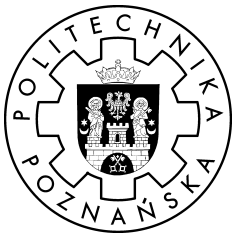
Poznań University of Technology
Institute of Computing Science



Marcin Szubert

Coevolutionary Reinforcement Learning and its Application to Othello

Master's thesis

Supervisor: dr hab. inż. Krzysztof Krawiec

Poznań, 2009

Abstract

This thesis focuses on the fields of evolutionary computation and machine learning. We present Coevolutionary Temporal Difference Learning – a novel way of hybridizing coevolutionary search with reinforcement learning that works by interlacing one-population competitive coevolution with temporal difference algorithm. Both constituent methods are capable of learning without human expertise and have shown promising results that indicate their complementary advantages. Nevertheless, their hybrids have not received much attention yet, and therefore seem to be innovative.

The coevolutionary part of our algorithm provides for exploration of the solution space, while the temporal difference learning performs its exploitation by local search. We apply the proposed method to the board game of Othello, using weighted piece counter for representing players' strategies. The formulation of Coevolutionary Temporal Difference Learning leads also to introducing Lamarckian form of coevolution, which we discuss in detail.

The results of an extensive computational experiment demonstrate that Coevolutionary Temporal Difference Learning is superior to coevolution and reinforcement learning alone, particularly when coevolution maintains an archive to guarantee historic progress. We investigate the role of the relative intensity of coevolutionary search and temporal difference search, which turns out to be an essential parameter. Additionally, we also verify how the initialization of strategies influences the quality of evolved solutions. Conclusions point to the need of further investigation of coevolutionary reinforcement learning approach. We propose a few potential directions for the future work.

Finally, in this thesis we also present design and implementation of the co-Evolutionary Computation in Java library based on ECJ – a well-known evolutionary computation framework. The developed software is easy to extend and allows flexible experiment definition.

Acknowledgements

I am very grateful for the advice and support of my supervisor Krzysztof Krawiec and his assistant Wojciech Jaśkowski, who have both shown a large interest in my work. I thank them especially for many constructive comments and careful revision of the text. Our numerous discussions have greatly improved this thesis. I wish to express my special gratitude to Wojciech Jaśkowski, for his constant inspiration, encouragement and, most importantly, useful criticism.

I would also like to thank all the people who has supported me and who has kept me going throughout the writing of this thesis.

Contents

Chapter 1
Introduction

> Those who are inspired by a model other than Nature,
> a mistress above all masters, are laboring in vain...
>
> *Leonardo di ser Piero da Vinci*

Nature has always served as a great source of inspiration for scientists and engineers. The concept of learning from nature and applying biological principles to produce novel technologies was coined *biomimetics* (also known as *bionics*) in the mid-twentieth century [Bar-Cohen 05]. However, the idea itself is much older. One of the first documented examples of biologically inspired engineering dates as far back as Leonardo da Vinci, who examined physiology of birds and fish in his famous designs of flying and swimming devices. Nowadays, biomimetics has numerous applications across variety of fields including molecular design of nano-materials, biomechanics, robotics and computer science. An impressive example, which can be viewed as a continuation of da Vinci's work on imitating flying animals, is the development of autonomous flying robots [Zufferey 08].

In the field of computer science, biology influenced the architecture of the first digital computer – its inventor, John von Neumann, used the human brain as the model for his design [Neumann 58]. Understanding how biological organisms process information and creating algorithms that exhibit human cognitive abilities became later the main goals of artificial intelligence (AI). Although these primary natural motivations have been neglected over the years, there is a branch of modern AI aimed at developing computational models that make use of biological concepts, namely, *bio-inspired artificial intelligence* [Floreano 08]. This field includes many innovative approaches, such as evolutionary computation, artificial neural networks, immune systems, biorobotics, and swarm intelligence. Despite the common origin, the key ideas behind these methods are twofold. Some of them, like for instance swarm intelligence, are examples of biomimetics, that emulate certain aspects of natural design or the functions of particular organisms or ecosystems. However, all these characteristics of biological systems, that are worth imitating, are nothing more than results of natural evolution. It is the evolutionary pressure that throughout billions of years has been forcing biological systems to be highly efficient and successful to survive in changing environments [Darwin 59]. Therefore, another modern approach

to AI is to artificially replicate the mechanisms of natural evolution. This idea has been successfully put into practice by evolutionary algorithms and robotics. In contrast to traditional biomimetic methods that rely on effects of natural evolution, simulating evolutionary process leads to alternative novel solutions that potentially have never appeared in nature.

Evolutionary algorithms are well-known for their successful results in the fields of optimization and machine learning [Goldberg 89]. Nevertheless, there is a primary, but often overlooked, difference between evolution simulated by these algorithms and evolution occurring in nature. Whereas natural evolution does not have a specified goal and is essentially an open-ended adaptation process, artificial evolution is an optimization procedure that attempts to find the best solution with respect to a predefined objective. This difference is especially consequential in domains where it is difficult to specify an objective or accurately measure its value. Using evolutionary algorithms with a hand-crafted objective function for such problems has many shortcomings. In particular, artificial evolution may not be able to match the diversity and creativity revealed by its natural counterpart. These problems were addressed by introducing coevolutionary algorithms which, in this context, can be considered as evolutionary algorithms with biomimetic fitness evaluation.

Artificial coevolution has recently became one of the most promising methods of bio-inspired AI and has been increasingly applied to many interesting machine learning tasks. Coevolutionary phenomenon of competitive arms race, that was at first observed in nature, confirms the general remark that the interplay between computer science and biology is noteworthy and has great potential. Among main applications of coevolutionary algorithms are problems with *interactive domains* [Ficici 04]. The canonical examples of such problems are board games, which are particularly appealing since they have always been a test-bed for AI. Indeed, the ability to play a board game is considered as a sign of intelligence while creating a machine capable of defeating human players has been a goal of many researchers for over half a century [Shannon 50, Samuel 59]. Although this goal was achieved by Deep Blue which in 1997 won a chess game against the world champion Garry Kasparov, the approach used by the machine in this confrontation was criticized by Noam Chomsky as being "as interesting as the fact that bulldozer can lift more than some weight lifter" [Chomsky 93]. More recently the emphasis of game-related AI research has changed to more "intelligent" methods aimed at better understanding of game characteristics instead of brute-force traversing the game tree as deep as possible. Coevolutionary learning perfectly fits in this trend. Moreover, it can be viewed as even more thought-provoking because it is able to autonomously acquire knowledge about the problem domain as it does happen in the natural human learning process. Consequently, similarly to other unsupervised approaches, it makes possible development of game-playing strategies without human expertise. This motivated us to perform experiments with coevolution and combine it with another method capable of autonomous learning, namely, temporal difference learning.

1.1 Scope and Objectives

This study focuses on the fields of evolutionary computation and machine learning. We present a novel approach for learning game-playing strategies using a combination of coevolutionary learning and temporal difference learning. Both methods are capable of learning without human expertise and have shown promising results exhibiting complementary advantages. However, their hybrids have not received much attention yet, and therefore seem to be relatively innovative. For purposes of testing our approach, we use the popular board game of Othello.

The main objective of this thesis is to verify experimentally if combining two different methods of autonomous machine learning may result in better performance than obtained by each of these methods independently. We have several additional objectives:

- Review of coevolutionary algorithms, their biological inspirations, applications in machine learning, benefits and pathologies observed in the previous research.
- Investigation of past experiments with coevolution and temporal difference methods applied to game learning problems and reported in the literature.
- Analysis of the proposed hybrid approach, its biological analogues, potential shortcomings, and similar techniques employed in the previous work.
- Design and development of a coevolutionary algorithms library *cECJ* as an extension of ECJ evolutionary computation system.
- Presenting the way of conducting experiments and measuring performance of compared methods.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 gives an overview of the concept of coevolution in nature and presents coevolutionary algorithms as an alternative to traditional evolutionary approaches. In Chapter 3 we describe the history and rules of the game of Othello, review the literature pertaining to conventional methods of learning Othello strategies, and introduce our hybrid approach drawing an analogy to Lamarckian evolution theory. Chapters 4 and 5 present design and implementation of the created software environment that was used in experiments. Chapter 6 explains the methods that have been used for conducting the experiments and reports their results. Finally, Chapter 7 summarizes the research findings, draws conclusions and proposes some directions of the future work.

