitnessless Coevolution proposed in Chapter 3 has been designed for symmetric test-based problems only. The questions i) whether this method could be extended to asymmetric test-based problems and ii) whether its theoretical properties will hold after such a generalization, might be important to answer.

- Results of Chapter 5 indicate that, despite the fact that the set of tests can be exchanged for a set of underlying objectives, the number of underlying objectives in many test-based problems (the problem dimension) remains still large. High-dimensional coordinate systems are impractical, thus methods of further reduction of underlying objectives should be designed. One of the conceivable approaches is as follows. The coordinate system described in Chapter 5 preserves all relations between candidate solutions and tests. It seems that in order to have a tractable number of underlying objectives, one can relax this strict restriction and trade-off preserving all dominance relations for smaller number of axes in the coordinate system of a problem. A similar problem (but on a smaller scale) manifests in many-objective optimization [13], thus some methods from that field could be potentially

adapted.

Another possibility to reduce the number of underlying objectives is to relax the strict dominance relation in favor of an approximate dominance measure such as ϵ-dominance, proposed by Laumanns et al. [83], and to build the coordinate system on top of it. However, a drawback of such an approach would be that, in contrast to the proper dominance relation, any approximate dominance relation is, in general, non-transitive.

- Coevolutionary algorithms are computationally demanding since the number of interactions in each generation depends not on the total number of individuals in all populations, but on the *product* of cardinalities of particular populations. Thus, coupling coevolutionary methods with some less demanding algorithms such as local search techniques or temporal difference learning might provide a well-balanced trade-off between exploration and exploitation. The validity of this idea has been already confirmed to some extent [136, 80].

A. Appendix

A.1. Ants Obtained with Fitnessless Coevolution

A.1.1. BrilliAnt

```
1  int ant_Move(int **grid, int my_row, int my_column){
2  #define FORYXN for (y=0;y<11;y++) for (x=0;x<11;x++)
3  #define FORYX1(y1,y2,x1,x2) for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++)
4  #define FORYX(y1,y2,x1,x2,w) { for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++)
       if (G[(11+y)%11][(11+x)%11]==w) c+=H[(11+y)%11][(11+x)%11]; }
5  #define N(n,x1,y1,x2,y2,w) c=0; if (D==0) FORYX(y1,y2,x1,x2,w) else if (D==1)
       FORYX(x1,x2,-(y2),-(y1),w) else if (D==2) FORYX(-(y2),-(y1),-(x2),-(x1),w)
       else FORYX(-(x2),-(x1),y1,y2,w); f[n]=c;
6
7  #define FORYXV(y1,y2,x1,x2) { for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++) c
       +=V[(11+y)%11][(11+x)%11]; }
8  #define B(n,x1,y1,x2,y2) c=0; if (D==0) FORYXV(y1,y2,x1,x2) else if (D==1) FORYXV(
       x1,x2,-(y2),-(y1)) else if (D==2) FORYXV(-(y2),-(y1),-(x2),-(x1)) else FORYXV
       (-(x2),-(x1),y1,y2); f[n]=c;
9
10 #define MAXF(y,x) if (G[(R+11+y)%11][(C+11+x)%11]==1 && H[(R+11+y)%11][(C+11+x)
       %11]>s) s=H[(R+11+y)%11][(C+11+x)%11]
11 #define FOOD(y,x) (G[(R+11+y)%11][(C+11+x)%11]==1?H[(R+11+y)%11][(C+11+x)%11]:0)
12 #define RETF(a,b,x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6,x7,y7,x8,y8) if (D==a && d==b
       ) { MAXF(x1,y1);MAXF(x2,y2);MAXF(x3,y3);MAXF(x4,y4);MAXF(x5,y5);MAXF(x6,y6);
       MAXF(x7,y7); s+=FOOD(x8,y8); } else
13 #define F s=0; RETF(0,0,0,-1,-1,-1,-2,-1,-2,0,-2,1,-1,1,0,1,-1,0) RETF
       (0,1,0,-1,0,-2,-1,-2,-2,-2,-2,-1,-2,0,-1,0,-1,-1) RETF
       (1,0,-1,0,-1,1,-1,2,0,2,1,2,1,1,1,0,0,1) RETF
       (1,1,-1,0,-2,0,-2,1,-2,2,-1,2,0,2,0,1,-1,1) RETF
       (2,0,0,1,1,1,2,1,2,0,2,-1,1,-1,0,-1,1,0) RETF
       (2,1,0,1,0,2,1,2,2,2,2,1,2,0,1,0,1,1) RETF
       (3,0,1,0,1,-1,1,-2,0,-2,-1,-2,-1,-1,-1,0,0,-1) RETF
       (3,1,1,0,2,0,2,-1,2,-2,1,-2,0,-2,0,-1,1,-1);
14
15 static int T,P,G[11][11],V[11][11];
16 static float H[11][11];
17 register int x,y; int D,E,ret,R,C,d;
18 float r,best,s,c,f[37];
19 R=my_row;C=my_column;
20 for (x=0; x<11; ++x) for (y=0; y<x; ++y) { int t = grid[x][y]; grid[x][y] = grid[y
       ][x]; grid[y][x] = t; } /* hotfix due to the coordinates confusion */
```

```
21 D = 0;
22 E = (grid[R][C]==10)?100:10;
23 V[R][C]=1; FORYXN if (G[y][x]==1) H[y][x]*=0.9; else if (G[y][x]==E) G[y][x]=0;
       FORYX1(-2,2,-2,2) {G[(11+y)%11][(11+x)%11] = grid[(11+y)%11][(11+x)%11]; H
       [(11+y)%11][(11+x)%11]=1;}
24 best=0;
25 ret=-1;
26 for (;D<4;++D) {
27 d=0;
28 F N(0,-1,-3,1,-2,E) N(1,-3,-2,0,0,E) N(2,-1,-1,0,1,E) N(3,0,-3,1,-1,1) N
       (4,-1,-3,0,-2,1) N(5,-3,-4,0,-1,1) N(6,0,-1,1,1,E) N(7,-3,-3,0,-1,0) N
       (8,0,-1,2,0,E) N(9,-1,-2,0,-1,1) N(10,-3,-3,1,-1,E) N(11,-3,-2,0,0,E) N
       (12,-3,-2,0,0,E) N(13,-1,-1,0,1,E) N(14,-1,-1,1,0,E) N(15,-2,-3,-1,-2,1) N
       (16,0,-1,4,2,E) N(17,-1,-3,1,-2,E) N(18,1,-3,5,-1,E) N(19,-1,0,0,1,1) N
       (20,-1,-2,1,-1,E) N(21,-3,-2,-1,2,E) N(22,1,0,3,1,E) N(23,-3,-2,0,0,E) N
       (24,-3,-4,-1,0,0) N(25,-3,-1,-1,0,0) N(26,-2,-3,0,-1,1) N(27,-2,-1,0,0,E) N
       (28,-1,0,0,2,E) N(29,-3,-3,-1,-1,0) N(30,0,-3,1,-1,1) N(31,-1,-1,1,0,E) N
       (32,-1,-1,1,0,E) N(33,-1,-2,0,-1,1) N(34,-1,-3,1,-1,E) N(35,0,-2,1,-1,1)
29 r=((!((f[0])||((f[1])&&(f[2]))))?((f[3])?((s)*((f[4])?(f[5]):(((s)+(s))+((35-T)*(s
       ))))):(((((s)+(s))+((35-T)*(s)))-(15-P))+((f[6])?(((35-T)*(s))-(15-P)):(((5)
       +(15-P))+(-f[7])))))):((P)+(((f[8])&&(!(f[9])))?(((35-T)+(35-T))*(35-T)):(2))))
       +((((f[10])&&((f[11])&&(!(f[12]))))&&(((f[13])||((f[14])||(f[15])))&&(f[16])))
       ?(((f[17])&&((f[18])&&(f[19])))?(((!((15-P)==(P)))||((f[20])||(f[21])))?(((f
       [22])&&(f[23]))?((s)+(-f[24])):((35-T)+(-f[25]))):((s)+(s))):(((f[26])&&(!(f
       [27])))?((f[28])?((35-T)*(s)):(((5)+(15-P))+(-f[29]))):(15-P))):(((f[30])&&((f
       [31])||((f[32])||(f[33]))))?((f[34])?(P):((f[35])?((35-T)+((35-T)*(s))):(((5)
       +(15-P))+(3)))):(-0.31385064)));
30 if (r > best || ret == -1) { best=r; ret = 2*D+1; }
31 d=1;
32 F N(0,-1,-1,1,0,E) N(1,-1,-1,1,0,E) N(2,-1,-1,1,0,E) N(3,-4,-4,1,0,1) N
       (4,0,-4,1,-1,1) N(5,-1,-1,0,1,E) N(6,-1,-3,1,-2,E) N(7,-1,-3,1,-1,E) N
       (8,0,-4,1,-1,1) N(9,-1,-1,0,1,E) N(10,-3,-2,0,0,E) N(11,-1,-1,0,1,E) N
       (12,-3,-4,0,0,E) N(13,2,-5,5,-1,E) N(14,-1,-1,1,0,E) B(15,0,-6,5,-2) N
       (16,-1,-1,1,0,E) N(17,-1,-1,1,0,0) N(18,-2,-1,3,0,E) N(19,-3,-3,1,-1,1) N
       (20,-3,-4,0,0,E) N(21,-3,-3,0,-1,0) N(22,-2,-1,0,0,E) N(23,-1,-3,0,-2,1) N
       (24,-3,-4,0,0,E) N(25,0,-1,2,0,E) N(26,-1,-3,1,-1,E) N(27,-1,-1,1,0,E) N
       (28,-2,-1,0,0,E) N(29,-3,-2,0,0,E) N(30,0,-2,1,-1,1) N(31,-3,-3,0,-1,0) N
       (32,-3,-2,0,0,E) N(33,-4,-2,-2,3,0) N(34,-4,-2,-2,1,0) N(35,3,-5,4,-1,1) N
       (36,-3,-4,0,-1,1)
33 r=(((((!(f[0]))&&(f[1]))||(f[2]))||((f[3])||((f[4])&&((f[5])&&(f[6])))))?(((f[7])
       ||((f[8])||(f[9])))?(((f[10])&&(f[11]))?((35-T)*(35-T)):(3)):(4))
       :(-0.31385064))+(((f[12])?((!((f[13])&&(f[14])))?(-f[15]):((f[16])?(((5)+(15-P
       ))+(-f[17])):(15-P))):((((f[18])||(f[19]))&&(f[20]))?((P)+(15-P)):((35-T)+(-f
       [21]))))+(((((f[22])||(f[23]))&&(f[24]))&&((f[25])||((f[26])&&(f[27]))))?(((f
       [28])&&((f[29])&&(f[30])))?(-f[31]):(3)):((f[32])?(((-f[33])+(-f[34]))+(f[35])
       ):(((s)*((35-T)*(s)))+(((s)+(s))+(f[36]))))));
34 if (r > best) { best=r; ret = 2*D; }
35 }
36 ++T;
37 D=ret/2; N(0,ret%2-1,-1,ret%2-1,-1,1) if (f[0]) ++P;
38 return (8-ret)%8;
```

```
39 }
```

A.1.2. ExpertAnt

```
1  #define FORYXN for (y=0;y<11;y++) for (x=0;x<11;x++)
2  #define FORYX1(y1,y2,x1,x2) for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++)
3  #define FORYX(y1,y2,x1,x2) { for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++) if
       (G[(11+y)%11][(11+x)%11]==w) c+=H[(11+y)%11][(11+x)%11]; }
4  static float N(int D, int R, int C, int G[11][11], float H[11][11], int x1, int y1
       , int x2, int y2, int w) {
5  float c; register int y,x;
6  c=0;
7  if (D==0) FORYX(y1,y2,x1,x2) else if (D==1) FORYX(x1,x2,-(y2),-(y1)) else if (D
       ==2) FORYX(-(y2),-(y1),-(x2),-(x1)) else FORYX(-(x2),-(x1),y1,y2);
8  return c;
9  }
10
11 #define FORYXV(y1,y2,x1,x2) { for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++) c
       +=G[(11+y)%11][(11+x)%11]; }
12 static float B(int D, int R, int C, int G[11][11], int x1, int y1, int x2, int y2)
        {
13 int c; register int y,x;
14 c=0;
15 if (D==0) FORYXV(y1,y2,x1,x2) else if (D==1) FORYXV(x1,x2,-(y2),-(y1)) else if (D
       ==2) FORYXV(-(y2),-(y1),-(x2),-(x1)) else FORYXV(-(x2),-(x1),y1,y2);
16 return c;
17 }
18
19 #define MAXF(y,x) if (G[(R+11+y)%11][(C+11+x)%11]==1 && H[(R+11+y)%11][(C+11+x)
       %11]>s) s=H[(R+11+y)%11][(C+11+x)%11]
20 #define FOOD(y,x) (G[(R+11+y)%11][(C+11+x)%11]==1?H[(R+11+y)%11][(C+11+x)%11]:0)
21 static float F(int D, int d, int R, int C, int G[11][11], float H[11][11]) {
22 float s; s=0;
23 if (D==0 && d==0) { MAXF(0,-1);MAXF(-1,-1);MAXF(-2,-1);MAXF(-2,0);MAXF(-2,1);MAXF
       (-1,1);MAXF(0,1); return s+FOOD(-1,0); }
24 if (D==0 && d==1) { MAXF(0,-1);MAXF(0,-2);MAXF(-1,-2);MAXF(-2,-2);MAXF(-2,-1);MAXF
       (-2,0);MAXF(-1,0); return s+FOOD(-1,-1); }
25 if (D==1 && d==0) { MAXF(-1,0);MAXF(-1,1);MAXF(-1,2);MAXF(0,2);MAXF(1,2);MAXF(1,1)
       ;MAXF(1,0); return s+FOOD(0,1); }
26 if (D==1 && d==1) { MAXF(-1,0);MAXF(-2,0);MAXF(-2,1);MAXF(-2,2);MAXF(-1,2);MAXF
       (0,2);MAXF(0,1); return s+FOOD(-1,1); }
27 if (D==2 && d==0) { MAXF(0,1);MAXF(1,1);MAXF(2,1);MAXF(2,0);MAXF(2,-1);MAXF(1,-1);
       MAXF(0,-1); return s+FOOD(1,0); }
28 if (D==2 && d==1) { MAXF(0,1);MAXF(0,2);MAXF(1,2);MAXF(2,2);MAXF(2,1);MAXF(2,0);
       MAXF(1,0); return s+FOOD(1,1); }
29 if (D==3 && d==0) { MAXF(1,0);MAXF(1,-1);MAXF(1,-2);MAXF(0,-2);MAXF(-1,-2);MAXF
       (-1,-1);MAXF(-1,0); return s+FOOD(0,-1); }
30 { MAXF(1,0);MAXF(2,0);MAXF(2,-1);MAXF(2,-2);MAXF(1,-2);MAXF(0,-2);MAXF(0,-1);
       return s+FOOD(1,-1); }
31 }
32
```

```
33 int ant_Move(int **grid, int my_row, int my_column) {
34 static int T,P,G[11][11],V[11][11];
35 static float H[11][11];
36 register int x,y; int D,E,ret,R,C,d;
37 float r,best;
38 R=my_row;C=my_column;
39 if (C == -1) { return 0; }
40 if (R == -1) { T=P=0; FORYXN { V[y][x]=G[y][x]=0;H[y][x]=0;} return -1; }
41 D = 0;
42 E = (grid[R][C]==10)?100:10;
43 V[R][C]=1; FORYXN if (G[y][x]==1) H[y][x]*=0.9; else if (G[y][x]==E) G[y][x]=0;
       FORYX1(-2,2,-2,2) {G[(11+y)%11][(11+x)%11] = grid[(11+y)%11][(11+x)%11]; H
       [(11+y)%11][(11+x)%11]=1;}
44 best=0;
45 ret=-1;
46 for (;D<4;++D) {
47 d=0;
48 r=((!((N(D,R,C,G,H,-1,-3,1,-2,E))||((N(D,R,C,G,H,-3,-2,0,0,E))&&((N(D,R,C,G,H
       ,-3,-2,0,0,E))&&(N(D,R,C,G,H,-1,-1,0,1,E))))))?(((N(D,R,C,G,H,0,-2,1,-1,1))
       ||((N(D,R,C,G,H,-1,-1,1,0,E))&&(!(N(D,R,C,G,H,-3,-3,0,-2,E)))))?((F(D,d,R,C,G,
       H))*((N(D,R,C,G,H,-1,-3,0,-2,1))?(N(D,R,C,G,H,-3,-4,0,-1,1)):(((5)+(15-P))
       +((35-T)*(F(D,d,R,C,G,H))))):(((((F(D,d,R,C,G,H))+(F(D,d,R,C,G,H)))+((35-T)*(
       F(D,d,R,C,G,H))))-(15-P))+(((4)==(35-T))?((F(D,d,R,C,G,H))-(15-P)):(((5)+(15-P
       ))+(N(D,R,C,G,H,-3,-4,0,-1,1))))))):((P)+(((N(D,R,C,G,H,0,-1,2,0,E))&&(!(N(D,R,
       C,G,H,-1,-2,0,-1,1))))?(((35-T)+(35-T))*(35-T)):(2))))+(((N(D,R,C,G,H
       ,-1,-1,1,0,E))&&(((N(D,R,C,G,H,-2,-3,0,-2,E))||((N(D,R,C,G,H,-1,-1,1,0,E))||(N
       (D,R,C,G,H,-1,-2,0,-1,1))))&&(N(D,R,C,G,H,0,-1,4,2,E))))?((((N(D,R,C,G,H
       ,-1,-1,1,0,E))||(N(D,R,C,G,H,-1,-1,1,0,E)))&&(!((N(D,R,C,G,H,-4,-6,-1,-1,1))
       ==(15-P))))?(((N(D,R,C,G,H,-1,-1,1,0,E))||(N(D,R,C,G,H,-1,-1,0,0,1)))?(((N(D,R
       ,C,G,H,0,-1,2,0,E))&&(N(D,R,C,G,H,-3,-2,0,0,E)))?((N(D,R,C,G,H,-3,-4,0,-1,1))
       +(-N(D,R,C,G,H,-4,-2,-2,1,0))):(N(D,R,C,G,H,-3,-2,-2,1,1))):((35-T)*(F(D,d,R,C
       ,G,H)))):((((N(D,R,C,G,H,-1,-1,1,0,E))||(N(D,R,C,G,H,1,-3,2,-1,E)))&&((N(D,R,C
       ,G,H,-3,-2,0,0,E))&&(N(D,R,C,G,H,-1,-1,1,0,E))))?((N(D,R,C,G,H,-3,-2,0,0,E))
       ?(-N(D,R,C,G,H,-3,-3,0,-1,0)):((N(D,R,C,G,H,-3,-4,0,-1,1))+(-N(D,R,C,G,H
       ,-3,-3,0,-1,0)))):(15-P))):(((N(D,R,C,G,H,0,-3,1,-1,1))&&((N(D,R,C,G,H
       ,-1,-1,1,0,E))||((N(D,R,C,G,H,-1,-1,1,0,E))||(N(D,R,C,G,H,-1,-2,0,-1,1))))))?((
       N(D,R,C,G,H,-1,-3,1,-1,E))?(P):((N(D,R,C,G,H,0,-2,1,-1,1))?((35-T)+((35-T)*(F(
       D,d,R,C,G,H)))):(((5)+(15-P))+(3)))):(-0.31385064)));
49 if (r > best || ret == -1) { best=r; ret = 2*D+1; }
50 d=1;
51 r=((((((N(D,R,C,G,H,0,-4,1,-1,1))||(N(D,R,C,G,H,-1,-2,0,-1,1)))&&(N(D,R,C,G,H
       ,-3,-2,0,0,E)))||((N(D,R,C,G,H,-1,-1,1,0,E))&&(N(D,R,C,G,H,-1,-4,4,0,E))))||((
       N(D,R,C,G,H,-4,-4,1,0,1))||((N(D,R,C,G,H,-2,-1,0,0,E))&&(N(D,R,C,G,H
       ,-3,-2,0,0,E)))))?((((N(D,R,C,G,H,-1,-1,0,1,E))||(N(D,R,C,G,H,-2,-5,-1,-3,E)))
       ||((N(D,R,C,G,H,0,-4,1,-1,1))||(N(D,R,C,G,H,0,-4,1,-1,1))))?(((N(D,R,C,G,H
       ,-1,-1,1,0,E))&&((N(D,R,C,G,H,0,-4,1,-1,1))||(N(D,R,C,G,H,-3,-2,0,0,E))))
       ?((35-T)*(35-T)):(3)):(4)):(-0.31385064))+(((N(D,R,C,G,H,-3,-4,0,0,E))?((!((N(
       D,R,C,G,H,-1,-4,0,-1,1))&&(N(D,R,C,G,H,1,-3,2,-1,E))))?(-B(D,R,C,V,0,-6,5,-2))
       :((N(D,R,C,G,H,-1,-5,1,-1,1))?(((5)+(15-P))+(N(D,R,C,G,H,2,-5,5,-3,1)))
       :(-0.6274291))):((N(D,R,C,G,H,-3,-4,0,0,E))?((35-T)+(((35-T)*(F(D,d,R,C,G,H)))
       +((35-T)+(35-T)))):((35-T)+(-N(D,R,C,G,H,-3,-3,0,-1,0)))))+(((N(D,R,C,G,H
```

```
      ,-2,-2,0,0,E))&&((N(D,R,C,G,H,-1,-1,1,0,E))||(N(D,R,C,G,H,-1,-1,1,0,E))))?((F(
      D,d,R,C,G,H))*((N(D,R,C,G,H,-3,-2,0,0,E))?((N(D,R,C,G,H,-3,-2,0,0,E))?(P):((F(
      D,d,R,C,G,H))+(F(D,d,R,C,G,H)))):(((35-T)*(F(D,d,R,C,G,H)))+(F(D,d,R,C,G,H))))
      ):((N(D,R,C,G,H,-3,-2,0,0,E))?(((N(D,R,C,G,H,-3,-4,0,-1,1))+(-N(D,R,C,G,H
      ,-4,-2,-2,1,0)))+((N(D,R,C,G,H,-4,-5,-2,0,1))+(-N(D,R,C,G,H,-2,0,2,4,0)))):(((
      F(D,d,R,C,G,H))*((35-T)*(F(D,d,R,C,G,H))))+(((F(D,d,R,C,G,H))+(F(D,d,R,C,G,H))
      )+(N(D,R,C,G,H,-3,-4,0,-1,1)))))));
52  if (r > best) { best=r; ret = 2*D; }
53  }
54  ++T;
55  D=ret/2; if (N(D,R,C,G,H,ret%2-1,-1,ret%2-1,-1,1)) ++P;
56  return ret;
57  }
```