Chapter 2
Coevolution

This chapter is devoted to the natural phenomenon of coevolution and its artificial counterpart in computing which has shown promising results on difficult machine learning problems. To improve intuitive understanding of the coevolution concept, let us consider the following scenario.

During annual Tour de France cycling race a single cyclist tried to escape from the peloton[1]. He was riding 100 km alone, trying very hard to defend from the chasing group, but eventually he was caught just 5 km before the finishing line. Despite the huge effort, he lost this stage. As experience shows, a single escapee has minor chances to win after riding separately for such a long time. It is worth to note that in the course of this escape, an objective measure of the cyclist's performance was available all the time – in the form of a speedometer. However, in this case, much more important was the relative speed of the cyclist with respect to peloton riders. Although he could be provided with such information, he had no way to observe it continuously (unless he was riding in the peloton).

Next day, the same cyclist decided to make one more try and started escaping. However, this time five other riders followed him immediately. They all started cooperating in the same manner as the peloton does. Indeed, riders in a group can save a lot of energy by riding near each other. A traveling peloton can be compared to birds flying in a V-shaped formation. Such cooperation helps birds, as well as cyclists, in covering long distances efficiently, because only individuals in the front are exposed to higher loads while the rest can stay out of the wind. A rotation between individuals on different positions in the formation causes that physical fatigue is spread equally among group members. Returning to the considered escapees, thanks to their cooperation they managed to keep away from the chasing peloton. However, a few kilometers before the finish, when they were already quite exhausted, the cooperation ended because each of them wanted to win individually. The last fragment of the race they were riding neck to neck. Since all riders were really close to each other, they were working even harder because they felt they could win. It is their inner competition that allowed them to avoid being caught at the last meters.

[1]Peloton is a French word for "a densely packed group of cyclists".

Although the cyclist which initiated the escape did not win, he was in the top six, that is much better result than the day before.

As we have seen above, both competition and cooperation between individuals may yield benefits. Such situation is also observed in nature but there the scope of the relationship may be broader than a single group. By definition, coevolution refers to simultaneous evolution of two or more closely related populations of different species within the same environment. Similarly to peloton riders, there is a variety of ways in which individuals interact with each other. Natural coevolution is discussed in Section 2.1, while biologically-inspired competitive and cooperative coevolutionary algorithms are presented in Section 2.2.

2.1 Coevolution in Nature

Every living organism in the natural world exists in an environment with which it interacts. Together with all other life forms in a given area it forms an *ecosystem* where each species has its own place known as *ecological niche*. The key fact is that no organism is isolated from its surroundings. Indeed, there is a variety of biological interactions that occur between individual organisms or whole species living in the same ecosystem. These interactions can be classified by effect they have on each partner. Examples of such relationships with different consequences on participating organisms include *mutualism*, *antagonism* and *commensalism*. Mutualism occurs when two or more species derive mutual profit (e.g. pollination) whereas commensalism benefits only one of them and the other is not significantly harmed (e.g. *inquilinism* – phenomenon of using a second organism for housing). In antagonistic interactions like *parasitism* or *predation* one species derives fitness benefit at the expense of another.

The second way of classifying biological interactions is more general and leads to distinguishing *competition* and *symbiosis*. Competition can occur at multiple levels of biology (i.e., intra- or inter-specific) and it is caused by limited amount of resources like food, light or space, available in the environment. Organisms must compete for access to these resources just in order to survive. Competition for survival and reproduction is considered as a major force behind Darwinian principle of *natural selection* [Darwin 59]. Competition winners have greater chances to produce offspring and pass on traits that give them advantages over their opponents. This idea was extended by Gause who formulated the law of *competitive exclusion principle* [Gause 34]. According to this law, competing species cannot stably coexist in the same ecological niche. This leads to extinction of one of the competitors or its evolutionary adaptation to different niche. Besides competition, organisms can also respond to natural selection by cooperating with each other. Such cooperation can be a result of engagement in mutually beneficial symbiosis. Margulis and Sagan claim that this kind of interaction has even greater impact on evolution than competition [Margulis 02].

Regardless of the kind and consequences, all these relationships play critical role in natural *coevolution* which is defined as "reciprocally induced evolutionary change between two or more interacting species or populations" [Price 96]. In coevolution, the reproductive success of each individual depends on the outcomes of its interactions with other evolving individuals in the same environment. The *fitness landscape* of coevolving populations is *coupled* – fitness of each organism is determined, at least partially, by coevolving organisms [Miconi 08]. In this way each species exerts selective pressure on the others and thereby influences their evolutionary development. Since natural selection promotes the fittest individuals, each species evolution is aimed at outperforming the competition or at working well in a cooperation, because such behaviors are rewarded. In the long run, this leads to successive mutual adaptations which can have diametrically different results. There are two interesting phenomena in nature related to the competitive form of coevolution, which is our main interest – *Red Queen Effect* and *Evolutionary Arms Race*. It is important to note that these phenomena have equivalents in artificial coevolution and they will be presented in Section 2.2.

2.1.1 Red Queen Effect

One consequence of coevolution is that evolutionary improvements of particular species are negated by changes worked out by coevolving parties. Adaptations which were primarily supposed to increase fitness and let species obtain reproductive success, do not necessarily have an influence on the overall situation. Although the relative fitness of coevolving individuals is not changed significantly, this evolutionary effort is still indispensable for each species just in order to avoid extinction. This situation in nature is represented by the the metaphorical Red Queen from Lewis Carroll's *"Through the Looking-Glass"* ([Carroll 87], p. 36):

> *"Well, in our country," said Alice, still panting a little, "you'd generally get to somewhere else – if you run very fast for a long time, as we've been doing." "A slow sort of country!" said the Queen. "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"*

The same principle refers to nature, where each species must constantly "run" (evolve) if it wants to keep its place in the ecosystem. The evolutionary version of this rule is known as the *Red Queen Hypothesis* proposed by Leigh Van Valen [Van Valen 73]. This hypothesis says that the sum of *absolute fitnesses* of a group of interacting species is constant. The absolute fitness can be measured not in terms of its reproductive success relative to some other organism, but in terms of energy available for expansion. Expansion, in turn, can take different forms including reproduction (as in conventional thinking of fitness) and an increase in body size.

2.1.2 Evolutionary Arms Race

Regarding the Red Queen Hypothesis, an important question arises of what are results of successive adaptations in competing populations. The idea of *evolutionary arms race* is expressed by the belief that such mutual evolutionary changes of coevolving species lead to an overall progress of involved parties. An example of the arms race is competitive coevolution of predators and preys, for instance, polar bears and seals. A new defensive trait evolved by preys (cautiousness of seals) can be compensated only by developing better offensive trait by predators (stealth of bears). Although none of the competitors become relatively more successful, this race results in absolute improvements of their behavior.

To sum up the two coevolutionary hypotheses presented above, we will refer to the famous work of Dawkins and Krebs ([Dawkins 79], p. 506):

> *As the arms race progresses and predators "improve", this does not necessarily mean that they catch more prey. The prey lineage, after all, is improving too. There seems to be no general reason to expect the average success of animals at out-running or out-witting contemporary enemies, victims, prey or competitors, to improve over evolutionary time. Van Valen has put this point more generally in his "Red Queen Hypothesis". But if modern predators are in general no better at catching modern prey than Eocene predators were at catching Eocene prey, it does at first sight seem to be an expectation of the arms race idea that modern predators might massacre Eocene prey. And Eocene predators chasing modern prey might be in the same position as a Spitfire chasing a jet.*

2.2 Coevolution in Computing

Coevolution has been introduced into the field of artificial intelligence as an alternative to traditional Evolutionary Algorithms. Thus, we need to inspect these well-established algorithms (see Section 2.2.1) before we can explore in depth artificial coevolution. In Section 2.2.2 we present an overview of coevolutionary methods and distinguish differences from ordinary Evolutionary Algorithms. Finally, in Section 2.2.3 we discuss practical pros and cons of both approaches in practice with respect to a sample problem of learning game playing.