A.1.3. EvolAnt1

```
1   #define FORYXN for (y=1;y<11;y++) for (x=0;x<11;x++)
2   #define FORYX1(y1,y2,x1,x2) for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++)
3   #define FORYX(y1,y2,x1,x2) { for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++) if
        (G[(11+y)%11][(11+x)%11]==w) c+=H[(11+y)%11][(11+x)%11]; }
4   static float N(int D, int R, int C, int G[11][11], float H[11][11], int x1, int y1
        , int x2, int y2, int w) {
5   float c; register int y,x;
6   c=0;
7   if (D==0) FORYX(y1,y2,x1,x2) else if (D==1) FORYX(x1,x2,-(y2),-(y1)) else if (D
        ==2) FORYX(-(y2),-(y1),-(x2),-(x1)) else FORYX(-(x2),-(x1),y1,y2);
8   return c;
9   }
10
11  #define FORYXV(y1,y2,x1,x2) { for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++) c
        +=G[(11+y)%11][(11+x)%11]; }
12  static float B(int D, int R, int C, int G[11][11], int x1, int y1, int x2, int y2)
         {
13  int c; register int y,x;
14  c=0;
15  if (D==0) FORYXV(y1,y2,x1,x2) else if (D==1) FORYXV(x1,x2,-(y2),-(y1)) else if (D
        ==2) FORYXV(-(y2),-(y1),-(x2),-(x1)) else FORYXV(-(x2),-(x1),y1,y2);
16  return c;
17  }
18
19  #define MAXF(y,x) if (G[(R+11+y)%11][(C+11+x)%11]==1 && H[(R+11+y)%11][(C+11+x)
        %11]>s) s=H[(R+11+y)%11][(C+11+x)%11]
20  #define FOOD(y,x) (G[(R+11+y)%11][(C+11+x)%11]==1?H[(R+11+y)%11][(C+11+x)%11]:0)
21  static float F(int D, int d, int R, int C, int G[11][11], float H[11][11]) {
22  float s; s=0;
23  if (D==0 && d==0) { MAXF(0,-1);MAXF(-1,-1);MAXF(-2,-1);MAXF(-2,0);MAXF(-2,1);MAXF
        (-1,1);MAXF(0,1); return s+FOOD(-1,0); }
24  if (D==0 && d==1) { MAXF(0,-1);MAXF(0,-2);MAXF(-1,-2);MAXF(-2,-2);MAXF(-2,-1);MAXF
        (-2,0);MAXF(-1,0); return s+FOOD(-1,-1); }
25  if (D==1 && d==0) { MAXF(-1,0);MAXF(-1,1);MAXF(-1,2);MAXF(0,2);MAXF(1,2);MAXF(1,1)
        ;MAXF(1,0); return s+FOOD(0,1); }
```

```
26 if (D==1 && d==1) { MAXF(-1,0);MAXF(-2,0);MAXF(-2,1);MAXF(-2,2);MAXF(-1,2);MAXF
       (0,2);MAXF(0,1); return s+FOOD(-1,1); }
27 if (D==2 && d==0) { MAXF(0,1);MAXF(1,1);MAXF(2,1);MAXF(2,0);MAXF(2,-1);MAXF(1,-1);
       MAXF(0,-1); return s+FOOD(1,0); }
28 if (D==2 && d==1) { MAXF(0,1);MAXF(0,2);MAXF(1,2);MAXF(2,2);MAXF(2,1);MAXF(2,0);
       MAXF(1,0); return s+FOOD(1,1); }
29 if (D==3 && d==0) { MAXF(1,0);MAXF(1,-1);MAXF(1,-2);MAXF(0,-2);MAXF(-1,-2);MAXF
       (-1,-1);MAXF(-1,0); return s+FOOD(0,-1); }
30 { MAXF(1,0);MAXF(2,0);MAXF(2,-1);MAXF(2,-2);MAXF(1,-2);MAXF(0,-2);MAXF(0,-1);
       return s+FOOD(1,-1); }
31 }
32
33 int ant_Move(int **grid, int my_row, int my_column) {
34 static int T,P,G[11][11],V[11][11];
35 static float H[11][11];
36 register int x,y; int D,E,ret,R,C,d;
37 float r,best;
38 R=my_row;C=my_column;
39 if (C == -1) { return 0; }
40 if (R == -1) { T=P=0; FORYXN { V[y][x]=G[y][x]=0;H[y][x]=0;} return -1; }
41 D = 0;
42 E = (grid[R][C]==10)?100:10;
43 V[R][C]=1; FORYXN if (G[y][x]==1) H[y][x]*=0.9; else if (G[y][x]==E) G[y][x]=0;
       FORYX1(-2,2,-2,2) {G[(11+y)%11][(11+x)%11] = grid[(11+y)%11][(11+x)%11]; H
       [(11+y)%11][(11+x)%11]=1;}
44 best=0;
45 ret=-1;
46 for (;D<4;++D) {
47 d=0;
48 r=((!((N(D,R,C,G,H,-1,-3,1,-2,E))||(((N(D,R,C,G,H,1,-5,6,0,E))&&(N(D,R,C,G,H
       ,-1,-3,1,-2,E)))||(N(D,R,C,G,H,-1,-1,1,0,E)))))?((N(D,R,C,G,H,0,-3,1,-1,1))?((
       F(D,d,R,C,G,H))*((N(D,R,C,G,H,-1,-4,0,-2,1))?(N(D,R,C,G,H,-3,-4,0,-1,1))
       :(((35-T)*(F(D,d,R,C,G,H)))+(F(D,d,R,C,G,H))))):((((35-T)*(F(D,d,R,C,G,H)))
       -(15-P))+((N(D,R,C,G,H,-1,-1,1,0,E))?(((35-T)*(F(D,d,R,C,G,H)))+(F(D,d,R,C,G,H
       ))):(((5)+(15-P))+(-N(D,R,C,G,H,-3,-3,0,-1,0))))))):((-N(D,R,C,G,H,-2,-3,0,0,0)
       )+(((N(D,R,C,G,H,0,-1,2,0,E))&&(!(N(D,R,C,G,H,1,-5,6,0,E))))?(((35-T)+(35-T))
       *(35-T)):(3))))+(((((!(N(D,R,C,G,H,-1,-1,1,0,E)))&&((N(D,R,C,G,H,-5,-1,-1,3,E)
       )||(N(D,R,C,G,H,-4,-2,-1,2,E))))&&((N(D,R,C,G,H,-3,-4,2,0,1))||(N(D,R,C,G,H
       ,-1,-2,0,-1,1))))&&((N(D,R,C,G,H,-1,-1,1,0,E))&&(!(!(N(D,R,C,G,H,-1,-1,1,0,E))
       ))))?(((!(!(N(D,R,C,G,H,-1,-1,1,0,E))))&&((N(D,R,C,G,H,-1,-3,1,-2,E))&&(!(N(D,
       R,C,G,H,0,-5,5,0,E)))))?((N(D,R,C,G,H,-3,-4,2,0,1))?(((F(D,d,R,C,G,H))+(-N(D,R
       ,C,G,H,2,-2,3,0,0)))+(35-T)):((N(D,R,C,G,H,1,-4,2,-3,E))?(35-T):(3))):((N(D,R,
       C,G,H,0,1,1,3,E))?(F(D,d,R,C,G,H)):((F(D,d,R,C,G,H))*((35-T)+(-N(D,R,C,G,H
       ,-3,-3,0,-1,0))))))):(((N(D,R,C,G,H,0,-3,1,-1,1))&&((!(N(D,R,C,G,H
       ,-1,-2,0,-1,1)))||((N(D,R,C,G,H,-4,-3,-2,0,E))||(N(D,R,C,G,H,-1,-2,0,-1,1)))))
       ?((N(D,R,C,G,H,-1,-3,1,-1,E))?(4):((N(D,R,C,G,H,-1,-2,0,-1,1))?((35-T)+((35-T)
       *(F(D,d,R,C,G,H)))):((N(D,R,C,G,H,-3,-4,0,-1,1))+(3)))):(-0.31385064)));
49 if (r > best || ret == -1) { best=r; ret = 2*D+1; }
50 d=1;
51 r=(((((!(N(D,R,C,G,H,-3,-2,0,0,E)))&&(N(D,R,C,G,H,-1,-1,1,0,E)))||((N(D,R,C,G,H
       ,-3,-4,2,0,1))||(N(D,R,C,G,H,-1,-2,0,-1,E))))?((((N(D,R,C,G,H,0,-2,1,-1,E))||(
```

```
        N(D,R,C,G,H,0,-4,1,-1,1)))||((N(D,R,C,G,H,-1,-1,1,0,E))||(N(D,R,C,G,H
        ,1,-5,6,0,E))))?(((!(N(D,R,C,G,H,1,-1,4,0,E)))&&((N(D,R,C,G,H,-3,-2,0,0,E))&&(
        N(D,R,C,G,H,-1,-1,1,0,E))))?((35-T)*(35-T)):(3)):(4)):(-0.31385064))+(((N(D,R,
        C,G,H,-3,-4,0,0,E))?(((!(N(D,R,C,G,H,-4,-4,1,0,1)))||((35-T)==(N(D,R,C,G,H
        ,-2,-4,1,1,1))))?((N(D,R,C,G,H,-1,-2,0,-1,1))?((35-T)+(3)):((N(D,R,C,G,H
        ,-3,-4,0,-1,1))+(3))):((0.98849344)-((F(D,d,R,C,G,H))+(F(D,d,R,C,G,H))))):(((N
        (D,R,C,G,H,-1,-1,1,0,E))||(N(D,R,C,G,H,-1,-1,1,0,E)))?((N(D,R,C,G,H,-3,-2,0,0,
        E))?((35-T)+(35-T)):((F(D,d,R,C,G,H))+(-N(D,R,C,G,H,2,-2,3,0,0)))):((35-T)+(-N
        (D,R,C,G,H,-3,-3,0,-1,0))))):+((N(D,R,C,G,H,-1,-1,1,0,E))?((N(D,R,C,G,H
        ,-1,-2,0,-1,1))?(((35-T)*(F(D,d,R,C,G,H)))+(F(D,d,R,C,G,H))):(((5)+(15-P))+(4)
        )):((N(D,R,C,G,H,-3,-2,0,0,E))?(-N(D,R,C,G,H,-2,-3,0,0,0)):(((F(D,d,R,C,G,H))
        *((35-T)*(F(D,d,R,C,G,H))))+(((F(D,d,R,C,G,H))+(F(D,d,R,C,G,H)))+(N(D,R,C,G,H
        ,-3,-4,0,-1,1)))))));
52  if (r > best) { best=r; ret = 2*D; }
53  }
54  ++T;
55  D=ret/2; if (N(D,R,C,G,H,ret%2-1,-1,ret%2-1,-1,1)) ++P;
56  return ret;
57  }
```