There are two main types of coevolutionary algorithms, namely, Competitive Coevolutionary Algorithms and Cooperative Coevolutionary Algorithms (CCEA). The difference between them concerns the character of relationships between coevolving individuals (symbiotic cooperation or competition). Since we focus on the competitive form of interactions in this work, whenever the term "coevolution" is used, it shall be interpreted as competitive coevolution. The reader interested in cooperative coevolutionary algorithms is referred to the work of Potter and De Jong [Potter 00] and Wiegand *et al.* [Wiegand 01].

2.2.1 Evolutionary Algorithms

Evolutionary Algorithms (EA) are stochastic search and optimization metaheuristics which mimic the process of natural evolution including mechanisms such as genetic recombination, mutation, reproduction or selection [Bäck 97a, Bäck 97b, Eiben 03]. Although there are many different variants of EA, the common underlying ideas behind all these techniques are the same:

- EA utilize the collective learning process of a population of individuals. Each individual represents a candidate solution for a given problem, i.e., a search point in the space of potential solutions. Thus, in contrast to several other metaheuristics which process single search point at time, EA work on a set of concurrent solutions and can easily be parallelized.
- The offspring of population individuals is generated by randomized process that models natural phenomena of mutation and recombination. The former corresponds to self-replication of slightly modified individuals, while the latter exchanges information between two or more existing individuals.
- The *survival of the fittest* principle is used in order to refine the population of solutions iteratively. Fitness, which is considered as a measure of quality, is assigned to individuals by means of evaluating them in the environment. According to this value, the selection process favors better individuals to reproduce over those that are relatively worse.

A basic scheme of an evolutionary algorithm is illustrated in Algorithm 1. In the first steps, a population of candidate solutions is set up and evaluated according to the given objective function(s). If multiple such target functions are defined and all are subject to optimization then this is the case of Evolutionary Multi-Objective Optimization (EMOO) [Coello 98]. Regardless of the number of objectives, the algorithm use their values to determine an actual fitness of an individual. After the fitness is assigned, the evolutionary loop is entered and repeated until the termination condition is met. Fulfilling this condition may depend on the number of generations passed or a quality of the best individual in the population.

One round in an evolutionary loop corresponds to a single generation in natural evolution. Firstly, the selection process chooses the most promising individuals to act as parents of the next generation. Clearly, fitter individuals are picked with higher probabilities. In the reproduction step, offspring is created by exposing selected parents to genetic operators such as mutation and crossover. Depending on a replacement scheme, a new generation may be formed as a combination of the old one and its offspring (*preservative selection*) or contain only the offspring (*extinctive selection*). Algorithm 1 illustrates the second approach known also as *generational*. A newly bred population is evaluated. If termination condition is met, the evolution stops and the fittest individual is returned as a final solution.

Algorithm 1 Basic scheme of a generational evolutionary algorithm

```
1: P ← createRandomPopulation()
2: evaluatePopulation(P)
3: while ¬terminationCondition() do
4:    S ← selectParents(P)
5:    P ← recombineAndMutate(S)
6:    evaluatePopulation(P)
7: end while
8: return getFittestIndividual(P)
```

The family of EA is composed of a few methods that differ slightly in technical details, but all match the basic scheme presented in Algorithm 1. The most important difference between these methods concerns so called *representation* which defines a mapping from *phenotypes* onto a set of *genotypes* and specifies what data structures are employed in this encoding. Phenotypes are objects forming solutions to the original problem, i.e., points of the *problem space* of possible solutions. Genotypes, on the other hand, are used to denote points in the evolutionary *search space* which are subject to genetic operations. The process of genotype-phenotype decoding is intended to model natural phenomenon of *embryogenesis*. More detailed description of these terms can be found in [Weise 09].

Returning to different dialects of EA, candidate solutions are represented typically by strings over a finite (usually binary) alphabet in Genetic Algorithms (GA) [Holland 62], real-valued vectors in Evolution Strategies (ES) [Rechenberg 73], finite state machines in classical Evolutionary Programming (EP) [Fogel 95] and trees in Genetic Programming (GP) [Koza 92]. A certain representation might be preferable if it makes encoding solutions to a given problem more natural. Obviously, genetic operations of recombination and mutation must be adapted to chosen representation. For example, crossover in GP is usually based on exchanging subtrees between combined individuals.

The most significant advantage of EA lies in their flexibility and adaptability to the given task. This may be explained by their metaheuristic character of "black box" that makes only few assumptions about the underlying objective function which is the subject of optimization. EA are claimed to be robust problem solvers showing roughly good performance over a wide range of problems, as reported by Goldberg [Goldberg 89]. Especially the combination of EA with problem-specific heuristics including local-search based techniques, often make possible highly efficient optimization algorithms for many areas of application. Such hybridization of EA is getting popular due to their capabilities in handling real-world problems involving noisy environment, imprecision or uncertainty. The latest state-of-the-art methodologies in Hybrid Evolutionary Algorithms are reviewed in [Grosan 07].

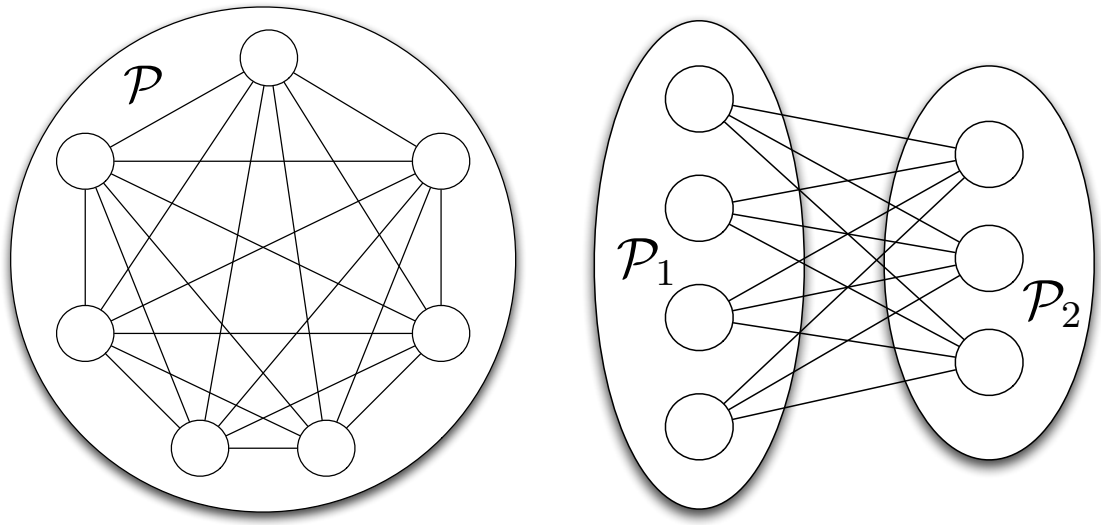
2.2.2 Coevolutionary Algorithms

Although Evolutionary Algorithms are based on biological evolutionary processes, most of these methods assume availability of pre-defined static fitness function which of course does not exist in biology. Indeed, fitness of living organisms is dependent upon dynamically changing environment in which they interact and coevolve with other organisms. Coevolutionary Algorithms (CEA) intend to model these interactions and thereby more truly imitate fitness evaluation which occurs in nature. Such evaluation may be particularly useful for problems for which an objective fitness function is unknown, not known a priori or difficult to compute.

Following above, the fundamental difference between CEA and EA concerns the process of evaluation; apart from that, both approaches can be considered quite analogous. Thus, the only thing that must be necessarily changed in Algorithm 1 to transform it to the simplest coevolutionary algorithm is the implementation of `evaluatePopulation`($\mathcal{P}$) function. In CEA, this function has access only to the outcomes of *interactions* between coevolving individuals and not to any objective fitness measure. The abstract notion of interaction denotes here a procedure that reveals information about a pair (or larger tuple) of candidate solutions. Similarly to nature, not every pair of individuals from coevolving populations is involved in a relationship. This issue is addressed in Section 2.2.2.1, where different interaction schemes are presented. A fitness of an individual is usually calculated by aggregating the outcomes of all interactions in which it participates into a single value. However, it is worth to point out that such approach has been recently criticized by many researchers including Bucci [Bucci 07], who claimed that such aggregation causes *the problem of measurement* which may be a reason of many *coevolutionary pathologies*.

Another difference between CEA and EA results from contextual sensitivity of evaluation in coevolution. The fitness assigned to an individual relies heavily on the state of still changing environment and, therefore, candidate solutions appearing well in one environment may be poor in the other. For this reason, the fitness is actually only relative, *subjective* measure of quality [Watson 01]. In CEA, this subjective fitness is directing selection process, but this do not imply increases in the *objective* quality of solutions. Consequently, guaranteeing objective progress is one of the main challenges for coevolutionary methods. There are several approaches addressing this issue including *coevolutionary archives* which are shortly discussed in Section 2.2.2.2. Note that in EA, thanks to availability of an objective evaluation function, progress can be obtained by *elitism* that preserves the best individuals.

In contrast to biology, the term *coevolution* in computing may also refer to co-adaptation of individuals within only a single population. However, it has been shown that using two populations is usually a better idea since "performing is not the same as informing" ([Bucci 07], p. 4) and, for instance, the perfect strategy for a game is not the perfect metric for an opponent's ability. This idea fits into previously proposed *host-parasite* (also known as *learner-teacher*) coevolutionary



(a) Round-robin tournament in one-population coevolution – requires $n(n-1)/2$ interactions, where $n = |\mathcal{P}|$

(b) Round-robin tournament in two-population coevolution – requires nm interactions, where $n = |\mathcal{P}_1|$, $m = |\mathcal{P}_2|$

Fig. 2.1: Round-robin tournament interaction pattern

paradigm [Pagie 97]. In this paradigm, populations of candidate solutions and tests are interacting with each other; solutions are rewarded for solving tests, whereas test are rewarded for failing solutions [Ficici 08].

2.2.2.1 Interaction Patterns

In CEA an individual is evaluated on the basis of interactions with other evolving individuals from the same or different population. At every generation in the course of evolution, the individuals in populations are paired up by some abstract pattern known also as *competition topology*. The role of this pattern is to specify the exact number and origin of the opponents for each individual. Since simulating competitions between accordingly coupled pairs is the dominant computational requirement of the evolution process, the competition topology is an important consideration. Different types of topologies were proposed and discussed by Angeline and Pollack [Angeline 93], Panait and Luke [Panait 02] and Sims [Sims 94b].

Round-Robin Tournament illustrated in Figure 2.1 is a common approach resulting in the most accurate evaluation. In this pattern each member of each population interacts with every other individual which can serve as a partner. Depending on the number of populations employed by the algorithm and their roles, the set of appropriate partners is different. Typically, in one-population coevolution all other members of the population are used as opponents (see Figure 2.1a), while in two-population coevolution – all members of the opposite population (see Figure 2.1b). The primary drawback of this topology is a substantial computational cost, which for

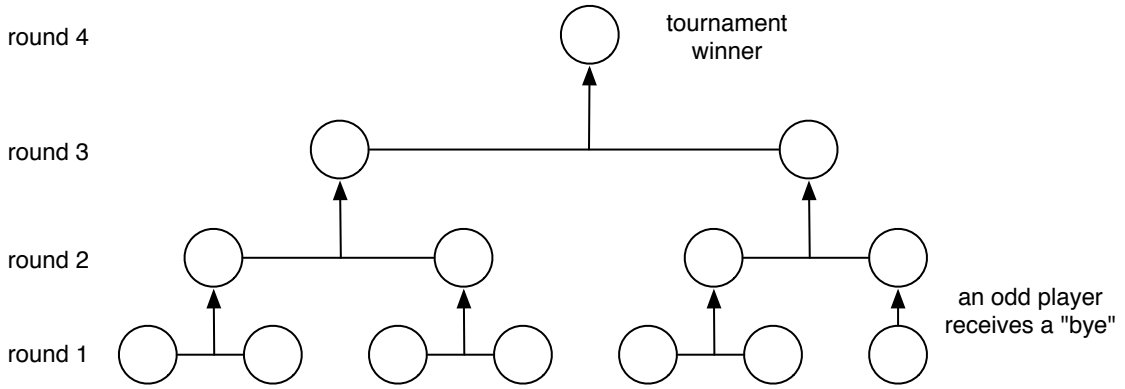


Fig. 2.2: Single Elimination Tournament

large populations can be unacceptable. Potential improvements include *K-Random Opponent* pattern, where instead of competing with all possible opponents only K of them are chosen randomly. Alternatively, to reduce the cost even more, each individual could compete only with one opponent which obtained the highest fitness in the previous generation. This is known as *Last Elite Opponent* topology.

Figure 2.2 presents *Single Elimination Tournament* (SET) which is a diametrically different type of interaction pattern. Initially, entire population is in the competition, individuals are paired at random and play one game per pair. The losers of the games are eliminated, while the winners advance to the next round of the tournament and are paired again. If a number of individuals competing in a certain round is odd, one player receives a "bye" and is promoted without playing a game. The tournament continues until a single champion remains. The fitness of an individual equals the number of the round it achieved in competition. This can be quite noisy fitness measure, because if there is no optimal solution in the population then it is possible for an average player to win the tournament. Also, this pattern does not apply easily to multi-population coevolution. However, a great advantage of this topology is that it needs only $n - 1$ competitions for a population of size n.

2.2.2.2 Coevolutionary Archives

An archive is a collection of the most promising individuals encountered by the evolutionary process. At every generation, populations are examined by the archive algorithm which selects individuals that should be retained according to implemented *solution concept*. Keeping past opponents in the archive allows to achieve *historical progress*, which is essentially a realization of the *arms race* hypothesis [Miconi 09].

Examples of archive algorithms include Hall of Fame [Rosin 97], Incremental Pareto Coevolution Archive [de Jong 04b], Layered Pareto Coevolution Archive [de Jong 04a] and MaxSolve [de Jong 05]. The role of archives is clearly identified by Rosin and Belew in their work concerning Hall of Fame ([Rosin 97], p. 6)

So, in competitive coevolution, we have two distinct reasons to save individuals. One reason is to contribute genetic material to future generations; this is important in any evolutionary algorithm. Selection serves this purpose. Elitism serves this purpose directly by making complete copies of top individuals.

The second reason to save individuals is for purposes of testing. To ensure progress, we may want to save individuals for an arbitrarily long time and continue testing against them. To this end, we introduce the 'Hall of Fame', which extends elitism in time for purposes of testing. The best individual from every generation, is retained for future testing.

2.2.3 Coevolution vs. Evolution in Practice

An important question arises: when and why shall we prefer coevolution rather than traditional evolutionary algorithm? As we mention above, EA assume that an objective evaluation function is available, what stays in conflict with natural evolution theory. It appears that this inconsistency can lead to serious difficulties if, in contrast to optimization, there is no objective function intrinsic to a given problem. This is the case in so-called *test-based problems* where the quality of a candidate solution is determined by the outcomes of a large or infinite set of tests [Bucci 04]. Examples of such problems include learning game-playing strategies and *open-ended* evolution [Sims 94a]. Since accurate and efficiently computable fitness function cannot be easily defined in such circumstances, CEA seems to naturally fit this class of problems, as we will discuss below.

For purposes of better understanding this issue, let us consider the problem of evolving chess player. Clearly, this is an example of a test-based problem, where each test is a game strategy and each interaction consists in playing a game with one opponent. Obviously, the number of tests in this case is finite but unimaginable. Moreover, it is important to note that board games in general are considered as canonical examples of *intrinsically interactive domains* which naturally require coevolution [Ficici 04] .

2.2.3.1 Using Evolutionary Algorithm for a Test-based Problem

Despite difficulties, it is still possible to apply traditional EA to test-based problems, but it requires designing a metric of quality for evolved solutions, that can be used as an objective evaluation function. However, constructing a reasonable method of rating individuals can lead to many potential pitfalls.