A.1.4. EvolAnt2

```
1   #define FORYXN for (y=1;y<11;y++) for (x=0;x<11;x++)
2   #define FORYX1(y1,y2,x1,x2) for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++)
3   #define FORYX(y1,y2,x1,x2) { for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++) if
        (G[(11+y)%11][(11+x)%11]==w) c+=H[(11+y)%11][(11+x)%11]; }
4   static float N(int D, int R, int C, int G[11][11], float H[11][11], int x1, int y1
        , int x2, int y2, int w) {
5   float c; register int y,x;
6   c=0;
7   if (D==0) FORYX(y1,y2,x1,x2) else if (D==1) FORYX(x1,x2,-(y2),-(y1)) else if (D
        ==2) FORYX(-(y2),-(y1),-(x2),-(x1)) else FORYX(-(x2),-(x1),y1,y2);
8   return c;
9   }
10
11  #define FORYXV(y1,y2,x1,x2) { for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++) c
        +=G[(11+y)%11][(11+x)%11]; }
12  static float B(int D, int R, int C, int G[11][11], int x1, int y1, int x2, int y2)
         {
13  int c; register int y,x;
14  c=0;
15  if (D==0) FORYXV(y1,y2,x1,x2) else if (D==1) FORYXV(x1,x2,-(y2),-(y1)) else if (D
        ==2) FORYXV(-(y2),-(y1),-(x2),-(x1)) else FORYXV(-(x2),-(x1),y1,y2);
16  return c;
17  }
18
19  #define MAXF(y,x) if (G[(R+11+y)%11][(C+11+x)%11]==1 && H[(R+11+y)%11][(C+11+x)
        %11]>s) s=H[(R+11+y)%11][(C+11+x)%11]
20  #define FOOD(y,x) (G[(R+11+y)%11][(C+11+x)%11]==1?H[(R+11+y)%11][(C+11+x)%11]:0)
21  static float F(int D, int d, int R, int C, int G[11][11], float H[11][11]) {
22  float s; s=0;
```

```
23 if (D==0 && d==0) { MAXF(0,-1);MAXF(-1,-1);MAXF(-2,-1);MAXF(-2,0);MAXF(-2,1);MAXF
       (-1,1);MAXF(0,1); return s+FOOD(-1,0); }
24 if (D==0 && d==1) { MAXF(0,-1);MAXF(0,-2);MAXF(-1,-2);MAXF(-2,-2);MAXF(-2,-1);MAXF
       (-2,0);MAXF(-1,0); return s+FOOD(-1,-1); }
25 if (D==1 && d==0) { MAXF(-1,0);MAXF(-1,1);MAXF(-1,2);MAXF(0,2);MAXF(1,2);MAXF(1,1)
       ;MAXF(1,0); return s+FOOD(0,1); }
26 if (D==1 && d==1) { MAXF(-1,0);MAXF(-2,0);MAXF(-2,1);MAXF(-2,2);MAXF(-1,2);MAXF
       (0,2);MAXF(0,1); return s+FOOD(-1,1); }
27 if (D==2 && d==0) { MAXF(0,1);MAXF(1,1);MAXF(2,1);MAXF(2,0);MAXF(2,-1);MAXF(1,-1);
       MAXF(0,-1); return s+FOOD(1,0); }
28 if (D==2 && d==1) { MAXF(0,1);MAXF(0,2);MAXF(1,2);MAXF(2,2);MAXF(2,1);MAXF(2,0);
       MAXF(1,0); return s+FOOD(1,1); }
29 if (D==3 && d==0) { MAXF(1,0);MAXF(1,-1);MAXF(1,-2);MAXF(0,-2);MAXF(-1,-2);MAXF
       (-1,-1);MAXF(-1,0); return s+FOOD(0,-1); }
30 { MAXF(1,0);MAXF(2,0);MAXF(2,-1);MAXF(2,-2);MAXF(1,-2);MAXF(0,-2);MAXF(0,-1);
       return s+FOOD(1,-1); }
31 }
32
33 int ant_Move(int **grid, int my_row, int my_column) {
34 static int T,P,G[11][11],V[11][11];
35 static float H[11][11];
36 register int x,y; int D,E,ret,R,C,d;
37 float r,best;
38 R=my_row;C=my_column;
39 if (C == -1) { return 0; }
40 if (R == -1) { T=P=0; FORYXN { V[y][x]=G[y][x]=0;H[y][x]=0;} return -1; }
41 D = 0;
42 E = (grid[R][C]==10)?100:10;
43 V[R][C]=1; FORYXN if (G[y][x]==1) H[y][x]*=0.9; else if (G[y][x]==E) G[y][x]=0;
       FORYX1(-2,2,-2,2) {G[(11+y)%11][(11+x)%11] = grid[(11+y)%11][(11+x)%11]; H
       [(11+y)%11][(11+x)%11]=1;}
44 best=0;
45 ret=-1;
46 for (;D<4;++D) {
47 d=0;
48 r=((!(N(D,R,C,G,H,-1,-6,2,-2,E)))?(((!(N(D,R,C,G,H,0,-2,1,0,E)))||(!(N(D,R,C,G,H
       ,-1,-1,0,0,E))))?((N(D,R,C,G,H,-4,-3,0,0,E))?(N(D,R,C,G,H,-2,-2,0,-1,1)):((N(D
       ,R,C,G,H,0,-1,1,1,1))*(N(D,R,C,G,H,0,-1,1,0,1)))):(((35-T)*(15-P))+((N(D,R,C,G
       ,H,-1,-6,2,-2,E))?(N(D,R,C,G,H,-3,-1,1,0,1)):(N(D,R,C,G,H,-1,0,0,1,1)))))
       :(((((F(D,d,R,C,G,H))-(35-T))+(N(D,R,C,G,H,0,-1,1,0,1)))<((N(D,R,C,G,H
       ,-2,-1,0,0,E))?((35-T)*(15-P)):(N(D,R,C,G,H,-2,-2,0,-1,1))))?((!(N(D,R,C,G,H
       ,-4,-3,0,0,E)))?((N(D,R,C,G,H,-1,-1,0,0,1))*(F(D,d,R,C,G,H))):((N(D,R,C,G,H
       ,-1,-1,1,1,1))*(F(D,d,R,C,G,H)))):((N(D,R,C,G,H,0,-1,1,0,1))*(15-P))))+((((2)
       <(N(D,R,C,G,H,-1,-1,0,0,1)))?(((F(D,d,R,C,G,H))-(35-T))+((N(D,R,C,G,H
       ,0,-1,1,1,1))*(15-P))):(((N(D,R,C,G,H,-1,-2,1,-1,E))?(P):(N(D,R,C,G,H
       ,0,-2,1,-1,1)))*(F(D,d,R,C,G,H))))+((F(D,d,R,C,G,H))*((N(D,R,C,G,H,-1,-2,1,-1,
       E))?(((F(D,d,R,C,G,H))-(35-T))+(N(D,R,C,G,H,-1,-1,0,0,1))):((N(D,R,C,G,H
       ,-1,-1,0,0,1))*(F(D,d,R,C,G,H))))));
49 if (r > best || ret == -1) { best=r; ret = 2*D+1; }
50 d=1;
```

```
51 r=((N(D,R,C,G,H,-1,-1,0,0,E))?((((15-P)+(-N(D,R,C,G,H,0,1,2,4,0)))-(15-P))*((N(D,R
       ,C,G,H,-2,-2,0,-1,E))?((N(D,R,C,G,H,-1,-1,0,1,1))*(-N(D,R,C,G,H,0,1,2,4,0)))
       :((15-P)+(N(D,R,C,G,H,2,-2,3,1,1)))))):((!(N(D,R,C,G,H,-4,-3,0,0,E)))?(N(D,R,C,
       G,H,-3,-3,-1,-1,1)):(((N(D,R,C,G,H,0,-1,1,0,1))+(N(D,R,C,G,H,-1,-1,0,0,1)))+((
       F(D,d,R,C,G,H))-(35-T)))))+((((35-T)+((N(D,R,C,G,H,-2,0,-1,5,1))+((F(D,d,R,C,G
       ,H))-(35-T))))==(35-T))?(((N(D,R,C,G,H,-1,-1,0,0,1))<((N(D,R,C,G,H
       ,-1,-1,0,0,1))*(N(D,R,C,G,H,-2,-1,0,0,1))))?(((-N(D,R,C,G,H,0,1,2,4,0))-(35-T)
       )+(N(D,R,C,G,H,-1,-1,0,0,1))):((!(N(D,R,C,G,H,-1,-2,1,-1,E)))?((F(D,d,R,C,G,H)
       )+(N(D,R,C,G,H,-2,-3,-1,-1,1))):((N(D,R,C,G,H,-1,-1,0,0,1))*(N(D,R,C,G,H
       ,-1,-1,0,0,1))))):(((N(D,R,C,G,H,-1,-1,0,0,1))<(N(D,R,C,G,H,0,-2,1,-1,1)))
       ?(((15-P)+((F(D,d,R,C,G,H))*(F(D,d,R,C,G,H))))-(15-P)):((0.008753981)+((N(D,R,
       C,G,H,-1,-1,1,0,1))*(F(D,d,R,C,G,H))))));
52 if (r > best) { best=r; ret = 2*D; }
53 }
54 ++T;
55 D=ret/2; if (N(D,R,C,G,H,ret%2-1,-1,ret%2-1,-1,1)) ++P;
56 return ret;
57 }
```

A.1.5. EvolAnt3

```
1  #define FORYXN for (y=0;y<11;y++) for (x=0;x<11;x++)
2  #define FORYX1(y1,y2,x1,x2) for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++)
3  #define FORYX(y1,y2,x1,x2) { for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++) if
       (G[(11+y)%11][(11+x)%11]==w) c+=H[(11+y)%11][(11+x)%11]; }
4  static float N(int D, int R, int C, int G[11][11], float H[11][11], int x1, int y1
       , int x2, int y2, int w) {
5  float c; register int y,x;
6  c=0;
7  if (D==0) FORYX(y1,y2,x1,x2) else if (D==1) FORYX(x1,x2,-(y2),-(y1)) else if (D
       ==2) FORYX(-(y2),-(y1),-(x2),-(x1)) else FORYX(-(x2),-(x1),y1,y2);
8  return c;
9  }
10
11 #define FORYXV(y1,y2,x1,x2) { for (y=y1+R;y<=y2+R;y++) for (x=x1+C;x<=x2+C;x++) c
       +=G[(11+y)%11][(11+x)%11]; }
12 static float B(int D, int R, int C, int G[11][11], int x1, int y1, int x2, int y2)
        {
13 int c; register int y,x;
14 c=0;
15 if (D==0) FORYXV(y1,y2,x1,x2) else if (D==1) FORYXV(x1,x2,-(y2),-(y1)) else if (D
       ==2) FORYXV(-(y2),-(y1),-(x2),-(x1)) else FORYXV(-(x2),-(x1),y1,y2);
16 return c;
17 }
18
19 #define MAXF(y,x) if (G[(R+11+y)%11][(C+11+x)%11]==1 && H[(R+11+y)%11][(C+11+x)
       %11]>s) s=H[(R+11+y)%11][(C+11+x)%11]
20 #define FOOD(y,x) (G[(R+11+y)%11][(C+11+x)%11]==1?H[(R+11+y)%11][(C+11+x)%11]:0)
21 static float F(int D, int d, int R, int C, int G[11][11], float H[11][11]) {
22 float s; s=0;
```

```
23 if (D==0 && d==0) { MAXF(0,-1);MAXF(-1,-1);MAXF(-2,-1);MAXF(-2,0);MAXF(-2,1);MAXF
       (-1,1);MAXF(0,1); return s+FOOD(-1,0); }
24 if (D==0 && d==1) { MAXF(0,-1);MAXF(0,-2);MAXF(-1,-2);MAXF(-2,-2);MAXF(-2,-1);MAXF
       (-2,0);MAXF(-1,0); return s+FOOD(-1,-1); }
25 if (D==1 && d==0) { MAXF(-1,0);MAXF(-1,1);MAXF(-1,2);MAXF(0,2);MAXF(1,2);MAXF(1,1)
       ;MAXF(1,0); return s+FOOD(0,1); }
26 if (D==1 && d==1) { MAXF(-1,0);MAXF(-2,0);MAXF(-2,1);MAXF(-2,2);MAXF(-1,2);MAXF
       (0,2);MAXF(0,1); return s+FOOD(-1,1); }
27 if (D==2 && d==0) { MAXF(0,1);MAXF(1,1);MAXF(2,1);MAXF(2,0);MAXF(2,-1);MAXF(1,-1);
       MAXF(0,-1); return s+FOOD(1,0); }
28 if (D==2 && d==1) { MAXF(0,1);MAXF(0,2);MAXF(1,2);MAXF(2,2);MAXF(2,1);MAXF(2,0);
       MAXF(1,0); return s+FOOD(1,1); }
29 if (D==3 && d==0) { MAXF(1,0);MAXF(1,-1);MAXF(1,-2);MAXF(0,-2);MAXF(-1,-2);MAXF
       (-1,-1);MAXF(-1,0); return s+FOOD(0,-1); }
30 { MAXF(1,0);MAXF(2,0);MAXF(2,-1);MAXF(2,-2);MAXF(1,-2);MAXF(0,-2);MAXF(0,-1);
       return s+FOOD(1,-1); }
31 }
32
33 int ant_Move(int **grid, int my_row, int my_column) {
34 static int T,P,G[11][11],V[11][11];
35 static float H[11][11];
36 register int x,y; int D,E,ret,R,C,d;
37 float r,best;
38 R=my_row;C=my_column;
39 if (C == -1) { return 0; }
40 if (R == -1) { T=P=0; FORYXN { V[y][x]=G[y][x]=0;H[y][x]=0;} return -1; }
41 D = 0;
42 E = (grid[R][C]==10)?100:10;
43 V[R][C]=1; FORYXN if (G[y][x]==1) H[y][x]*=0.9; else if (G[y][x]==E) G[y][x]=0;
       FORYX1(-2,2,-2,2) {G[(11+y)%11][(11+x)%11] = grid[(11+y)%11][(11+x)%11]; H
       [(11+y)%11][(11+x)%11]=1;}
44 best=0;
45 ret=-1;
46 for (;D<4;++D) {
47 d=0;
48 r=(N(D,R,C,G,H,-1,-4,0,-1,1))+((!((!((!((1)==(F(D,d,R,C,G,H))))||(((N(D,R,C,G,H
       ,-4,-1,1,2,1))||(-0.19937818)<(35-T)))||(N(D,R,C,G,H,0,-2,1,-1,1)))))||(N(D,R
       ,C,G,H,-1,-2,1,1,E))))?((((N(D,R,C,G,H,-4,-1,0,0,E))||(N(D,R,C,G,H,-3,-5,2,0,E
       )))||((N(D,R,C,G,H,0,-1,1,0,1))&&(N(D,R,C,G,H,-2,-1,0,1,1))))?((N(D,R,C,G,H
       ,0,-2,3,-1,1))+((F(D,d,R,C,G,H))*(35-T))):(((N(D,R,C,G,H,0,-1,1,0,E))&&(N(D,R,
       C,G,H,0,-1,1,0,1)))?(35-T):(N(D,R,C,G,H,-3,-3,-1,-1,1)))):((((N(D,R,C,G,H
       ,0,2,1,3,E))&&(N(D,R,C,G,H,-4,-1,0,0,E)))?(F(D,d,R,C,G,H)):(P))+(((N(D,R,C,G,H
       ,0,-1,1,0,E))&&(N(D,R,C,G,H,-4,-1,0,0,E)))?(35-T):(-0.7490231))));
49 if (r > best || ret == -1) { best=r; ret = 2*D+1; }
50 d=1;
51 r=(((((N(D,R,C,G,H,-4,-1,0,0,E))&&(N(D,R,C,G,H,-4,-1,0,0,E)))&&((N(D,R,C,G,H
       ,-1,-3,1,-1,E))&&(N(D,R,C,G,H,-3,-3,0,0,E))))?((F(D,d,R,C,G,H))*(35-T)):((N(D,
       R,C,G,H,-3,-3,0,0,E))?(-N(D,R,C,G,H,-2,-4,3,1,0)):(F(D,d,R,C,G,H))))+((35-T)
       +((F(D,d,R,C,G,H))+(N(D,R,C,G,H,-1,-3,1,-1,1)))))+((-N(D,R,C,G,H,-1,-1,0,0,0))
       +((((N(D,R,C,G,H,-4,-1,0,0,E))&&(N(D,R,C,G,H,-3,-3,0,0,E)))&&((N(D,R,C,G,H
       ,-4,-1,0,2,E))&&(N(D,R,C,G,H,-2,-3,-1,-1,E))))?(((N(D,R,C,G,H,-3,-2,2,-1,E))
```

```
    &&(N(D,R,C,G,H,1,-4,4,-2,E)))?(F(D,d,R,C,G,H)):(P)):(((N(D,R,C,G,H,-3,-5,2,0,E
    ))&&(N(D,R,C,G,H,-3,-5,2,0,E)))?(F(D,d,R,C,G,H)):(N(D,R,C,G,H,-3,-3,-1,-1,1)))
    ));
52  if (r > best) { best=r; ret = 2*D; }
53  }
54  ++T;
55  D=ret/2; if (N(D,R,C,G,H,ret%2-1,-1,ret%2-1,-1,1)) ++P;
56  return ret;
57  }
```

A.2. Designed Ants

A.2.1. Utils

```
1   const int DIRS[8][2]
2     = {{-1,-1},{-1,0},{-1,+1},{0,+1},{+1,+1},{+1,0},{+1,-1},{0,-1}};
3
4   int randint(int a, int b)
5   {
6       return rand()%(b-a+1)+a;
7   }
8   void swap(int* a, int* b)
9   {
10      int t = *a;
11      *a = *b;
12      *b = t;
13  }
14  void randarr(int* arr, int n)
15  {
16      int i;
17      for (i=0; i<n; ++i)
18          swap(&arr[i], &arr[randint(i,n-1)]);
19  }
```

A.2.2. HyperHumant

```
1   #include "utils.h"
2
3   int hiper_points;
4   int hiper_time;
5   int hiper_my_row;
6   int hiper_my_col;
7   int hiper_grid[GRID_SIZE][GRID_SIZE];
8   int hiper_seen[GRID_SIZE][GRID_SIZE];
9
10  static int randbestinertia(int dir, int* evals) {
11      int i,mx=evals[0]-1;
12      int diff = 10;
13      int best[8],nbests=0;
14      for (i=0; i<8; ++i) {
15          int ndiff = abs((dir-i+12)%8-4);
```

```
16          if (evals[i]>mx || (evals[i]==mx && ndiff < diff)) {
17              diff = ndiff;
18              mx=evals[i];
19              best[0]=i;
20              nbests=1;
21          } else if (evals[i]==mx && ndiff==diff)
22              best[nbests++]=i;
23      }
24      return best[rand()%nbests];
25  }
26  static int is_in_vicinity(int **grid, int row, int col, int val)
27  {
28      int d;
29      for (d=0; d<8; ++d) {
30          int nrow = row + DIRS[d][0];
31          int ncol = col + DIRS[d][1];
32          if (CELL(grid,nrow,ncol) == val)
33              return 1;
34      }
35      return 0;
36  }
37  static int DIST(int x1, int x2)
38  {
39      int x = MOD(x1-x2,GRID_SIZE);
40      return MIN(x, GRID_SIZE - x);
41  }
42  static int dist(int x1, int y1, int x2, int y2)
43  {
44      return MAX(DIST(x1,x2),DIST(y1,y2));
45  }
46  static int eval_exploration_dir(int **grid, int row, int col)
47  {
48      int best = 100000;
49      int i,j;
50      for (i=0; i<GRID_SIZE; ++i)
51          for (j=0; j<GRID_SIZE; ++j) if (CELL(hiper_grid,i,j)==CELL_FOOD) {
52              int v;
53              v = dist(row, col, i, j) + 0.1*(hiper_time-CELL(hiper_seen,i,j));
54              if (v < best) {
55                  best = v;
56              }
57          }
58      return -best;
59  }
60  static int foodhope(int **grid, int row, int col, int time)
61  {
62      int ptsx[16];
63      int ptsy[16];
64      int vis[16];
65      int npts = 0;
66      int i,j;
```

```
67      int timeleft = time;
68      int px = row;
69      int py = col;
70      int hope = 0;
71      for (i=-2; i<=2; ++i) for (j=-2; j<=2; ++j) {
72          int nx = hiper_my_row + i;
73          int ny = hiper_my_col + j;
74          if (CELL(grid,nx,ny) == CELL_FOOD) {
75              ptsx[npts] = nx;
76              ptsy[npts] = ny;
77              vis[npts] = 0;
78              npts++;
79          }
80      }
81      while (1) {
82          int best = -1;
83          int bestdist = 10000;
84          for (i=0; i<npts; ++i) if (!vis[i]) {
85              int d = dist(ptsx[i], ptsy[i], px, py);
86              if (d <= timeleft && d < bestdist) {
87                  bestdist = d;
88                  best = i;
89              }
90          }
91          if (best == -1)
92              break;
93          vis[best] = 1;
94          timeleft -= bestdist;
95          hope += timeleft*timeleft;
96          px = ptsx[best];
97          py = ptsy[best];
98      }
99      return hope;
100 }
101 static int evalfield(int **grid, int me, int row, int col)
102 {
103     int eval = 0;
104     int d;
105     /* Obvious end game rules */
106     if (CELL(grid,row,col) == CELL_FOOD) {
107         if (hiper_points == 7)
108             return 100000002;
109         if (hiper_time == 34)
110             return 100000001;
111         if (hiper_points == 6 && hiper_time >= 33)
112             return 100000000;
113     }
114     if (CELL(grid,row,col) == CELL_ENEMY(me))
115         return 10000000;
116     if (is_in_vicinity(grid, row, col, CELL_ENEMY(me)))
117         return -1000000000;
```

```
118     eval = foodhope(grid, row, col, 3)*1000;
119     if (eval == 0) {
120         eval = eval_exploration_dir(grid, row, col);
121     }
122     return eval;
123 }
124 static void evalfields(int **grid, int row, int col, int* evals)
125 {
126     int me = CELL(grid, row, col);
127     int d;
128     for (d=0; d<8; ++d) {
129         int nrow = row + DIRS[d][0];
130         int ncol = col + DIRS[d][1];
131         evals[d] = evalfield(grid, me, nrow, ncol);
132     }
133 }
134 int ant_Move(int **grid, int my_row, int my_column)
135 {
136     static int dir;
137     int i,j;
138     int evals[8];
139     if (my_row==-1) {
140         hiper_points = 0;
141         hiper_time = 0;
142         for (i=0; i<GRID_SIZE; ++i)
143             for (j=0; j<GRID_SIZE; ++j) {
144                 hiper_grid[i][j] = 0;
145                 hiper_seen[i][j] = -1000;
146             }
147         dir=0;
148         return -1;
149     }
150     hiper_my_row = my_row;
151     hiper_my_col = my_column;
152     for (i=-2; i<=2; ++i)
153         for (j=-2; j<=2; ++j) {
154             int nr = my_row + i;
155             int nc = my_column + j;
156             CELL(hiper_seen, nr, nc) = hiper_time;
157             CELL(hiper_grid, nr, nc) = (CELL(grid, nr, nc) == CELL_FOOD ?
                    CELL_FOOD : CELL_EMPTY);
158         }
159     evalfields(grid, my_row, my_column, evals);
160     dir = randbestinertia(dir, evals);
161     if (CELL(grid, my_row + DIRS[dir][0], my_column + DIRS[dir][1]) == CELL_FOOD)
162         hiper_points++;
163     hiper_time++;
164     return dir;
165 }
```

A.2.3. SuperHumant

```
#include "utils.h"

int super_points;
int super_time;
int super_grid[GRID_SIZE][GRID_SIZE];
int super_seen[GRID_SIZE][GRID_SIZE];

static int randbestinertia(int dir, int* evals) {
    int i,mx=evals[0]-1;
    int diff = 10;
    int best[8],nbests=0;
    for (i=0; i<8; ++i) {
        int ndiff = abs((dir-i+12)%8-4);
        if (evals[i]>mx || (evals[i]==mx && ndiff < diff)) {
            diff = ndiff;
            mx=evals[i];
            best[0]=i;
            nbests=1;
        } else if (evals[i]==mx && ndiff==diff)
            best[nbests++]=i;
    }
    return best[rand()%nbests];
}
static int is_in_vicinity(int **grid, int row, int col, int val)
{
    int d;
    for (d=0; d<8; ++d) {
        int nrow = row + DIRS[d][0];
        int ncol = col + DIRS[d][1];
        if (CELL(grid,nrow,ncol) == val)
            return 1;
    }
    return 0;
}
static int DIST(int x1, int x2)
{
    int x = MOD(x1-x2,GRID_SIZE);
    return MIN(x, GRID_SIZE - x);
}
static int dist(int x1, int y1, int x2, int y2)
{
    return MAX(DIST(x1,x2),DIST(y1,y2));
}
static int eval_exploration_dir(int **grid, int row, int col)
{
    int best = 100000;
    int i,j;
    for (i=0; i<GRID_SIZE; ++i)
        for (j=0; j<GRID_SIZE; ++j) if (CELL(super_grid,i,j)==CELL_FOOD) {
            int v;
            v = dist(row, col, i, j) + 0.1*(super_time-CELL(super_seen,i,j));
```

```
52              if (v < best) {
53                  best = v;
54              }
55          }
56      return -best;
57  }
58  static int evalfield(int **grid, int me, int row, int col)
59  {
60      int eval = 0;
61      int d;
62      /* Obvious end game rules */
63      if (CELL(grid,row,col) == CELL_FOOD) {
64          if (super_points == 7)
65              return 10000;
66          if (super_time == 34)
67              return 10000;
68          if (super_points == 6 && super_time >= 33)
69              return 1000;
70      }
71      if (CELL(grid,row,col) == CELL_ENEMY(me))
72          return 100;
73      if (CELL(grid,row,col) == CELL_FOOD)
74          eval += 50;
75      for (d=0; d<8; ++d) {
76          int nrow = row + DIRS[d][0];
77          int ncol = col + DIRS[d][1];
78          if (CELL(grid,nrow,ncol) == CELL_ENEMY(me))
79              return -100;
80          if (CELL(grid,nrow,ncol) == CELL_FOOD)
81              eval += 8;
82      }
83      if (eval == 0) {
84          eval = eval_exploration_dir(grid, row, col);
85      }
86      return eval;
87  }
88  static void evalfields(int **grid, int row, int col, int* evals)
89  {
90      int me = CELL(grid, row, col);
91      int d;
92      for (d=0; d<8; ++d) {
93          int nrow = row + DIRS[d][0];
94          int ncol = col + DIRS[d][1];
95          evals[d] = evalfield(grid, me, nrow, ncol);
96      }
97  }
98  int ant_Move(int **grid, int my_row, int my_column)
99  {
100     static int dir;
101     int i,j;
102     int evals[8];
```

```
103     if (my_row==-1) {
104         super_points = 0;
105         super_time = 0;
106         for (i=0; i<GRID_SIZE; ++i)
107             for (j=0; j<GRID_SIZE; ++j) {
108                 super_grid[i][j] = 0;
109                 super_seen[i][j] = -1000;
110             }
111         dir=0;
112         return -1;
113     }
114     for (i=-2; i<=2; ++i)
115         for (j=-2; j<=2; ++j) {
116             int nr = my_row + i;
117             int nc = my_column + j;
118             CELL(super_seen, nr, nc) = super_time;
119             CELL(super_grid, nr, nc) = (CELL(grid, nr, nc) == CELL_FOOD ?
                    CELL_FOOD : CELL_EMPTY);
120         }
121     evalfields(grid, my_row, my_column, evals);
122     dir = randbestinertia(dir, evals);
123     if (CELL(grid, my_row + DIRS[dir][0], my_column + DIRS[dir][1]) == CELL_FOOD)
124         super_points++;
125     super_time++;
126     return dir;
127 }
```

A.2.4. SmartHumant

```
1  #include "utils.h"
2
3  static int randbest(int* evals) {
4      int i,mx=evals[0]-1;
5      int best[8],nbests=0;
6      for (i=0; i<8; ++i) {
7          if (evals[i]>mx) {
8              mx=evals[i];
9              best[0]=i;
10             nbests=1;
11         } else if (evals[i]==mx)
12             best[nbests++]=i;
13     }
14     return best[rand()%nbests];
15 }
16 static int evalfield(int **grid, int me, int row, int col)
17 {
18     int eval = 0;
19     int d;
20     if (CELL(grid,row,col) == CELL_ENEMY(me))
21         return 100;
22     if (CELL(grid,row,col) == CELL_FOOD)
```

```
23          eval += 6.0;
24      for (d=0; d<8; ++d) {
25          int nrow = row + DIRS[d][0];
26          int ncol = col + DIRS[d][1];
27          if (CELL(grid,nrow,ncol) == CELL_ENEMY(me))
28              return -100;
29          if (CELL(grid,nrow,ncol))
30              eval += 1;
31      }
32      return eval;
33  }
34  static void evalfields(int **grid, int row, int col, int* evals)
35  {
36      int me = CELL(grid, row, col);
37      int d;
38      for (d=0; d<8; ++d) {
39          int nrow = row + DIRS[d][0];
40          int ncol = col + DIRS[d][1];
41          evals[d] = evalfield(grid, me, nrow, ncol);
42      }
43  }
44  int ant_Move(int **grid, int my_row, int my_column)
45  {
46      int evals[8];
47      if (my_row <= -1)
48          return -1;
49      evalfields(grid, my_row, my_column, evals);
50      return randbest(evals);
51  }
```

Bibliography

[1] Noga Alon, Dana Moshkovitz, and Shmuel Safra. Algorithmic construction of sets for k-restrictions. *ACM Transactions on Algorithms (TALG)*, 2(2):177, 2006.

[2] Helmut Alt, Norbert Blum, Kurt Mehlhorn, and Markus Paul. Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}m/\log(n))$. *Information Processing Letters*, 37(4):237–240, 1991.

[3] Peter J. Angeline and Jordan B. Pollack. Competitive environments evolve better solutions for complex tasks. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 264–270, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[4] Robert Axelrod. The evolution of strategies in the iterated prisoner's dilemma. In L. Davis, editor, *Genetic Algorithms in Simulated Annealing*, pages 32–41. Pitman, London, 1987.

[5] Yaniv Azaria and Moshe Sipper. GP-gammon: Genetically programming backgammon players. *Genetic Programming and Evolvable Machines*, 6(3):283–300, 2005.

[6] Yaniv Azaria and Moshe Sipper. GP-Gammon: Using genetic programming to evolve backgammon players. In M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 132–142, Lausanne, Switzerland, 2005. Springer.

[7] Thomas B
"ack, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation*. Oxford University Press, 1997.

[8] Anurag Bhatt, Pratul Varshney, and Kalyanmoy Deb. In search of no-loss strategies for the game of tic-tac-toe using a customized genetic algorithm. In Maarten Keijzer, Giuliano Antoniol, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Nikolaus Hansen, John H. Holmes, Gregory S. Hornby, Daniel Howard, James

Kennedy, Sanjeev Kumar, Fernando G. Lobo, Julian Francis Miller, Jason Moore, Frank Neumann, Martin Pelikan, Jordan Pollack, Kumara Sastry, Kenneth Stanley, Adrian Stoica, El-Ghazali Talbi, and Ingo Wegener, editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 889–896, Atlanta, GA, USA, 12-16 July 2008. ACM.

[9] Alan D. Blair and Jordan B. Pollack. What makes a good co-evolutionary learning environment. *Australian Journal of Intelligent Information Processing Systems*, 4(3/4):166–175, 1997.

[10] Josh C. Bongard and Hod Lipson. 'managed challenge' alleviates disengagement in co-evolutionary system identification. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 531–538, New York, NY, USA, 2005. ACM.

[11] Bruno Bouzy and Tristan Cazenave. Computer go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[12] Dimo Brockhoff, Tobias Friedrich, Nils Hebbinghaus, Christian Klein, Frank Neumann, and Eckart Zitzler. Do additional objectives make a problem harder? In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 765–772. ACM, 2007.

[13] Dimo Brockhoff and Eckart Zitzler. Objective reduction in evolutionary multiobjective optimization: Theory and applications. *Evolutionary Computation*, 17(2):135–166, 2009.

[14] Bernd Brügmann. Monte Carlo Go. Unpublished technical report, 1993.

[15] Anthony Bucci. *Emergent Geometric Organization and Informative Dimensions in Coevolutionary Algorithms*. PhD thesis, Michtom School of Computer Science, Brandeis University, 2007.

[16] Anthony Bucci and Jordan B. Pollack. Order-theoretic analysis of coevolution problems: Coevolutionary statics. In *Proceedings of the GECCO-2002 Workshop on Coevolution: Understanding Coevolution*, pages 229–235, 2002.

[17] Anthony Bucci, Jordan B. Pollack, and Edwin de Jong. Automated extraction of problem structure. In Kalyanmoy Deb et al., editor, *Genetic and Evolutionary Computation – GECCO-2004, Part I*, volume 3102 of *Lecture Notes in Computer Science*, pages 501–512, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[18] John Peter Cartlidge. *Rules of Engagement: Competitive coevolutionary dynamics in computational systems.* PhD thesis, University of Leeds, 2004.

[19] James B. Caverlee. A genetic algorithm approach to discovering an optimal blackjack strategy. In John R. Koza, editor, *Genetic Algorithms and Genetic Programming at Stanford 2000*, pages 70–79. Stanford Bookstore, Stanford, California, 94305-3079 USA, June 2000.

[20] Kumar Chellapilla and David B. Fogel. Evolving neural networks to play checkers without relying on expert knowledge. *Neural Networks, IEEE Transactions on*, 10(6):1382–1391, 1999.

[21] Kumar Chellapilla and David B. Fogel. Evolving an expert checkers playing program without using human expertise. *Evolutionary Computation, IEEE Transactions on*, 5(4):422–428, 2001.

[22] Siang Y. Chong, Mei K. Tan, and Jonathon D. White. Observing the evolution of neural networks learning to play the game of othello. *Evolutionary Computation, IEEE Transactions on*, 9(3):240 – 251, 2005.

[23] Fulvio Corno, Ernesto Sanchez, and Giovanni Squillero. On the evolution of corewar warriors. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 133–138, Portland, Oregon, 20-23 June 2004. IEEE Press.