A quality of a particular chess strategy can be accurately evaluated by playing against all other possible strategies. In chess, as well as in most real-world test-based problems, enumerating the space of possible solutions is impossible or at least extremely expensive. The easiest way to reduce the computational complexity of the evaluation process is to choose only a subset of possible strategies as opponents.

Here, we have two possible approaches – we can randomly sample the set of all strategies only once before the algorithm starts or more than once, whenever fitness need to be evaluated. The first approach leads to optimizing performance against a fixed set of strategies which can reveal very specific style of play and thereby result in lack of responsiveness to learner's performance (i.e., regardless of learner's abilities, opponents' abilities are constant). Additionally, since this fixed set of opponents can be viewed from supervised learning perspective as a training set, there is a risk of overfitting in this case. The second approach, in turn, results in the problem of fitness inconsistency because each fitness value is highly dependent on the specific subset of strategies that happen to be chosen as opponents. Due to the large number of possible tests, selecting a subset of them which is sufficiently representative is usually impossible.

Another potential method of evaluating the quality of a chess strategy consists of competing with an expert player. If the game is deterministic, this idea surely minimizes the cost of evaluation to a single duel; otherwise, it may be needed to repeat it several times for statistical significance. It is also possible to introduce some sort of randomness into a deterministic game to obtain more robust evaluation. Nevertheless, the main disadvantage of this measure is that it relies on the knowledge of the expert strategy for a particular game and, therefore, require potentially substantial human effort to construct it. Furthermore, a perfect player is not an ideal trainer [Epstein 94] as it does not provide satisfactory *gradient*. Since a beginning learner very rarely wins with an expert, it would be hard to judge which strategies should be preferred in the evolutionary search.

A slightly different approach, applicable to many test-based problems, is based on human-designed fitness function. This method requires intimate knowledge of the problem domain and its precise articulation. For evaluating board games strategies, it would consist of characterizing different aspects of expert play and observing these characteristics in behavior of evaluated player. A useful notion here is that of *underlying objectives* of a problem [de Jong 04c]. Explicit knowledge of such objectives would be sufficient to determine the outcomes of all possible tests. Since they are usually unknown, a hand-designed function may only attempt to estimate them. The early work of Samuel [Samuel 59], though does not involve evolutionary algorithm, used a closely related idea of orthogonal set of terms – *scoring parameters*. Regarding checkers, these parameters include inability to move and piece advantage.

2.2.3.2 Benefits of Using Coevolution

Coevolution can in principle avoid all biases resulting from the use of a fixed evaluation function for a test-based problem. Regarding the evolution of a chess playing strategy, fitness evaluation is based on games between evolving players. Clearly, the set of opponents for each individual is determined by a certain interaction pattern (see Section 2.2.2.1). To apply the learner-teacher paradigm, two populations could be used, representing strategies of black and white player, respectively.

Evaluation in such environment is performed in an adaptive, dynamic manner. The tests which are used for measuring quality are identified by the search process itself. This results in greater responsiveness to learners performance and construction of learnable gradient [Viswanathan 05]. Consequently, coevolution is believed to encourage the *arms race* between coevolving populations. This belief was expressed among other potential advantages of artificial coevolution by Nolfi and Floreano in their work concerning evolutionary robotics ([Nolfi 98], p. 2):

> *First, the co-evolution of competing populations may produce increasingly complex evolving challenges. As discussed by Dawkins and Krebs, competing populations may reciprocally drive one another to increasing level of complexity by producing an evolutionary "arms race". (...) it is like producing a pedagogical series of challenges that gradually increase the complexity of the corresponding solutions. (...)*
>
> *Secondly, because the performance of the individual in a population depends also on the individual strategies of the other population which vary during the evolutionary process, the ability for which individuals are selected is more general (i.e., it has to cope with a variety of different cases) than in the case of an evolutionary process in which co-evolution is not involved. (...)*
>
> *Finally, competing co-evolutionary systems are appealing because the everchanging fitness landscape, due to changes in the co-evolving species is potentially useful in preventing stagnation in local minima.*

2.2.3.3 Coevolutionary Pathologies

Apart from hypothetical benefits supported by successful applications [Hillis 92, Sims 94a, Miller 94, Juillé 96], coevolutionary algorithms are also reported to demonstrate a variety of pathological behaviors including *disengagement*, *cyclic dynamics* and *evolutionary forgetting* [Paredis 97, Ficici 98, Watson 01]. Although a lot of algorithmic remedies were proposed to alleviate these pathologies, a considerable amount of work has been recently focused on identifying their common reason. As a result, it has been found that among theoretical causes of such misbehaviors are lack of rigor in implementing the desired *solution concept* [Ficici 04] and the problem of *aggregate measurement* [Bucci 07]. The most often encountered pathologies are shortly described below.

Disengagement occurs when coevolving populations represent diametrically different levels of performance. In our example, if none of white chess player can win with any of black players then all of the white players will appear equally poor and all of the black players equally superior. In such situation, all evolving individuals in each population obtain the same fitness and cannot be distinguished regarding their level of play. Consequently, this leads to a loss of fitness gradient.

Cyclic dynamics, also known as *Red Queen dynamics* or *mediocre stable states*, is a problem caused by intransitivity in the problem domain. The best known example of a problem with such property is the game of Rock-Paper-Scissors. The phenomenon of cyclic dynamics occurs when individuals continuously change (possibly improving their relative fitness), but without making overall objective progress. This corresponds to getting stuck in a local optimum of the coevolutionary search.

Evolutionary forgetting can be a result of some of the misbehaviors described above. This pathology refers to the situation when "previously acquired and subsequently discarded trait is once again desired" [Ficici 04]. In this context, a "trait" represents any measurable aspect of behavior.

Chapter 3

Othello and Coevolutionary Reinforcement Learning

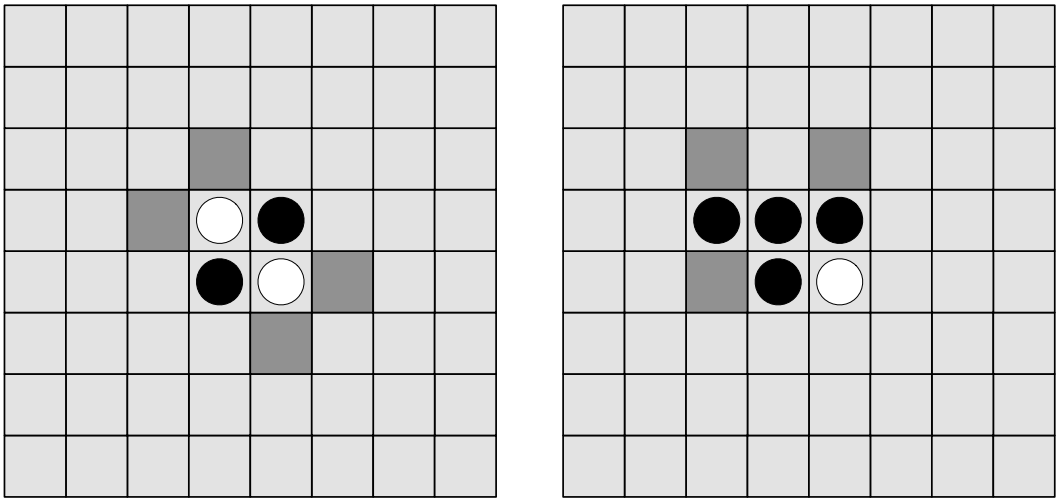
In this chapter we describe the game of Othello and discuss the methods of learning game-playing strategy. After introducing the game rules and its historical background, we present the canonical strategy representation of weighted piece counter (WPC). Next, we try to answer the question of how to produce a competitive Othello player by means of machine learning. We focus on the approaches which can be considered the most intelligent, as they are able to produce nontrivial strategies starting with zero knowledge. These learning algorithms differ significantly from the methods developed in the early years of games-related AI research. For this reason, we present also previous work on Othello-playing computer programs and discuss how the emphasis of the research has changed throughout the years. Among examples of modern approaches to learning Othello strategies are Temporal Difference Learning (TDL) and Coevolutionary Learning (CEL). In Section 3.2 we review the literature in order to describe how these methods can be applied for this problem. Finally, in Section 3.3 we introduce a hybrid approach of Coevolutionary Temporal Difference Learning and draw an analogy to Lamarckian evolution theory.

3.1 Othello

A motto for the game of Othello says:

A minute to learn . . . a lifetime to master

Indeed, Othello is a very simple classic board game. However, despite the apparent simplicity, Othello is at the same time one of the most challenging games with numerous tournaments and regular world championship matches. The exact origin of the game is unknown but rumors say that it arose from an old Chinese game called *Fan Mian*. Certainly, Othello is based upon two English games marketed in 19th century – *Annexation* and *Reversi* designed by John W. Mollet and Lewis Waterman, respectively [Parlett 99].



(a) Starting board configuration **(b)** Board state after black player's move

Fig. 3.1: Examples of Othello board configurations

The modern rules of Othello, which are formally adopted around the world now, were invented by Japanese salesman Goro Hasegawa in 1971. His father named the game after the classical play written by William Shakespeare (*"Othello, the Moor of Venice"*) to remark that the game was full of dramatic reversals caused by rapid changes in dominance on the board. Othello had very quickly gained considerable popularity in Japan and it started to spread to Europe and North America.

3.1.1 Game Rules

The game of Othello is played by two players on an 8×8 board. Typically, pieces are disks with a white and black face, each side representing one player. Figure 3.1a shows the initial state of the board – each player starts with two stones in the middle of the grid. The black player moves first and has four possible moves which are illustrated as shaded locations. Players make moves alternately until the board is completely filled or until neither player is able to move. The goal of the game is to have a majority of squares occupied by own pieces. If both players have the same number of disks then the game is a draw.

A legal move consists of placing a piece on an empty square and flipping appropriate opponent's pieces to the color of the current player. To place a new piece on a certain square, two following conditions must be fulfilled. Firstly, a position of the piece must be adjacent to position occupied by opponent's piece. Secondly, at least one straight line must exist between the new piece and some other piece of the current player, with a contiguous sequence of opponent's pieces in between. This line can be formed vertically, horizontally, or diagonally. After placing the piece,

all opponent's pieces lying on such lines are flipped. If multiple lines exist, flipping affect all of them. This makes the game particularly "dramatic" – in a single move a player may gain a large number of pieces and swap players' chances of winning. A legal move requires flipping of at least one of the opponent's pieces. If a player has no valid moves then he passes his turn and his opponent plays a second turn in a row. However, if a player has one or more legal moves available he may not pass his turn. An exemplary move of the black player and the resulting board state are presented in Figure 3.1b.

3.1.2 Strategy Representation

One of the main issues to consider when learning game strategy is the architecture of the learner, which is to a great extent determined by the strategy representation. There is a multitude of reasonable strategy representations; here we rely on a heuristic assumption that to judge the utility of a particular board state it is enough to *independently* consider the occupancy of all board locations. This principle is implemented by the position-weighted piece counter (WPC) representation which was commonly used in previous research [Lucas 06, Runarsson 07, Kim 07a]. The WPC is a linear weighted evaluation function that assigns a weight w_i to each board location i and uses scalar product to calculate the utility f of board state $\mathbf{b}$:

$$f(\mathbf{b}) = \sum_{i=1}^{8\times 8} w_i b_i. \tag{3.1}$$

The input values b_i depends on the occupancy of particular board locations, being 0 for empty location, +1 for black piece and -1 for white piece. The output of this function indicates how favorable is a given board state $\mathbf{b}$ for a certain player. The players interpret this value in a complementary manner: the black player prefers moves leading to states with larger values, while smaller values are favored by the white player.

The main advantage of WPC is its simplicity resulting in a very fast evaluation. Regarding the game of Othello, only 64 weights need to be learned. Moreover, a strategy represented by WPC can be often easily understood just by inspecting the weight values. For example, Figure 3.2 shows the weights matrix of an exemplary heuristic player. As it can be seen, this player focuses at taking possession of the corners because they are given the highest values. Conversely, locations adjacent to the corners are avoided by this strategy.

Alternatively, this type of representation can be considered as a simple artificial neural network composed of a single linear neuron connected to 64 inputs, without any hidden layers. Since a direct coding scheme is employed, exactly one input for each board position exists.

```
 1.00 -0.25 0.10 0.05 0.05 0.10 -0.25  1.00
-0.25 -0.25 0.01 0.01 0.01 0.01 -0.25 -0.25
 0.10  0.01 0.05 0.02 0.02 0.05  0.01  0.10
 0.05  0.01 0.02 0.01 0.01 0.02  0.01  0.05
 0.05  0.01 0.02 0.01 0.01 0.02  0.01  0.05
 0.10  0.01 0.05 0.02 0.02 0.05  0.01  0.10
-0.25 -0.25 0.01 0.01 0.01 0.01 -0.25 -0.25
 1.00 -0.25 0.10 0.05 0.05 0.10 -0.25  1.00
```

Fig. 3.2: The heuristic player's strategy represented by WPC

3.1.3 Previous Research

The game of Othello has been a subject of computational intelligence research for more than 20 years. A significant interest in this game may be explained by its large state space complexity and high divergence rate causing that it remains unsolved – a perfect Othello player has not been developed yet.

Conventional Othello-playing programs are based on a thorough human analysis of the game leading to sophisticated hand-crafted evaluation functions. These programs often incorporate supervised learning techniques with use of large expert-labeled game databases and efficient look-ahead game tree search. One of the first examples representing such approach was BILL [Lee 90]. Besides using pre-computed tables of board patterns, it employed Bayesian learning to build in so called *features* into evaluation function estimating the probability of winning. Today, one of the strongest Othello programs is Logistello [Buro 95, Buro 02] which also makes use of advanced search techniques but also applies a several new approaches for the construction of evaluation features and learning from previous games. Nevertheless, it still relies on the powerful hardware which is one of the main factors that allowed Logistello to beat the world champion Takeshi Murakami in 1997 [Buro 97].

Recently, the emphasis of the research has moved towards better understanding of which types of learning algorithms and player architectures work best. This is the aim of CEC Othello Competitions[1] where the ply depth is limited to one to eliminate previous approaches based upon brute-force game tree search. Although the standard WPC representation is among strategy representations acceptable by competition rules, all the best players were based on more complex architectures involving number of parameters. Examples of such architectures are: symmetric n-tuple network, multi-layer perceptron (MLP) and spatial MLP.

Another interesting issue is learning without any reference to human knowledge or game strategy given a priori. This task formulation is addressed by methods such as Temporal Difference Learning (TDL) and Coevolutionary Learning (CEL) which were investigated in the context of Othello by Lucas and Runarsson [Lucas 06]. That study inspired our research and will be presented in the following section.

[1] http://algoval.essex.ac.uk:8080/othello/html/Othello.html

3.2 Conventional Learning Methods

Most machine learning systems require a considerable amount of human knowledge, introduced by the designer, in order to succeed. This knowledge, which is a form of *inductive bias* [Mitchell 97], may be expressed by an appropriate representation, specific search operators, or a fixed training environment which provides a good learning gradient. However, when there is a little or no prior understanding of the problem domain, such techniques of bias engineering become impractical. This issue can be addressed by approaches that are able to learn from scratch, without any human expertise. Examples of such methods are Coevolutionary Learning (CEL) and Temporal Difference Learning (TDL).

In this section we discuss theoretical foundations of CEL (Section 3.2.1) and TDL (Section 3.2.2) and their application to learning game-playing strategies. Throughout the following discussion, we will refer to works of Lucas and Runarsson [Lucas 06, Runarsson 05], where the WPC strategy representation is employed to acquire position evaluation functions for the games of Othello and small-board Go. By reviewing the experiments reported in these publications, we show how TDL and CEL can be applied in practice.