[24] Robert De Caux. Using genetic programming to evolve strategies for the iterated prisoner's dilemma. Master's thesis, University College, London, September 2001.

[25] Edvin D. de Jong and Jordan B. Pollack. Learning the ideal evaluation function. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-03*, pages 274–285, Berlin, 2003, 2003. Springer.

[26] Edwin D. de Jong. The Incremental Pareto-Coevolution Archive. In Kalyanmoy Deb et al., editor, *Genetic and Evolutionary Computation–GECCO 2004. Proceedings of the Genetic and Evolutionary Computation Conference. Part I*, pages 525–536, Seattle, Washington, USA, June 2004. Springer-Verlag, Lecture Notes in Computer Science Vol. 3102.

[27] Edwin D. de Jong. Intransitivity in Coevolution. In Xin Yao et al., editor, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *Lecture Notes in Computer Science (LNCS)*, pages 843–851, Birmingham, UK, September 2004. Springer-Verlag (New York).

[28] Edwin D. de Jong. Towards a Bounded Pareto-Coevolution Archive. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, volume 2, pages 2341–2348, Portland, Oregon, USA, June 2004. IEEE Service Center.

[29] Edwin D. de Jong. The maxsolve algorithm for coevolution. In Hans-Georg Beyer et al., editor, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 1, pages 483–489, Washington DC, USA, 25-29 June 2005. ACM Press.

[30] Edwin D. de Jong. A Monotonic Archive for Pareto-Coevolution. *Evolutionary Computation*, 15(1):61–93, Spring 2007.

[31] Edwin D. de Jong. Objective fitness correlation. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 440–447, New York, NY, USA, 2007. ACM Press.

[32] Edwin D. de Jong and Anthony Bucci. DECA: dimension extracting coevolutionary algorithm. In Mike Cattolico et al., editor, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 313–320, Seattle, Washington, USA, 2006. ACM Press.

[33] Edwin D. de Jong and Anthony Bucci. Objective Set Compression. Test-Based Problems and Multiobjective Optimization. In Joshua Knowles et al., editor, *Multiobjective Problem Solving from Nature: From Concepts to Applications*, pages 357–376. Springer, Berlin, 2008.

[34] Edwin D. de Jong and Jordan B. Pollack. Ideal Evaluation from Coevolution. *Evolutionary Computation*, 12(2):159–192, Summer 2004.

[35] Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 32–43, New York, NY, USA, 2003. ACM.

[36] Robert P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51:161–166, 1950.

[37] Ben Dushnik and Edwin W. Miller. Partially ordered sets. *American Journal of Mathematics*, 63(3):600–610, 1941.

[38] Andries P. Engelbrecht. *Computational intelligence: an introduction.* Wiley, 2007.

[39] Stefan Felsner, Vijay Raghavan, and Jeremy Spinrad. Recognition algorithms for orders of small width and graphs of small Dilworth number. *Order*, 20(4):351–364, 2003.

[40] Sevan G. Ficici. *Solution concepts in coevolutionary algorithms.* PhD thesis, Waltham, MA, USA, 2004. Adviser-Pollack, Jordan B.

[41] Sevan G. Ficici. Multiobjective Optimization and Coevolution. In Joshua Knowles, David Corne, and Kalyanmoy Deb, editors, *Multi-Objective Problem Solving from Nature: From Concepts to Applications*, pages 31–52. Springer, Berlin, 2008. ISBN 978-3-540-72963-1.

[42] Sevan G. Ficici and Jordan B. Pollack. Challenges in coevolutionary learning: Arms-race dynamics, open-endedness, and mediocre stable states. In *Proceedings of the Sixth International Conference on Artificial Life*, pages 238–247. MIT Press, 1998.

[43] Sevan G. Ficici and Jordan B. Pollack. Pareto optimality in coevolutionary learning. In Jozef Kelemen and Petr Sosík, editors, *Advances in Artificial Life, 6th European Conference, ECAL 2001*, volume 2159 of *Lecture Notes in Computer Science*, pages 316–325, Prague, Czech Republic, 2001. Springer.

[44] Sevan G. Ficici and Jordan B. Pollack. A game-theoretic memory mechanism for coevolution. In E. Cantú-Paz et al., editor, *Genetic and Evolutionary Computation - GECCO 2003*, volume 2723 of *Lecture Notes in Computer Science*, pages 286–297, Chicago, IL, 2003. Springer.

[45] David B. Fogel. *Blondie24: Playing at the Edge of AI.* Morgan Kaufmann Publishers, September 2001.

[46] David B. Fogel, Timothy J. Hays, Sarah L. Hahn, and James Quon. Further evolution of a self-learning chess program. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games, IEEE, Piscataway, NJ.* Citeseer, 2005.

[47] David B. Fogel, Timothy J. Hays, Sarah L. Hahn, and James Quon. The Blondie25 chess program competes against Fritz 8.0 and a human chess master. In *Computational Intelligence and Games, 2006 IEEE Symposium on*, pages 230–235. IEEE, 2006.

[48] P. Gach, GL Kurdyumov, and LA Levin. One-dimensional uniform arrays that wash out finite islands. *Problemy Peredachi Informatsii*, 14(3):92–96, 1978.

[49] Robert Gibbons. *A primer in game theory*. FT Prentice Hall, 1992.

[50] Chi-Keong Goh and Kay Chen Tan. A competitive-cooperative coevolutionary paradigm for dynamic multiobjective optimization. *Evolutionary Computation, IEEE Transactions on*, 13(1):103–127, 2009.

[51] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-wesley, 1989.

[52] Jinwei Gu, Manzhan Gu, Cuiwen Cao, and Xingsheng Gu. A novel competitive co-evolutionary quantum genetic algorithm for stochastic job shop scheduling problem. *Computers and Operations Research*, 37(5):927–937, 2010.

[53] Ami Hauptman and Moshe Sipper. Evolution of an efficient search algorithm for the mate-in-N problem in chess. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 78–89, Valencia, Spain, 11 - 13 April 2007. Springer.

[54] W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial life II*, volume 10 of *Sante Fe Institute Studies in the Sciences of Complexity*, pages 313–324, Redwood City, Calif., 1992. Addison-Wesley.

[55] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Computing*, 2:225, 1973.

[56] Philip Husbands and Frank Mill. Simulated co-evolution as the mechanism for emergent planning and scheduling. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 264–270. Morgan Kaufmann Publishers, 1991.

[57] Hisao Ishibuchi, Noritaka Tsukamoto, and Yusuke Nojima. Evolutionary many-objective optimization: A short review. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 2419–2426. IEEE, 2008.

[58] Wojciech Jaśkowski and Wojciech Kotlowski. On selecting the best individual in noisy environments. In Maarten Keijzer, Giuliano Antoniol, Clare Bates Congdon,

Kalyanmoy Deb, Benjamin Doerr, Nikolaus Hansen, John H. Holmes, Gregory S. Hornby, Daniel Howard, James Kennedy, Sanjeev Kumar, Fernando G. Lobo, Julian Francis Miller, Jason Moore, Frank Neumann, Martin Pelikan, Jordan Pollack, Kumara Sastry, Kenneth Stanley, Adrian Stoica, El-Ghazali Talbi, and Ingo Wegener, editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 961–968, Atlanta, GA, USA, jul 2008. ACM Press.

[59] Wojciech Jaśkowski and Krzysztof Krawiec. Formal analysis and algorithms for extracting coordinate systems of games. In *IEEE Symposium on Computational Intelligence and Games*, pages 201–208, Milano, Italy, 2009.

[60] Wojciech Jaśkowski and Krzysztof Krawiec. Coordinate system archive for coevolution. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–10, Barcelona, 2010. IEEE.

[61] Wojciech Jaśkowski and Krzysztof Krawiec. How many dimensions in cooptimization? In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. Association of Computing Machinery (ACM), 2011 in press.

[62] Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch. Antwars applet, 2007. (http://www.cs.put.poznan.pl/kkrawiec/antwars/).

[63] Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch. Evolving strategy for a probabilistic game of imperfect information using genetic programming. *Genetic Programming and Evolvable Machiness*, 9(4):281–294, 2008.

[64] Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch. Winning ant wars: Evolving a human-competitive game strategy using fitnessless selection. In M. O'Neill et al., editor, *Genetic Programming 11th European Conference, EuroGP 2008, Proceedings*, volume 4971 of *Lecture Notes in Computer Science*, pages 13–24. Springer-Verlag, mar 2008.

[65] Wojciech Jaśkowski, Bartosz Wieloch, and Krzysztof Krawiec. Fitnessless coevolution. In Maarten Keijzer et al., editor, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 355–362, Atlanta, GA, USA, jul 2008. ACM.

[66] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[67] Hugues Juillé and Jordan B. Pollack. Co-evolving intertwined spirals. In *Proceedings of the Fifth Annual Conference on Evolutionary Programming*, pages 461–468, 1996.

[68] Hugues Juillé and Jordan B. Pollack. Dynamics of co-evolutionary learning. In Pattie Maes, Maja J. Mataric, Jean-Arcady Meyer, Jordan Pollack, and Stewart W. Wilson, editors, *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior: From animals to animats 4*, pages 526–534, Cape Code, USA, 9-13 1996. MIT Press.

[69] Hugues Juillé and Jordan B. Pollack. Coevolutionary learning: a case study. In *In Proceedings of the 15th International Conference on Machine Learning*, pages 251–259. Morgan Kaufmann, 1998.

[70] Hugues Juillé and Jordan B. Pollack. Coevolving the "ideal" trainer: Application to the discovery of cellular automata rules. In *University of Wisconsin*, pages 519–527. Morgan Kaufmann, 1998.

[71] Richard M. Karp. Reducibility Among Combinatorial Problems. *Complexity of computer computations: proceedings*, page 85, 1972.

[72] Jae Y. Kim, Yeo K. Kim, and Yeongho Kim. Tournament competition and its merits for coevolutionary algorithms. *Journal of Heuristics*, 9(3):249–268, 2003.

[73] Yeo K. Kim, Jae Y. Kim, and Yeongho Kim. A tournament-based competitive coevolutionary algorithm. *Applied Intelligence*, 20(3):267–281, 2004.

[74] Joshua D. Knowles, Richard A. Watson, and David Corne. Reducing local optima in single-objective problems by multi-objectivization. In *EMO '01: Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization*, pages 269–283, London, UK, 2001. Springer-Verlag.

[75] John R. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical report, Computer Science Department, Stanford University, 1990.

[76] John R. Koza. Genetic evolution and co-evolution of game strategies. In *The International Conference on Game Theory and Its Applications*, Stony Brook, New York, July 15 1992.

[77] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[78] John R. Koza. *Genetic programming III: darwinian invention and problem solving*. Morgan Kaufmann Pub, 1999.

[79] John R. Koza. *Genetic programming IV: Routine human-competitive machine intelligence*. Kluwer Academic Pub, 2003.

[80] Krzysztof Krawiec, Wojciech Jaśkowski, and Marcin Szubert. Evolving small-board go players using coevolutionary temporal difference learning with archive. *International Journal of Applied Mathematics and Computer Science*, 2011 in press.

[81] Mark Land and Richard K. Belew. No perfect two-state cellular automata for density classification exists. *Phys. Rev. Lett.*, 74(25):5148–5150, Jun 1995.

[82] Christopher G. Langton. *Artificial life: An overview*. The MIT Press, 1997.

[83] Marco Laumanns, Lothal Thiele, Kalyanmoy Deb, and Eckart Zitzler. Combining convergence and diversity in evolutionary multiobjective optimization. *Evolutionary computation*, 10(3):263–282, 2002.

[84] Alex Lubberts and Risto Miikkulainen. Co-evolving a go-playing neural network. In Richard K. Belew and Hugues Juillè, editors, *Coevolution: Turning Adaptive Algorithms upon Themselves*, pages 14–19, San Francisco, California, USA, 7 July 2001.

[85] Simon M. Lucas. Computational intelligence and games: Challenges and opportunities. *International Journal of Automation and Computing*, 5(1):45–57, 2008.

[86] Simon M. Lucas and Thomas P. Runarsson. Temporal difference learning versus co-evolution for acquiring othello position evaluation. In *IEEE Symposium on Computational Intelligence and Games*, pages 52–59. IEEE, 2006.

[87] Sean Luke. Genetic programming produced competitive soccer softbot teams for robocup97. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[88] Sean Luke. ECJ 15: A Java evolutionary computation library. http://cs.gmu.edu/∼eclab/projects/ecj/, 2006.

[89] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Co-evolving soccer softbot team coordination with genetic programming. In Hiroaki Kitano, editor, *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence*, Lecture Notes in Computer Science, pages 398–411, Nagoya, Japan, 1997. Springer.

[90] Sean Luke and R. Paul Wiegand. Guaranteeing coevolutionary objective measures. In Kenneth A. de Jong, Riccardo Poli, and Jonathan E. Rowe, editors, *Foundations of Genetic Algorithms VII*, pages 237–251, Torremolinos, Spain, 2002. Morgan Kaufman.

[91] Sean Luke and R. Paul Wiegand. When coevolutionary algorithms exhibit evolutionary dynamics. In *2002 Genetic and Evolutionary Computation Conference Workshop Program*, pages 236–241, 2002.

[92] Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM (JACM)*, 41(5):960–981, 1994.

[93] Edward P. Manning. Coevolution in a large search space using resource-limited nash memory. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 999–1006. ACM, 2010.

[94] Edward P. Manning. Using resource-limited nash memory to improve an othello evaluation function. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(1):40 –53, march 2010.

[95] Ben McKay, Mark J. Willis, and Geoffrey W. Barton. Using a tree structured genetic algorithm to perform symbolic regression. In *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)*, pages 487–492. IET, 1995.

[96] Thomas Miconi. Why coevolution doesn't "work": Superiority and progress in coevolution. In *EuroGP*, 2009.

[97] Geoffrey F. Miller and Dave Cliff. Protean behavior in dynamic games: arguments for the co-evolution of pursuit-evasion tactics. In *Proceedings of the third international conference on Simulation of adaptive behavior : from animals to animats*

3: from animals to animats 3, pages 411–420, Cambridge, MA, USA, 1994. MIT Press.

[98] Melanie Mitchell, James P. Crutchfield, and Peter T. Hraber. Evolving cellular automata to perform computations: mechanisms and impediments. *Physica D: Nonlinear Phenomena*, 75:361–391, 1994.

[99] Rolf H. Möhring. Algorithmic aspects of comparability graphs and interval graphs. *Graphs and Order: The Role of Graphs in the Theory of Ordered Sets and Its Applications*, pages 41–102, 1984.

[100] German A. Monroy, Kenneth O. Stanley, and Risto Miikkulainen. Coevolution of neural networks using a layered pareto archive. In Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 329–336, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[101] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[102] Jason Noble. Finding robust Texas Holdem poker strategies using Pareto coevolution and deterministic crowding. In *Proceedings of the 2002 International Conference on Machine Learning and Applications (ICMLA'02)*, pages 233–239. CSREA Press, 2002.

[103] Jason Noble and Richard A. Watson. Pareto coevolution: Using performance against coevolved opponents in a game as dimensions for pareto selection. In Lee Spector et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 493–500, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[104] Stefano Nolfi and Dario Floreano. Coevolving Predator and Prey Robots: Do "Arms Races" Arise in Artificial Evolution? *Artificial Life*, 4(4):311–335, 1998.

[105] Björn Olsson. Evaluation of a Simple Host-Parasite Genetic Algorithm. In *Evolutionary Programming VII: 7th International Conference, Ep98, San Diego, California, Usa, March 25-27, 1998: Proceedings*, page 53. Springer, 1998.

[106] Björn Olsson. *Algorithms for coevolution of solutions and fitness cases in asymmetric problem domains*. PhD thesis, University of Exeter, 1999.

[107] Björn Olsson. Co-evolutionary search in asymmetric spaces. *Information Sciences*, 133(3-4):103 – 125, 2001.

[108] Liviu Panait and Sean Luke. A comparison of two competitive fitness functions. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 503–511, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[109] Jan Paredis. Co-evolutionary constraint satisfaction. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Manner, editors, *Parallel Problem Solving from Nature PPSN III*, volume 866 of *Lecture Notes in Computer Science*, pages 46–55. Springer Berlin / Heidelberg, 1994.

[110] Jan Paredis. Steps toward co-evolutionary classification neural networks. In *Artificial Life IV: Proc. 4th Int. Workshop on the Synthesis and Simulation of Living Systems*, pages 102–108. Cambridge, MA: MIT Press, 1994.

[111] Jan Paredis. Coevolutionary computation. *Artificial Life*, 2(4):355–375, 1995.

[112] Jan Paredis. Coevolving cellular automata: Be aware of the red queen. In *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 393–400, 1997.

[113] Zdzisław. Pawlak. *Rough sets: Theoretical aspects of reasoning about data*. Springer, 1991.

[114] Jordan B. Pollack and Alan D. Blair. Co-evolution in the successful learning of backgammon strategy. *Machine Learning*, 32(3):225–240, 1998.

[115] Elena Popovici, Anthony Bucci, R. Paul Wiegand, and Edwin D. de Jong. *Handbook of Natural Computing*, chapter Coevolutionary Principles. Springer-Verlag, 2011.

[116] Elena Popovici and Kenneth De Jong. Understanding competitive co-evolutionary dynamics via fitness landscapes. In *Artificial Multiagent Symposium. Part of the 2004 AAAI Fall Symposium on Artificial Intelligence*, 2004.

[117] Elena Popovici and Kenneth De Jong. Monotonicity versus performance in co-optimization. In *FOGA '09: Proceedings of the tenth ACM SIGEVO workshop on Foundations of genetic algorithms*, pages 151–170, New York, NY, USA, 2009. ACM.

[118] Ran Raz and Shmuel Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 475–484. ACM New York, NY, USA, 1997.

[119] Craig Reynolds. Competition, coevolution and the game of tag. In R. A. Brooks and P. Maes, editors, *Artificial Life IV, Proceedings of the fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 59–69, MIT, Cambridge, MA, USA, 1994. MIT Press.

[120] Chrisotpher D. Rosin and Richard K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.

[121] Christopher D. Rosin. *Coevolutionary Search Among Adversaries*. PhD thesis, UNIVERSITY OF CALIFORNIA, SAN DIEGO, 1997.

[122] Christopher D. Rosin and Richard K. Belew. Methods for competitive co-evolution: Finding opponents worth beating. In Larry J. Eshelman, editor, *ICGA*, pages 373–381, San Francisco, CA, 1995. Morgan Kaufmann.

[123] Thomas P. Runarsson and Simon M. Lucas. Coevolution versus self-play temporal difference learning for acquiring position evaluation in small-board go. *Evolutionary Computation, IEEE Transactions on*, 9(6):628 – 640, dec. 2005.

[124] S.J. Russell, P. Norvig, J.F. Canny, J.M. Malik, and D.D. Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, NJ, 1995.

[125] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):211–229, 1959.

[126] Tatsuya Sato and Takaya Arita. Competitive co-evolutionary algorithms can solve function optimization problems. *Artificial Life and Robotics*, 14:440–443, 2009. 10.1007/s10015-009-0721-y.

[127] Travis C. Service and Daniel R. Tauritz. Co-optimization algorithms. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 387–388, New York, NY, USA, 2008. ACM.

[128] Yehonatan Shichel, Eran Ziserman, and Moshe Sipper. GP-robocode: Using genetic programming to evolve robocode players. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 143–154, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

[129] Karl Sims. Evolving 3D morphology and behavior by competition. *Artificial Life*, 1(4):353–372, 1994.

[130] Moshe Sipper. Attaining human-competitive game playing with genetic programming. In Samira El Yacoubi, Bastien Chopard, and Stefania Bandini, editors, *Proceedings of the 7th International Conference on Cellular Automata, for Research and Industry, ACRI*, volume 4173 of *Lecture Notes in Computer Science*, page 13, Perpignan, France, September 20-23 2006. Springer. Invited Lectures.

[131] Moshe Sipper and Eytan Ruppin. Co-evolving architectures for cellular machines. *Physica D: Nonlinear Phenomena*, 99(4):428–441, 1997.

[132] Kevin C. Smilak. Finding the ultimate video poker player using genetic programming. In John R. Koza, editor, *Genetic Algorithms and Genetic Programming at Stanford 1999*, pages 209–217. Stanford Bookstore, Stanford, California, 94305-3079 USA, 15 March 1999.

[133] Lee Spector and Herbert J. Bernstein. Communication capacities of some quantum gates, discovered in part through genetic programming. In *Proc. 6th Int. Conf. Quantum Communication, Measurement, and Computing (QCMC)*, pages 500–503, 2003.

[134] Kenneth O. Stanley, Bobby Bryant, and Risto Miikkulainen. Real-time neuroevolution in the nero video game. *Evolutionary Computation, IEEE Transactions on*, 9(6):653–668, 2005.

[135] B.H. Sumida and W.D. Hamilton. Both Wrightian and "parasite" peak shifts enhance genetic algorithm performance in the travelling salesman problem. In Ray

Paton, editor, *Computing with Biological Metaphors*, pages 254–279. Chapman and Hall, 1994.

[136] Marcin Szubert, Wojciech Jaśkowski, and Krzysztof Krawiec. Coevolutionary temporal difference learning for othello. In *IEEE Symposium on Computational Intelligence and Games*, pages 104–111, Milano, Italy, 2009.

[137] Joc Cing Tay, Cheun Hou Tng, and Chee Siong Chan. Environmental effects on the coevolution of pursuit and evasion strategies. *Genetic Programming and Evolvable Machines*, 9:5–37, 2008. Online First.

[138] Andrea G. B. Tettamanzi. Genetic programming without fitness. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 193–195, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore.

[139] William T. Trotter. *Combinatorics and partially ordered sets: Dimension theory.* Johns Hopkins University Press, 1992.

[140] Richard A. Watson and Jordan B. Pollack. Coevolutionary dynamics in a minimal substrate. In Lee Spector et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 702–709, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[141] Darrell Whitley, Soraya Rana, and Robert B. Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7(1):33–47, 1999.

[142] Mark Wittkamp and Luigi Barone. Evolving adaptive play for the game of spoof using genetic programming. In Sushil J. Louis and Graham Kendall, editors, *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG06)*, pages 164–172, University of Nevada, Reno, campus in Reno/Lake Tahoe, USA, 22 - 24 May 2006. IEEE.

[143] Yang Xiaomei, Zeng Jianchao, Liang Jiye, and Liang Jiahua. A genetic algorithm for job shop scheduling problem using co-evolution and competition mechanism. In *Artificial Intelligence and Computational Intelligence (AICI), 2010 International Conference on*, volume 2, pages 133 –136, oct. 2010.

[144] Liping Yang, Houkuan Huang, and Xiaohong Yang. An efficient pareto-coevolution archive. In *Natural Computation, 2007. ICNC 2007. Third International Conference on*, volume 4, pages 484–488, Aug. 2007.

[145] Liping Yang, Houkuan Huang, and Xiaohong Yang. A Simple Coevolution Archive Based on Bidirectional Dimension Extraction. In *2009 International Conference on Artificial Intelligence and Computational Intelligence*, pages 596–600. IEEE, 2009.

[146] Mihalis Yannakakis. The complexity of the partial order dimension problem. *SIAM Journal on Algebraic and Discrete Methods*, 3(3):351–358, 1982.

[147] Ting-Shuo Yo and Edwin D. de Jong. A comparison of evaluation methods in coevolution. In Hod Lipson, editor, *GECCO*, pages 479–487. ACM, 2007.

Politechnika Poznańska
Instytut Informatyki

Algorytmy dla Problemów Opartych na Testach

Wojciech Jaśkowski

Streszczenie rozprawy doktorskiej

Promotor

dr hab. inż. Krzysztof Krawiec

Poznań, 2011

1 Wstęp

1.1 Motywacja

Niniejsza praca wpisuje się w badania inteligencji obliczeniowej (ang. computational intelligence, [14]), dyscypliny zajmującej się rozwiązywaniem problemów za pomocą algorytmów inspirowanych biologicznie. Problemy rozważane w inteligencji obliczeniowej są podobne do tych będących przedmiotem dociekań w sztucznej inteligencji (ang. artificial intelligence, [38]). Różnica pomiędzy tymi dwoma została celnie wyrażona przez Lucasa [27, strona 45]: "*In AI research the emphasis is on producing apparently intelligent behaviour using whatever technuiques are appropriate for a given problem. In computational intelligence research, the emphasis is placed on intelligence being an emergent property*".

Inteligencja obliczeniowa zajmuje się między innymi środowiskami, w których pewne elementarne obiekty wchodzą ze sobą w interakcje. Programy grające uczą się poprzez rozgrywanie gier między sobą. Algorytmy uczenia maszynowego generują hipotezy i testują je na przykładach uczących. Algorytmy ewolucyjne symulują wyewoluowane projekty w różnych środowiskach. Wspólną cechą tych scenariuszy jest koncepcja *interakcji* pomiędzy *kandydatem* (ang. candidate, tu: strategia gracza, hipoteza, projekt) i *testem* (odpowiednio: strategia przeciwnika, przykład uczący, środowisko). Wspólną cechą tych problemów jest fakt, iż liczba testów, z którymi oddziałują rozwiązania może być bardzo duża lub nawet, w niektórych przypadkach, nieskończona.

Problemy, których przykłady podano powyżej, można zaliczyć do klasy problemów opartych na testach (ang. test-based problems, [8]). Istnieje wiele algorytmów zaprojektowanych dla konkretnych podklas problemów opartych na testach, np. metody statystycznego uczenia się dla problemów uczenia maszynowego lub algorytmy minimaksowe dla gier. Te metody korzystają jednak ze szczególnych, dodatkowych cech problemów, nieujętych w definicji problemów opartych na testach. Ogólną metodą dla problemów opartych na testach jest *kompetetywny algorytm koewolucyjny* (ang. competitive coevolution [2, 20, 36, 37]), który naśladuje koewolucję gatunków występującą w przyrodzie. Kompetytywny algorytm koewolucyjny wykorzystuje interakcje zachodzące pomiędzy kandydatami i testami, aby wytworzyć „wyścig zbrojeń“ tych dwóch kategorii osobni-

ków. Algorytm koewolucyjny używa mechanizmów znanych z algorytmów ewolucyjnych takich jak mutacja, krzyżowanie lub selekcja.

Algorytmy koewolucyjne stosowano do problemów modelowanych jako problemy oparte na testach takich jak projektowanie sieci neuronowych [22], uczenie się strategii dla gier strategicznych [39] lub znajdowanie reguł dla automatów komórkowych [24].

Mimo entuzjazmu, z jakim początkowo spotkały się algorytmy koewolucyjne, szybko okazało się, że metody te charakteryzują się trudną do przewidzenia dynamiką [23]. Ponadto, dla nietrywialnych problemów algorytmy koewolucyjne często nie są w stanie utrzymywać monotonicznego wzrostu [29]. Pokazano także, iż algorytmy koewolucyjne przejawiają wiele niepożądanych zachowań zwanych *patologiami* [23]. W konsekwencji, w przypadku wielu problemów opartych na testach, algorytmy koewolucyjne nie generują efektywnych rozwiązań.

Jednym z powodów takiego stanu rzeczy jest *agregacja* wyników interakcji pomiędzy osobnikami reprezentującymi kandydatów i testy [7]. Agregacja taka zachodzi zwykle podczas fazy oceny osobników, w której przyporządkowuje się im skalarną wartość interpretowaną jako przystosowanie (ang. fitness). Istnieje więc potrzeba zaprojektowania nowych mechanizmów dla algorytmów koewolucyjnych, które unikałyby agregacji. Jest to jednocześnie motywacja dla badań opisanych w tej rozprawie.

1.2 Cel pracy

W kontekście powyżej nakreślonej problematyki, głównym celem pracy jest *analiza problemów opartych na testach, ich cech, i zaprojektowanie nowych algorytmów koewolucyjnych, unikających problemu związanego z agregacją wyników.* Cele pośrednie pracy obejmują następujące zagadnienia:

- Zaprojektowanie algorytmu koewolucyjnego, który nie wymaga agregacji wyników interakcji pomiędzy osobnikami w fazie oceny.
- Teoretyczna analiza dynamiki tego algorytmu.
- Teoretyczna analiza wewnętrznej struktury problemów opartych na testach w oparciu o koncepcję Pareto-koewolucji, która unika agregacji wyników, traktując problem oparty na testach jako zadanie optymalizacji wielokryterialnej.
- Zaprojektowanie efektywnego algorytmu ekstrakcji wewnętrznej struktury problemów opartych na testach.

- Zaprojektowanie koewolucyjnego algorytmu dla problemów opartych na testach, wykorzystującego ekstrakcję wewnętrznej struktury problemu.
- Eksperymentalna weryfikacja zaproponowanych koncepcji i algorytmów na sztucznych i rzeczywistych problemach.

2 Problemy oparte na testach

2.1 Definicja

W teorii optymalizacji problem optymalizacyjny definiuje się poprzez określenie jego dziedziny oraz funkcji celu zdefiniowanej na jej elementach. Klasyczny problem optymalizacyjny polega na znalezieniu takiego elementu dziedziny, który maksymalizuje lub minimalizuje zadaną funkcję celu. Takie sformułowanie problemu pozwala na modelowanie wielu sytuacji praktycznych. Istnieją jednak problemy, w których poznanie wartości funkcji celu w danym punkcie jest na tyle kosztowne obliczeniowo, że w praktyce niemożliwe do osiągnięcia. Przykładem jest szukanie najlepszej strategii gracza w grze dwuosobowej, np. Go, która jest uznawana za jedno z największych wyzwań sztucznej inteligencji [6, 5]. Funkcję celu można w tym przypadku zdefiniować jako maksymalizację wartości oczekiwanej wyniku gry po wszystkich możliwych strategiach przeciwnika, występujących z równym prawdopodobieństwem. Obliczenie wartości funkcji celu dla danej strategii gracza czarnego (potencjalne rozwiązanie) wymaga zatem rozegrania gier ze wszystkimi możliwymi strategiami gracza białego (testy). Strategii tych jest jednak tak dużo, że jest to w praktyce niemożliwe.

Problemy które charakteryzują się tym, iż jakość potencjalnego rozwiązania zależy od wyników interakcji z elementami (zwykle dużego) zbioru testów nazywane są problemami opartymi na testach [8]. Formalnie definiuje się je następująco:

Definicja 1. Problem oparty na testach jest obiektem $\mathcal{H} = (S, T, G, \mathcal{P}, \mathcal{P}^+)$, składającym się ze:

- *zbioru kandydatów* S (ang. candidates [8] or candidate solutions [21]),
- *zbioru testów* T,
- *funkcji interakcji* $G : S \times T \to O$, gdzie O jest zbiorem całkowicie uporządkowanym,
- *zbioru potencjalnych rozwiązań* $\mathcal{P}$ zbudowanym na zbiorze kandydatów, oraz

- *pojęcia rozwiązania* (ang. solution concept [15]), które rozdziela zbiór potencjalnych rozwiązań $\mathcal{P}$ na rozłączne podzbiory *rozwiązań* problemu $\mathcal{P}^+$ i elementów niebędących rozwiązaniami problemu $\mathcal{P}^-$.

Aby zilustrować powyższą definicję rozważmy grę w szachy. Załóżmy, że szukamy najlepszej strategii dla gracza białego, więc zbiór S zawiera wszystkie możliwe strategie tego gracza, podczas gdy zbiór T zawiera wszystkie możliwe strategie gracza czarnego. Funkcja interakcji G jest w tym przypadku interpretowana jako rozgrywka pomiędzy graczami, a jej przeciwdziedziną jest uporządkowany zbiór $\{porażka < remis < wygrana\}$. Zbiór potencjalnych rozwiązań $\mathcal{P}$ jest tożsamy ze zbiorem kandydatów S, a rozwiązaniami problemu należącymi do zbioru $\mathcal{P}^+$ są kandydaci maksymalizujący oczekiwany wynik gry.

Mimo iż *pojęcie rozwiązania* jednoznacznie determinuje rozwiązanie którego szukamy, to jeśli dwa potencjalne rozwiązania nie należą do zbioru $\mathcal{P}^+$, pojęcie rozwiązania nie udziela odpowiedzi na pytanie: które z nich jest preferowane? Dlatego, pojęcie rozwiązania z powyższej definicji można zastąpić poprzez

- *relację preferencji* $\preceq$ na zbiorze $\mathcal{P}$, gdzie $P_1 \preceq P_2$ jest interpretowane jako P_1 jest nie gorsze niż P_2, gdy $P_1, P_2 \in \mathcal{P}$.

Relacja preferencji jest, w ogólności, preporządkiem i jest uogólnieniem *pojęcia rozwiązania*. W szczególnym przypadku, zbiór jej maksymalnych elementów może być równy zbiorowi rozwiązań $\mathcal{P}^+$. Relacja preferencji uogólniająca pojęcie rozwiązania może być zdefiniowana na wiele sposobów i zależy tylko i wyłącznie od preferencji decydenta. Gdy problem oparty na testach występuje bez zdefiniowanej relacji preferencji, nazywamy go problemem koprzeszukiwania (ang. co-search test-based problem), w przeciwnym wypadku mówimy o problemie kooptymalizacyjnym (ang. *co-optimization test-based problem* [34]).

2.2 Pojęcie rozwiązania

W przykładzie dotyczącym gry w szachy, opisanym w sekcji 2, założyliśmy, że zbiór potencjalnych rozwiązań $\mathcal{P}$ jest identyczny ze zbiorem kandydatów S i, w konsekwencji, rozwiązania problemu (elementy zbioru $\mathcal{P}^+$) są elementami zbioru S. Zaznaczmy jednak, że nie musi tak być w ogólności. Na przykład, możemy wymagać, aby rozwiązanie składało się ze wszystkich Pareto-niezdominowanych kandydatów.