3.2.1 Coevolutionary Learning

Coevolutionary learning (CEL) is defined as a search procedure that involves a population of learners coevolving with the learning environment such that continuous progress results from this interaction [Juillé 98]. As a result, it is possible to start with a basic learning environment, created without expert knowledge, because it gets more challenging as soon as learners improve themselves. Coevolutionary learning is considered as having great potential because of the *competitive arms race* (see Section 2.1.2) which can provide a properly constructed learning gradient.

Regarding the task of learning game-playing strategy, CEL typically starts with generating a random initial population of player individuals. Population members play games with each other, and the results of these confrontations determine the fitness which is assigned to each player. The best strategies are selected, undergo genetic modifications such as mutation or crossover, and its offspring replace former individuals. Depending on a particular replacement scheme, either the whole population or only its part is replaced. Although this general scheme seems straightforward and easy to apply in practice, there is still many design choices to be made. Some of these choices concern the overall evolutionary computation domain (population size, variation operators, selection scheme, etc.) while the other are related particularly to coevolutionary learning (interaction pattern, fitness aggregation method, etc.). Certainly, CEL embraces a broad class of algorithms, some of which we shortly review in following.

In their influential study on CEL in games, Pollack and Blair [Pollack 98] used the simplest evolutionary algorithm – a random hill-climber – to successfully address the problem of learning backgammon strategy. The algorithm is essentially a basic Evolution Strategy operating on a population of size two (i.e., (1 + 1) ES). Similar approaches of $(1 + \lambda)$ and $(1, \lambda)$ ES were reused in the referred work of Lucas and Runarsson [Lucas 06] to learn the Othello strategy. An important design choice adapted from the previous research was the geometrical parent-child averaging. Instead of replacing a parent by the best of the new offspring, the parent's strategy is only adjusted towards the child strategy by linear combination. Moreover, usage of self-adapting mutation strength was considered, but eventually it was utilized only in the context of evolving small-board Go players [Runarsson 05].

Various forms of CEL have been successfully applied to many two-person games including Blackjack [Caverlee 00], Checkers [Fogel 02], NERO [Stanley 05], Pong [Monroy 06], AntWars [Jaśkowski 07] and a small version of Go [Lubberts 01].

3.2.2 Temporal Difference Learning

Temporal Difference Learning (TDL) is a prediction method proposed by Sutton [Sutton 88] which has become a popular approach for solving reinforcement learning tasks. One of the most spectacular successes of temporal difference learning in game playing is undoubtedly Tesauro's TD-Gammon [Tesauro 95]. This influential work has triggered off a lot of research in reinforcement learning and TD methods. Among successful application of TDL to Othello are [Leouski 95, Manning 07]. Before we explore details of applying this method to learning game-playing strategies, we need to introduce reinforcement learning paradigm.

3.2.2.1 Reinforcement Learning Paradigm

Reinforcement learning (RL) is a machine learning paradigm focused on solving a general class of *sequential decision tasks*. In this specific genre of learning problems, an *agent* (learner) dynamically interacts with an unknown *environment* by making *decisions* at discrete time steps. At each time step t, the agent receives current *state* of the environment, $s_t \in \mathcal{S}$, and on that basis selects an *action* a_t from the set of available actions in this state $\mathcal{A}(s_t)$. As a consequence, the system enters a new state s_{t+1} and the agent receives a real-valued *reward* r_{t+1} which indicates desirability of the last decision and the resulting state. The objective is to find an optimal *decision policy* which describes a mapping from system *states* to probabilities of selecting each possible action. Optimal strategy maximizes the cumulative total reward of a decision sequence starting from randomly chosen state, discounted over time with a factor $\gamma \in [0, 1]$. This goal is specified by the *expected reward*:

$$E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]. \tag{3.2}$$

Reinforcement learning, in contrast to supervised learning, do not need a human supervision or any examples of correct behavior. The learner is not told how to respond to given situation, but instead it must discover which actions yield the most reward by trying them. Through such *trial and error* search the agent gathers experience about possible system states, actions, transitions and rewards. However, very rarely a learner receives exact information about its performance directly after each action – usually such information is delayed. For example, in the game of chess, it is difficult to deduce how single decisions influence the overall game situation, and thereby, the learner may not receive any rewards until the end of the game. Since actions may affect all subsequent rewards, there is a problem known as *temporal credit assignment*. The agent must determine which actions are to be credited with producing the eventual rewards. Another difference from supervised learning is that the evaluation of the system is often concurrent with learning – there is no separate training phase. As a result at the beginning of the learning process the agent has to operate despite significant uncertainty about the environment it faces.

The characteristic feature of reinforcement learning is the trade-off between *exploration* and *exploitation*. On the one hand, to obtain a lot of reward, the agent must favor exploiting states and actions that it has tried in the past and already learned that they yield high reward. On the other hand, the only way to discover such actions, that are effective in producing a reward, is to explore unknown states and actions. Thus, in order to achieve successful learning, the agent must keep a proper balance between these two experimentation strategies.

Due to flexibility of the problem specification, potential applications of reinforcement learning are numerous. RL methods are especially useful when precise domain knowledge is not available or is costly to obtain. Examples include controlling mobile robot, optimizing factories operation and playing board games. The reader interested in more detailed description of reinforcement learning paradigm is referred to the works of Sutton and Barto [Sutton 98], Moriarty *et al.* [Moriarty 97], and Kaelbling *et al.* [Kaelbling 96].

3.2.2.2 Temporal Difference Method

The family of $TD(\lambda)$ learning procedures is dedicated for prediction learning problems – that is for estimating the future behavior of an incompletely known system using the past experience. The basic idea is that learning occurs whenever system's state changes over time and it is based on the error between temporally successive predictions. The goal of learning is to make the preceding prediction to more closely match the current prediction (taking into account distinct system states observed in the corresponding time steps). In general, prediction at a certain time step can be considered as a function of two arguments: the outcome of system observation and the vector of modifiable weights.

The *TD* algorithm is expressed by the following weight update rule:

$$\Delta \mathbf{w}_t = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k, \tag{3.3}$$

where λ is the learning rate, P_t is the prediction at time t, and the gradient $\nabla_w P_t$ is the vector of partial derivatives of P_t with respect to each weight. This general formulation of TD takes into account the entire history of the learning process; in the case of $TD(0)$, the weight update is determined only by its effect on the most recent prediction P_t:

$$\Delta \mathbf{w}_t = \alpha(P_{t+1} - P_t) \nabla_w P_t. \tag{3.4}$$

Regarding the problem of learning Othello strategy represented by a WPC, P_t estimates the chances of winning from the game state $\mathbf{b}_t$ at time t. The WPC function f computes the dot product of the board state vector $\mathbf{b}_t$ and the weight vector $\mathbf{w}$ (see equation 3.1), and the obtained value is subsequently mapped to a closed interval [-1, 1] using hyperbolic tangent, so that P_t has the form:

$$P_t = tanh(f(\mathbf{b}_t)) = \frac{2}{exp(-2f(\mathbf{b}_t)) + 1} - 1 \tag{3.5}$$

By applying (3.5) to the $TD(0)$ update rule (3.4) and calculating the gradient, we get the change of weight w_i at the time step t:

$$\Delta w_{it} = \alpha(P_{t+1} - P_t)(1 - P_t^2) b_i \tag{3.6}$$

If the state observed at time $t+1$ is a terminal, then the exact outcome of the game is known and the final result can be used instead of the prediction P_{t+1}. The result value is: +1 if the winner is Black, -1 if White, and 0 when the game ends in a draw.

The process of learning consists of applying above formula to the weights of WPC vector after each move. The training data (i.e., collection of games) according to which the presented algorithm can proceed, may be obtained for example by self-play. This is a popular technique whose major advantage is that it does not need anything besides the learning system itself. During game play, moves are selected on the basis of the most recent evaluation function.

Othello is a deterministic game, thus the outcome and the course of the game between certain players is always the same. This is unfavorable, because it reduces the number of game trees to be explored. To remedy this situation, at each turn, a random move is forced with certain probability. After such a random move, no weight update occurs.