Mimo iż *pojęcie rozwiązania* może być zdefiniowane dowolnie, warto jest zidentyfikować te pojęcia rozwiązania, które są często używane. Poniżej, za [34] podajemy dwa

z nich:

Maksymalizacja Wartości Oczekiwanej W tym przypadku $\mathcal{P} = S$ i szukamy kandydata, który maksymalizuje oczekiwany wynik interakcji, zatem

$$\mathcal{P}^+ = \text{argmax}_{s \in S} \text{E}[G(s,t)],$$

gdzie E jest operatorem wartości oczekiwanej, a t jest losowana ze zbioru T.

Zbiór Pareto Optymalny To pojęcie rozwiązania traktuje każdy test jako oddzielne kryterium a cały problem oparty na testach jako optymalizację wielokryterialną. W zbiorze potencjalnych rozwiązań $\mathcal{P} = 2^S$ szukamy frontu Pareto

$$\mathcal{F} = \left\{ s \in S \mid \forall_{s' \in S} \left(\exists_{t \in T} G(s,t) \leq G(s',t) \implies G(s,t) = G(s',t) \right) \right\}.$$

W tym przypadku zbiór rozwiązań zawiera jeden element: front Pareto, ergo

$$\mathcal{P}^+ = \{\mathcal{F}\}.$$

2.3 Rozwiązywanie problemów opartych na testach za pomocą algorytmów koewolucyjnych

2.3.1 Algorytmy koewolucyjne

Kompetytywne algorytmy koewolucyjne, podobnie jak algorytmy ewolucyjne [3], utrzymują zbiór osobników, które są poddawane operatorom ewolucyjnym takim jak mutacja, krzyżowania i selekcja. Siłą napędową algorytmów koewolucyjnych jest ciągły wyścig zbrojeń pomiędzy (zwykle) dwoma konkurującymi ze sobą populacjami osobników [32]. Odpowiada to koewolucji między gatunkami obserwowanej w przyrodzie. Różnica pomiędzy algorytmami koewolucyjnymi i ewolucyjnymi leży w fazie oceny osobników. Algorytmy ewolucyjne, rozwiązujące problemy optymalizacyjne, mają dostęp do obiektywnej funkcji celu, więc ocena przystosowania osobników jest obliczana bezpośrednio. W algorytmach koewolucyjnych, osobniki oceniane są zwykle poprzez agregację wyników wielokrotnych interakcji z osobnikami z przeciwnej populacji. W związku z tym w algorytmach koewolucyjnych funkcja interakcji zastępuje obiektywną funkcję celu obecną w algorytmach ewolucyjnych.

Algorytmy koewolucyjne są naturalną metodą rozwiązywania problemów opartych na

testach z kilku powodów. Po pierwsze, nie wymagają dostępu do obiektywnej funkcji celu, a jedynie do wyników interakcji pomiędzy osobnikami (funkcja interakcji). Po drugie, utrzymują (zwykle) dwie populacje osobników, co odpowiada dwóm rolom obecnym w problemach opartych na testach: kandydatom i testom. Po trzecie, są one generycznymi metaheurystykami, które działają na wszystkich problemach opartych na testach, jeśli tylko zdefiniuje się odpowiednie operatory genetyczne.

2.3.2 Patologie i archiwa

Algorytmy koewolucyjne przejawiają złożoną i trudną do zrozumienia dynamikę [23, 35], w której zidentyfikowano tzw. patologie koewolucyjne (ang. *coevolutionary pathologies*) utrudniające lub uniemożliwiające monotoniczny wzrost jakości rozwiązań w czasie działania algorytmów [17]. Przykładami takich patologii są *zmowa* [4], *efekt Czerwonej Królowej* [33], *wpadanie w cykl* (ang. *cycling* [40]), *zapominanie* [16] i *nadmierna specjalizacja* [40].

Celem archiwów koewolucyjnych jest podtrzymywanie monotonicznego wzrostu jakości rozwiązań podczas działania algorytmów koewolucyjnych i, tym samym, przeciwdziałanie patologiom. Typowe archiwum jest zbiorem wybranych, (zwykle) różnorodnych osobników znalezionych przez algorytm do tej pory. Gdy nowe osobniki dodawane są do archiwum, stare, jeśli już nie są przydatne, mogę zostać usunięte. Archiwa z założenia pełnią rolę podobną do *elityzmu* w algorytmach ewolucyjnych. Przykłady archiwów koewolucyjnych obejmują metody Hall of Fame [37], DECA [12], EPCA [41], Nash Memory [19], DELPHI [13], IPCA [9] i LAPCA [10].

Jeśli archiwum jest częścią algorytmu koewolucyjnego, możemy mówić o *schemacie generator-archiwum* [11] dla rozwiązywania problemów opartych na testach. Rola generatora może być odgrywana przez każdy algorytm, który jest w stanie generować nowe testy i nowych kandydatów, niezależnie od tego w jaki sposób będzie to robił (np. przez mutację osobników z aktualnej populacji).

Schemat generator-archiwum jest przedstawiony jako Algorytm 2.1. Po inicjalizacji początkowych populacji (linie 2-4) następuje główna pętla (linie 5-13). W pętli, generowane są kandydaci i testy (np. poprzez mutację i krzyżowanie osobników z aktualnej populacji S' i T', linie 6-7), a następnie wszystkie osobniki są zgłaszane do archiwum, które zwykle akceptuje niektóre z nich, a inne odrzuca. Ostatecznie osobniki z obu populacji poddawane są ocenie i selekcji. Zauważmy, że archiwum, oprócz tego że jest uaktualniane (w linii 10), jest również używane (w liniach 6 i 7), aby potencjalnie dostarczyć pożądanego materiału genetycznego dla generatora.

Algorytm 2.1 Schemat generator-archiwum dla rozwiązywania problemów opartych na testach.

```
1:  procedure CoEvolution
2:      S', T' ← Initialize populations
3:      S_arch ← ∅
4:      T_arch ← ∅
5:      while ¬stopped do
6:          S_new ← GenerateNewSolutions(S', S_arch)
7:          T_new ← GenerateNewTests(T', T_arch)
8:          S' ← S' ∪ S_new
9:          T' ← T' ∪ T_new
10:         Archive.Submit(S', T')          ▷ Updates S_arch and T_arch
11:         Evaluate(S', T')
12:         S', T' ← Select(S', T')
13:     end while
14: end procedure
```

3 Algorytm Fitnessless Coevolution

W tym rozdziale rozważamy symetryczne problemy oparte na testach, czyli takie, w których zbiór testów jest identyczny ze zbiorem kandydatów ($S = T$). Dla takich problemów można użyć *koewolucji jednopopulacyjnej* [28], w której utrzymywana jest tylko jedna populacja i interakcje odbywają się pomiędzy jej osobnikami. Ze względu na to iż ten mechanizm jest głównie stosowany dla gier, w tym rozdziale używać będziemy terminologii związanej z grami.

Jedna z najważniejszych decyzji przy projektowaniu algorytmu koewolucyjnego jest wybór metody oceny przystosowania osobników w trakcie działania algorytmu. Zaprojektowano kilka takich metod. Jedną z nich jest *system kołowy* (ang. round-robin tournament), w ramach którego każdy osobnik z populacji gra (interakcja) z każdym innym, a ocena przystosowania osobnika jest sumą otrzymanych wyników interakcji. Alternatywną metodą jest *system pucharowy* [1], w którym osobniki otrzymują ocenę przystosowania równą liczbie gier, które w tym systemie wygrali. Jest również metoda *k-losowych osobników* [36], która od każdego osobnika wymaga rozegrania gier z k losowo dobranymi przeciwnikami.

Wszystkie te metody wpisują się w schemat ocena-selekcja-rekombinacja. Jednak przecież rozgrywanie gier jest selektywne samo w sobie, warto więc postawić sobie pytanie: dlaczego nie użyć wyniku gry, aby decydować bezpośrednio o selekcji? Ta obserwacja doprowadziła nas do zdefiniowania algorytmu koewolucyjnego o nazwie *Fitnessless Coevolution,* w której omijamy fazę oceny dzięki selekcji, która nie wymaga ocen przystosowania osobników. Technicznie rzecz biorąc, w fazie selekcji używamy systemu pucharowego, który polega na wielokrotnym rozgrywaniu potyczek dla (niewielkich) podzbiorów k wylosowanych osobników. Osobnik, który wygrywa cały turniej jest wynikiem selekcji. Selekcja jest stosowana n razy, aby wygenerować nowa populację o wielkości n.

W pracy zostało dowiedzione, iż pod warunkiem tranzytywności macierzy wypłat G, algorytm Fitnessless Coevolution jest dynamicznie równoważny algorytmowi ewolucyjnemu używającemu selekcji turniejowej. Oznacza to że wychodząc od tej samej populacji i zakładając identyczne wyniki losowań, populacje w kolejnych pokoleniach obu metod składają się z identycznych osobników.

W części eksperymentalnej porównaliśmy zaproponowany algorytm z systemem kołowym, pucharowym oraz metodą k-losowych osobników na zestawie kilku problemów: gra w kółko i krzyżyk, gra Nim, funkcje Rosenbrock oraz Rastrigin. Wyniki pokazały, iż algorytm Fitnessless Coevolution jest na wszystkich problemach statystycznie nie gorszy niż metody konkurencyjne, a na niektórych daje wyniki lepsze.

4 Zastosowanie Fitnessless Coevolution

W tym rozdziale stosujemy metodę zaproponowaną w rozdziale poprzednim do rzeczywistego problemu gry Ant Wars, która była problemem konkursowym na konferencji *Genetic and Evolutionary Computation Conference* (GECCO, Londyn, 7–12 lipca 2007), największej międzynarodowej konferencji obliczeń ewolucyjnych i genetycznych. Konkurs polegał na wyewoluowaniu strategii dla wirtualnego agenta (mrówki), który w obecności przeciwnika porusza się na toroidalnej planszy i zbiera z niej pokarm. Ant Wars jest grą probabilistyczną, w której agenci mają tylko częściową informację o aktualnym stanie gry. Strategia gracza jest zakodowana w postaci kontrolera będącego programem w języku ANSI-C.

Uzyskane przez nas rozwiązanie o nazwie BrilliAnt zostało wyewoluowane za pomocą algorytmu Fitnessless Coevolution oraz programowania genetycznego [25] z systemem typów [30] i zostało uznane za najlepsze w konkursie.

Wyewoluowanie agenta dla tej gry wymagało zaprojektowania kodowania jego strategii tak, aby uwzględniała ona nie tylko wiedzę agenta o aktualnym stanie gry, ale również aby agent przy podejmowaniu decyzji mógł korzystać z wiedzy zebranej we wcześniejszych etapach rozgrywki.

Aby ocenić jakość wyewoluowanych rozwiązań zaimplementowaliśmy ręcznie kilka nietrywialnych strategii graczy w języku ANSI-C i przeprowadziliśmy turniej kołowy pomiędzy wszystkimi posiadanymi agentami. Wyniki konfrontacji agentów przedstawia tabela 4.1. Wynika z niej, iż najlepsze rozwiązanie jakim dysponujemy (ExpertAnt)[1] zostało osiągnięte za pomocą Fitnessless Coevolution i jest lepsze niż wszystkie rozwiązania zaprojektowane ręcznie.

Przeprowadzono także analizę behawioralną uzyskanego rozwiązania. Wynika z niej, iż BrilliAnt wyewoluował zaskakująco zaawansowane mechanizmy analizy planszy i podejmowania decyzji, podobne do tych, które zaprojektowaliśmy ręcznie.

[1] ExpertAnt został wyewoluowany dopiero po zakończeniu konkursu.

Tablica 4.1: Wyniki turnieju kołowego. Wyewoluowane rozwiązania zaznaczono pogrubioną czcionką.

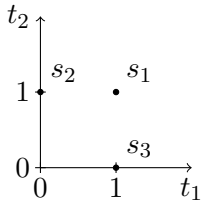
Gracz	Liczba wygranych gier
ExpertAnt	760,669
HyperHumant	754,303
BrilliAnt	753,212
EvolAnt3	736,862
SuperHumant	725,269
EvolAnt2	721,856
EvolAnt1	699,320
SmartHumant	448,509

5 Układ współrzędnych dla problemów opartych na testach

Agregacja wyników interakcji jest jednym z powodów, dla których obserwujemy patologie koewolucyjne. Mechanizmem, który pozwala uniknąć agregacji jest Pareto-koewolucja (ang. Pareto coevolution) [18, 31], która traktuje każdy test jako osobne *kryterium*, a cały problem jako *zadanie optymalizacji wielokryterialn*ej. Pozwala to na zastosowanie relacji Pareto dominacji — kandydat s_1 jest nie gorszy niż kandydat s_2 wtedy i tylko wtedy, gdy na wszystkich testach osiąga wyniki co najmniej takie same jak s_2 (Rys. 5.0.1). Niestety w problemach opartych na testach liczba testów jest zwykle bardzo duża (np. w prostej grze w kółko i krzyżyk liczba strategii dla gracza grającego krzyżykami jest rzędu 10^{162}). Stąd wymiarowość takiej przestrzeni przeszukiwania jest ogromna.

Liczba kryteriów w Pareto koewolucji może jednak zostać zmniejszona, ponieważ wiele problemów opartych na testach posiada *wewnętrzną strukturę*. Ta struktura objawia się poprzez fakt istnienia grup testów, które oceniają ten sam aspekt (tą samą „zdolność“) kandydata, ale z różną intensywnością. Zamiast więc konstruować dla każdego z tych testów osobne kryterium, można umieścić wszystkie testy z takiej grupy, uporządkowane pod względem trudności, na wspólnej osi współrzędnych, którą nazywamy *ukrytym kryterium* problemu (ang. underlying objective, [13]). Ukryte kryteria nie są znane a priori i muszą być odkryte podczas eksploracji problemu. Formalnie wewnętrzną strukturę problemu definiuje się w postaci *układu współrzędnych* [8], składającego się z wielu takich kryteriów. Istotną cechą układu współrzędnych jest fakt, iż poddając „kompresji“ oryginalne kryteria do zbioru ukrytych kryteriów, jednocześnie zachowuje się relacje dominacji pomiędzy kandydatami a testami. Celnie wyraził to Bucci i współpracownicy: *the structure space captures essential information about a problem in an efficient manner* [8].

W tym rozdziale zakładamy, że przeciwdziedzina funkcji interakcji jest zbiorem binarnym $\{0 < 1\}$ i formalnie definiujemy układ współrzędnych. Następnie przeprowadzamy szczegółową analizę właściwości układów współrzędnych dla problemów opartych na testach, proponując między innymi równoważną definicję alternatywną. Ponadto pokazu-



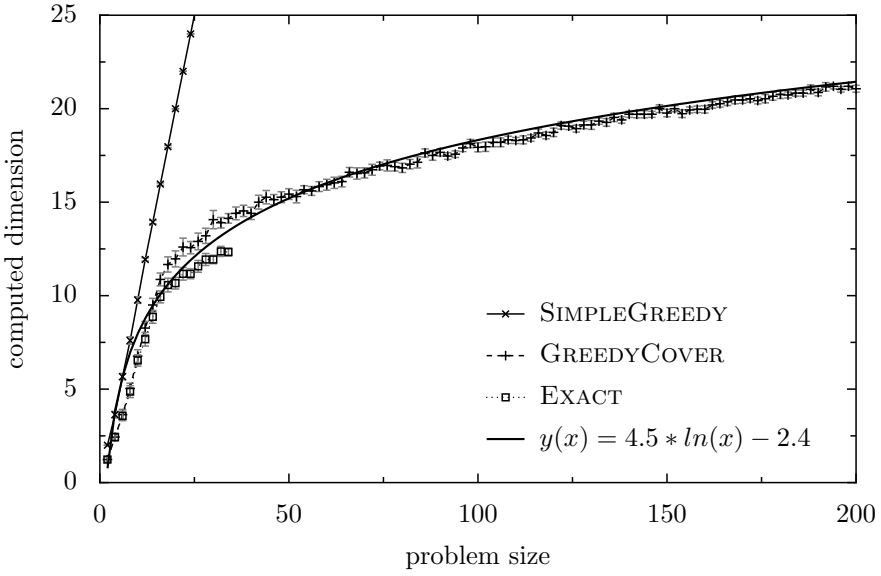
Rysunek 5.0.1: Pareto koewolucja. Każdy test (tutaj: t_1 i t_2) traktowany jest jak osobne kryterium. Kandydaci (tutaj: s_1, s_2 i s_3) są zanurzeni w przestrzeni rozpiętej przez kryteria (testy) i umieszczeni zgodnie z wynikami interakcji z testami. Przykładowo, kandydat s_3 rozwiązuje test t_1, ale nie test t_2.

jemy zależność pomiędzy *wymiarowością problemu*, rozumianą jako liczba osi w układzie współrzędnych, oraz szerokością częściowo uporządkowanego zbioru testów. Wyprowadzamy również również wzory na górne i dolne ograniczenie na wymiarowość problemu. W części teoretycznej dowodzimy, że problem obliczenia wymiarowości problemu jest NP-trudny.

Aby wyznaczyć minimalny układ współrzędnych dla problemu opartego na testach (i tym samym jego wymiarowość) zaproponowaliśmy algorytm dokładny (Exact) oraz efektywną heurystykę (GreedyCover). Algorytmy te zostały porównanie z algorytmem SimpleGreedy podanym wcześniej przez Bucci'ego i współpracowników [8]. Eksperymenty obliczeniowe pokazały, iż algorytm GreedyCover przy podobnej złożoności obliczeniowej daje znacznie lepsze wyniki niż SimpleGreedy. Wynik eksperymentu dla losowego problemu przedstawia Rys. 5.0.2.

Ponadto, algorytm GreedyCover został zastosowany do dwóch gier numerycznych (ang. numbers games, [40]): compare-on-one [13] i compare-on-all [8], których wymiarowość jest znana. Pokazaliśmy, że o ile dla compare-on-one algorytm poprawnie znajduje wymiarowość problemu, to znacznie ją przeszacowuje dla problemu compare-on-all.

Algorytm GreedyCover zastosowaliśmy również do *problemu klasyfikacji gęstości* (ang. density classification task), oraz do *gry w kółko i krzyżyk*, aby estymować wymiarowość tych problemów. Wyniki eksperymentów obliczeniowych wskazują na fakt, iż wymiarowość gry w kółko i krzyżyk jest mniejsza niż wymiarowość problemu klasyfikacji gęstości, mimo iż liczba testów w tym pierwszym znacznie przewyższa liczbę testów w drugim.



Rysunek 5.0.2: Porównanie trzech algorytmów: SimpleGreedy, Exact i GreedyCover na losowym problemie w funkcji wielkości problemu.

6 Algorytm Coordinate System Archive

W tym rozdziale wprowadzamy nowy algorytm archiwum koewolucyjnego o nazwie Coordinate System Archive (COSA), który bazuje na idei ekstrakcji struktury problemu za pomocą układów współrzędnych, wprowadzonych w rozdziale 5. COSA, przedstawiony jako Alg. 6.1, może współpracować z dowolnym generatorem w ramach schematu generator-archiwum. Algorytm w każdej iteracji wyznacza układ współrzędnych problemu złożonego z kandydatów i testów bieżącej populacji za pomocą algorytmu GreedyCover. Archiwum przechowuje po jednym teście z każdej osi współrzędnych układu. Dodatkowo, COSA utrzymuje *zbiór Pareto*, zawierający Pareto niezdominowanych kandydatów, reprezentujący najlepsze potencjalne rozwiązanie problemu znalezione do tej pory. Dzięki temu, jeśli rozwiązywany problem jest niskiej wymiarowości, liczba elementów przechowywanych przez archiwum jest mała. W rezultacie, COSA jest efektywna obliczeniowo.

COSA została porównana z innymi archiwami koewolucyjnymi: Iterated ParetoCoevolutionary Archive (IPCA) [9, 11] i Layered Pareto-Coevolutionary Archive (LAPCA) [10] na dwóch sztucznych problemach: compare-on-one [13] i compare-on-all [8]. Do porównania wydajności algorytmów użyte zostały dwie różne miary postępu. Wyniki eksperymentów obliczeniowych pokazały, iż COSA jest znacznie lepsza niż pozostałe algorytmy na problemie compare-on-one (w szczególności, na jego wielowymiarowej wersji). Z kolei na problemie compare-on-all algorytm COSA jest gorszy niż IPCA i LAPCA. Wynika to z charakterystyki algorytmu GreedyCover, który, jak pokazano w poprzednim rozdziale, idealnie znajduje wymiarowość compare-on-one, ale znacznie przeszacowuje wymiarowość dla problemu compare-on-all

Algorytm 6.1 Coordinate System Archive (COSA)

procedure SUBMIT(S_{new}, T_{new})
  $T_{tmp} \leftarrow T_{arch} \cup T_{new}$
  $S_{tmp} \leftarrow S_{arch} \cup S_{new}$
  $T_{tmp} \leftarrow$ GETUNIQUE(T_{tmp}, S_{tmp})
  $S_{tmp} \leftarrow$ GETUNIQUE(S_{tmp}, T_{tmp})
  $S_{Pareto} \leftarrow \{s \in S_{tmp} | \forall_{s' \in S_{tmp}} s' \leq_{T_{tmp}} s\}$
  $T_{base} \leftarrow$ FINDBASEAXISSET(T_{tmp}, S_{Pareto})
  $S_{req} \leftarrow$ PAIRSETCOVER($S_{Pareto}, S_{tmp}, T_{base}$)
  $T_{req} \leftarrow$ PAIRSETCOVER($T_{base}, T_{tmp}, S_{Pareto}$)
  $S_{arch} \leftarrow S_{req}$
  $T_{arch} \leftarrow T_{req}$
end procedure

procedure GETUNIQUE(A, B)
  $U \leftarrow \emptyset$
  for $a \in A$ **do**
    $I \leftarrow \{e \in A | \forall_{b \in B} G(a, b) = G(e, b)\}$
    $U \leftarrow U \cup$ oldest individual from I
    $A \leftarrow A \setminus I$
  end for
  return U
end procedure

procedure PAIRSETCOVER(A_{must}, A, B)
  $A \leftarrow A \setminus A_{must}$
  $\mathcal{N} \leftarrow \{(b_1, b_2) | b_1, b_2 \in B$ ▷ Pairs to be ordered
    $\wedge \exists_{a \in A} b_1 <_a b_2) \wedge \nexists_{a \in A_{must}} b_1 <_a b_2)\}$
  $V \leftarrow A_{must}$
  while $\mathcal{N} \neq \emptyset$ **do** ▷ Are all pairs ordered?
    $u \leftarrow \text{argmax}_{a \in A \setminus V} |\{(b_1, b_2) \in \mathcal{N} | b_1 <_a b_2\}|$
    $\mathcal{N} \leftarrow \mathcal{N} \setminus \{(b_1, b_2) \in \mathcal{N} | b_1 <_u b_2\}$
    $V \leftarrow V \cup \{u\}$
  end while
  return V
end procedure

Algorytm 6.2 Procedura znajdująca testy, które powinny być trzymane w archiwum.

```
1: procedure FindBaseAxisSet(T_tmp, S_Pareto)
2:     C ← ChainPartition(T_tmp, ≤)
3:     n_dims = |C|
4:     S_list ← S_Pareto sorted descendingly by min(GetPos(s, C))
5:     for s ∈ S_list do
6:         T' ← {t ∈ T_tmp | G(s, t)}
7:         (A, found) ← the greatest antichain in poset (T', ≤)
8:         if found then
9:             return A
10:        end if
11:    end for
12:    return T_tmp
13: end procedure
14:
15: procedure ChainPartition(X, P)
16:    return minimal chain partition of poset (X, P)
17: end procedure
18:
19: procedure GetPos(s, C)
20:    n_dims ← |C|
21:    P ← array[1 ... n_dims]
22:    for i = 1 ... n_dims do
23:        P[i] ← |{c ∈ C[i] | G(s, c)}|
24:    end for
25:    return P
26: end procedure
```

7 Podsumowanie

Klasa problemów opartych na testach obejmuje zadania pochodzące z wielu dyscyplin. Koewolucja kompetytywna jest ogólną metodą rozwiązywania takich problemów. Agregacja wyników interakcji podczas oceny przystosowania osobników jest wadą algorytmów koewolucyjnych, która objawia się patologiami koewolucyjnymi. Patologie są interesującymi fenomenami dla biologów i lub badaczy zajmujących się dziedziną sztucznego życia [26], której celem jest imitowanie życia biologicznego przy pomocy komputerów. Z kolei inteligencja obliczeniowa kładzie nacisk na *użyteczność* metod inspirowanych biologicznie. Skoro więc koewolucja, w formie jaką znamy z przyrody, przejawia wiele zachowań utrudniających otrzymywanie dobrych rozwiązań, uczynienie algorytmów koewolucyjnych efektywnymi wymaga ich ulepszenia poprzez wprowadzenie do nich dodatkowych mechanizmów, nieznanych w naturze. Zaproponowane w tej pracy algorytmy Fitnessless Coevolution i COSA są krokami w tym właśnie kierunku.

Podsumowując, najważniejsze wyniki pracy to:

- Wprowadzenie algorytmu Fitnessless Coevolution dla problemów opartych na testach.
- Dowód twierdzenia mówiącego że, jeśli macierz wypłat jest tranzytywna, algorytm Fitnessless Coevolution jest dynamicznie równoważny algorytmowi genetycznemu używającemu selekcji turniejowej.
- Studium przypadku, w którym algorytm Fitnessless Coevolution zastosowano do gry Ant Wars, kodując rozwiązania (strategie graczy) za pomocą programowania genetycznego.
- Analiza układów współrzędnych dla problemów opartych na testach, obejmująca:
  - Alternatywną definicję układu współrzędnych, równoważną definicji oryginalnej. [8]
  - Dowód kilku interesujących właściwości układów współrzędnych.
  - Dowód twierdzenia, iż problem szukania minimalnego układu współrzędnego (tym samym wymiarowości problemu) jest NP-trudny.

- Zaprojektowanie i analiza trzech algorytmów konstruujących układ współrzędny dla problemów opartych na testach, w tym zaprojektowanie algorytmu GreedyCover, który jest lepszy niż najlepszy dotąd znany algorytmy SimpleGreedy.
- Pokazanie, iż wymiarowość problemu jest zazwyczaj znacznie mniejsza niż liczba testów go charakteryzujących.
- Zaprojektowanie algorytmu archiwum koewolucyjnego COSA, który stanowi dowód na możliwość praktycznego wykorzystania koncepcji układów współrzędnych.

Bibliografia

[1] Peter J. Angeline and Jordan B. Pollack. Competitive environments evolve better solutions for complex tasks. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 264–270, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[2] Robert Axelrod. The evolution of strategies in the iterated prisoner's dilemma. In L. Davis, editor, *Genetic Algorithms in Simulated Annealing*, pages 32–41. Pitman, London, 1987.

[3] Thomas B
ąck, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation*. Oxford University Press, 1997.

[4] Alan D. Blair and Jordan B. Pollack. What makes a good co-evolutionary learning environment. *Australian Journal of Intelligent Information Processing Systems*, 4(3/4):166–175, 1997.

[5] Bruno Bouzy and Tristan Cazenave. Computer go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[6] Bernd Brügmann. Monte Carlo Go. Unpublished technical report, 1993.

[7] Anthony Bucci. *Emergent Geometric Organization and Informative Dimensions in Coevolutionary Algorithms*. PhD thesis, Michtom School of Computer Science, Brandeis University, 2007.

[8] Anthony Bucci, Jordan B. Pollack, and Edwin de Jong. Automated extraction of problem structure. In Kalyanmoy Deb et al., editor, *Genetic and Evolutionary Computation – GECCO-2004, Part I*, volume 3102 of *Lecture Notes in Computer Science*, pages 501–512, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[9] Edwin D. de Jong. The Incremental Pareto-Coevolution Archive. In Kalyanmoy Deb et al., editor, *Genetic and Evolutionary Computation–GECCO 2004. Proceedings*

of the Genetic and Evolutionary Computation Conference. Part I, pages 525–536, Seattle, Washington, USA, June 2004. Springer-Verlag, Lecture Notes in Computer Science Vol. 3102.

[10] Edwin D. de Jong. Towards a Bounded Pareto-Coevolution Archive. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, volume 2, pages 2341–2348, Portland, Oregon, USA, June 2004. IEEE Service Center.

[11] Edwin D. de Jong. A Monotonic Archive for Pareto-Coevolution. *Evolutionary Computation*, 15(1):61–93, Spring 2007.

[12] Edwin D. de Jong and Anthony Bucci. DECA: dimension extracting coevolutionary algorithm. In Mike Cattolico et al., editor, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 313–320, Seattle, Washington, USA, 2006. ACM Press.

[13] Edwin D. de Jong and Jordan B. Pollack. Ideal Evaluation from Coevolution. *Evolutionary Computation*, 12(2):159–192, Summer 2004.

[14] Andries P. Engelbrecht. *Computational intelligence: an introduction.* Wiley, 2007.

[15] Sevan G. Ficici. *Solution concepts in coevolutionary algorithms.* PhD thesis, Waltham, MA, USA, 2004. Adviser-Pollack, Jordan B.

[16] Sevan G. Ficici. Multiobjective Optimization and Coevolution. In Joshua Knowles, David Corne, and Kalyanmoy Deb, editors, *Multi-Objective Problem Solving from Nature: From Concepts to Applications*, pages 31–52. Springer, Berlin, 2008. ISBN 978-3-540-72963-1.

[17] Sevan G. Ficici and Jordan B. Pollack. Challenges in coevolutionary learning: Arms-race dynamics, open-endedness, and mediocre stable states. In *Proceedings of the Sixth International Conference on Artificial Life*, pages 238–247. MIT Press, 1998.

[18] Sevan G. Ficici and Jordan B. Pollack. Pareto optimality in coevolutionary learning. In Jozef Kelemen and Petr Sosík, editors, *Advances in Artificial Life, 6th European Conference, ECAL 2001*, volume 2159 of *Lecture Notes in Computer Science*, pages 316–325, Prague, Czech Republic, 2001. Springer.

[19] Sevan G. Ficici and Jordan B. Pollack. A game-theoretic memory mechanism for coevolution. In E. Cantú-Paz et al., editor, *Genetic and Evolutionary Computation - GECCO 2003*, volume 2723 of *Lecture Notes in Computer Science*, pages 286–297, Chicago, IL, 2003. Springer.

[20] W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial life II*, volume 10 of *Sante Fe Institute Studies in the Sciences of Complexity*, pages 313–324, Redwood City, Calif., 1992. Addison-Wesley.

[21] Wojciech Jaśkowski and Krzysztof Krawiec. Coordinate system archive for coevolution. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–10, Barcelona, 2010. IEEE.

[22] Hugues Juillé and Jordan B. Pollack. Co-evolving intertwined spirals. In *Proceedings of the Fifth Annual Conference on Evolutionary Programming*, pages 461–468, 1996.

[23] Hugues Juillé and Jordan B. Pollack. Dynamics of co-evolutionary learning. In Pattie Maes, Maja J. Mataric, Jean-Arcady Meyer, Jordan Pollack, and Stewart W. Wilson, editors, *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior: From animals to animats 4*, pages 526–534, Cape Code, USA, 9-13 1996. MIT Press.

[24] Hugues Juillé and Jordan B. Pollack. Coevolutionary learning: a case study. In *In Proceedings of the 15th International Conference on Machine Learning*, pages 251–259. Morgan Kaufmann, 1998.

[25] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[26] Christopher G. Langton. *Artificial life: An overview*. The MIT Press, 1997.

[27] Simon M. Lucas. Computational intelligence and games: Challenges and opportunities. *International Journal of Automation and Computing*, 5(1):45–57, 2008.

[28] Sean Luke and R. Paul Wiegand. When coevolutionary algorithms exhibit evolutionary dynamics. In *2002 Genetic and Evolutionary Computation Conference Workshop Program*, pages 236–241, 2002.

[29] Thomas Miconi. Why coevolution doesn't "work": Superiority and progress in coevolution. In *EuroGP*, 2009.

[30] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[31] Jason Noble and Richard A. Watson. Pareto coevolution: Using performance against coevolved opponents in a game as dimensions for pareto selection. In Lee Spector et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 493–500, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[32] Stefano Nolfi and Dario Floreano. Coevolving Predator and Prey Robots: Do Ąrms RacesĄrise in Artificial Evolution? *Artificial Life*, 4(4):311–335, 1998.

[33] Jan Paredis. Coevolving cellular automata: Be aware of the red queen. In *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 393–400, 1997.

[34] Elena Popovici, Anthony Bucci, R. Paul Wiegand, and Edwin D. de Jong. *Handbook of Natural Computing*, chapter Coevolutionary Principles. Springer-Verlag, 2011.

[35] Elena Popovici and Kenneth De Jong. Understanding competitive co-evolutionary dynamics via fitness landscapes. In *Artificial Multiagent Symposium. Part of the 2004 AAAI Fall Symposium on Artificial Intelligence*, 2004.

[36] Craig Reynolds. Competition, coevolution and the game of tag. In R. A. Brooks and P. Maes, editors, *Artificial Life IV, Proceedings of the fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 59–69, MIT, Cambridge, MA, USA, 1994. MIT Press.

[37] Chrisotpher D. Rosin and Richard K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.

[38] S.J. Russell, P. Norvig, J.F. Canny, J.M. Malik, and D.D. Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, NJ, 1995.

[39] Marcin Szubert, Wojciech Jaśkowski, and Krzysztof Krawiec. Coevolutionary temporal difference learning for othello. In *IEEE Symposium on Computational Intelligence and Games*, pages 104–111, Milano, Italy, 2009.

[40] Richard A. Watson and Jordan B. Pollack. Coevolutionary dynamics in a minimal substrate. In Lee Spector et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 702–709, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[41] Liping Yang, Houkuan Huang, and Xiaohong Yang. An efficient pareto-coevolution archive. In *Natural Computation, 2007. ICNC 2007. Third International Conference on*, volume 4, pages 484–488, Aug. 2007.