3.3 Hybrid Coevolutionary Algorithms

The past results of learning WPC strategies of Othello [Lucas 06] and small-board Go [Runarsson 05] demonstrate that TDL and CEL exhibit complementary features. The main observation is that TDL learns much faster but it fails to converge to its best results. Moreover, the best performance of TDL player is achieved after just a several hundred of games and later it stays at the same level or even degrades despite processing millions of training games. This is in sharp contrast with CEL which progresses slower, but, if properly tuned, eventually outperforms TDL.

Therefore, it sounds reasonable to combine these approaches into one hybrid algorithm exploiting advantages revealed independently by each method. To benefit from the complementary advantages of TDL and CEL we propose a method termed *Coevolutionary Temporal Difference Learning*. We present this method in Section 3.3.1. We also review previous approaches basing on similar ideas in Section 3.3.2. It is interesting to note that, in general, such approach to machine learning can be considered as a counterpart of Lamarckian evolution theory in nature, which we briefly describe in Section 3.3.3.

3.3.1 Coevolutionary Temporal Difference Learning

Coevolutionary Temporal Difference Learning (CTDL) maintains a population of players and alternately performs temporal difference learning and coevolutionary learning. In the TD phase, each player is subject to a self-play $TD(0)$. Then, in the CEL phase, individuals take part in a round-robin tournament and each of them receives a (subjective) fitness. Finally, a new generation of individuals is obtained using standard selection and variation operators and the cycle repeats.

However, there is one potential pitfall of the idea presented above resulting from different aims of both learning algorithms. TD methods attempt to learn the real utility function describing chances of winning from each state, whereas CEL strive only to find a relative ordering on the set of these states. Following above, both methods are trying to solve a bit different problems [Moriarty 97]. Consequently, the simplest hybridization may lead to a clash since each method will guide the learning process in a different direction. An important consideration in this situation concerns a proper balance between number of games played in each phase. Intuitively, it might seem that it should be equal to give each method fair chances. Note, however, that CEL uses only the information about the final game result while TDL exploit precisely every piece of knowledge contained in a single move. Hence, the CEL is expected to learn much faster and, therefore, the number of games played by this method should be potentially much smaller. This issue will be thoroughly investigated in Chapter 6.

3.3.2 Other Hybrid Approaches

Although temporal difference and widely considered evolutionary computation are known for good results in learning games strategies, their hybrids have been occasionally considered in past. Kim *et al.* [Kim 07b] trained a population of neural networks with TD(0) and used the resulting strategies as an input for the standard genetic algorithm with mutation as the only variation operator. This method was based on the winning entry of CEC 2006 Othello Competition.

In another work, Singer has shown [Singer 01] that reinforcement learning may be superior to random mutation as an exploration mechanism. He applied his concept to produce Othello-playing strategies represented by 3-layer neural networks. Similarly to our approach, the training procedure consists of successive learning phases followed by evolutionary phases. In the learning phase, a round robin tournament is played 200 times with different learning rates. During tournament games, Othello players modify the weights of their neural networks with every single move according to the standard backpropagation algorithm. The tournament results in this phase are discarded. In the evolutionary phase only one round-robin tournament is arranged and its results are used to determine a fitness of each player. The key idea of Singer's genetic algorithm is the concept of feature-level crossover. Mutation as a variation operator plays a minor role due to very low mutation probability. The experiment with this method was claimed to be moderately successful; it yielded a program that is competitive with an intermediate level human-designed Othello-player. Nevertheless, the author have not made any comparison with preexisting methods. It seems that the emphasis of his experiment has been put mainly on reinforcement learning since the number of games played in a single learning phase is much larger than in evolutionary phase.

3.3.3 Lamarckian Coevolution Perspective

The idea introduced above can be considered as a form of coevolutionary *Memetic Algorithm*. Memetic Algorithms are hybrid approaches coupling a population-based global search method with individual learning procedures capable of performing local improvements. Since these algorithms usually employ evolutionary search, they are often referred to as *Lamarckian Evolution* or *Genetic Local Search*. Before we describe technical details, we need to introduce Lamarckian evolution theory.

Jean-Baptiste de Lamarck published his theory of evolution in 1801, almost 60 years before Darwin's seminal work *On the Origin of Species* appeared. In contrast to well-known principles of natural selection, Lamarck stated that evolution is directed by individuals adapting to their environments over the course of their lifetime and passing these adaptations on to offspring. This can be interpreted as a concept of the inheritance of acquired traits [Burkhardt 77]. Although this *theory of adaptation* was finally discredited in the field of biology, it was successfully implemented within evolutionary computation systems.

Lamarckian Genetic Algorithms (LGA) alternately perform evolutionary search for the population and local search for individual solutions. Heuristic local search can be viewed as analogous to learning that occurs during an organism's lifetime. Such algorithms are considered very fast and outperform pure evolutionary approaches on many problems [Dozier 98, Katayama 00]. However, besides reporting good results it was also observed that Lamarckian evolution may encourage premature convergence to local optima [Whitley 94]. An interesting remedy to this problem is incorporating so called *Baldwin Effect* [Baldwin 96], which is consistent with Darwinian theory and regarded as real phenomenon occurring in nature. The underlying idea is that individuals can learn but without altering their genotypes. Thus, the only effect of local refinements is changing the fitness landscape – selection will favor individuals with increased capacity for learning new skills.

In this context, our Coevolutionary Temporal Difference Learning is an example of what might be termed *Lamarckian Coevolution* or *Lamarckian Coevolutionary Algorithm*. The reinforcement learning phase can be treated as a form of local search technique improving individuals, especially as $TD(0)$, which we used, is actually a gradient-descent method. We find this observation very interesting, as it enables us to perform local search in coevolution, where the fitness function is not given explicitly and implementing typical local search is nontrivial. Note, however, that this technique cannot be used for all coevolutionary problems, but only for these where Temporal Difference Learning is applicable. Moreover, we demonstrated only how to use this technique in single-population coevolution. Using the same idea for two-population coevolution is a promising further research idea, because it would open the possibility to use more advanced archive methods such as LAPCA and IPCA [de Jong 07] that could be beneficial.

Similarly, to Lamarckian Genetic Algorithms, in our Lamarckian Coevolution, the most important settings are the intensity and frequency of individual learning. These parameters define the inherent tradeoff between exploration and exploitation. If the reinforcement learning stage (responsible for exploitation) is too intensive then it can lead to rapid convergence to a local optimum and the coevolutionary stage (performing exploration) would be unable to jump out of this optimum in this case. There are several interesting directions of future work concerning the issue of Lamarckian Coevolution. For example, one appealing possibility concerns investigating the influence of including the Baldwin effect into Coevolutionary Temporal Difference Learning method.

Chapter 4
cECJ Design

This chapter presents an overview of *cECJ – co-Evolutionary Computation in Java*. cECJ is a coevolutionary algorithms library built upon ECJ, a well-known freeware evolutionary computation research system in Java created at George Mason University by Sean Luke *et al.* [Luke 08]. Although there is a `coevolve` package included in the standard ECJ distribution, it provides only basic coevolutionary methods and it is not prepared to be extended. Moreover, this module had been designed before the modern archive-based coevolutionary algorithms were developed. Because it would be quite difficult to reuse this implementation, building a completely new software package was required.

4.1 ECJ Overview

ECJ system supports a variety of evolutionary computation techniques including genetic algorithms, genetic programming, evolutionary strategies and differential evolution. Moreover, independent contributors provide also a few extensions of the system such as cartesian genetic programming or gene expression programming. Another extension is available for integrating ECJ with DREAM project (Distributed Resource Evolutionary Algorithm Machine). The system itself has a few interesting features like GUI with charting possibilities, platform-independent checkpointing and distributing the computations among multiple threads or multiple machines.

In the following sections we will briefly describe the most important parts of ECJ, whose implementation influenced the way we designed our coevolutionary extension. We will refer to the latest release of ECJ, which, at the time of writing these words, has number 18.

4.1.1 Evolutionary Process within ECJ

A high-level evolutionary process is defined in `ec.EvolutionState` class which holds a complete state of evolution at any time. Figure 4.1 depicts how the main evolutionary loop proceeds and which ECJ classes are responsible for particular stages

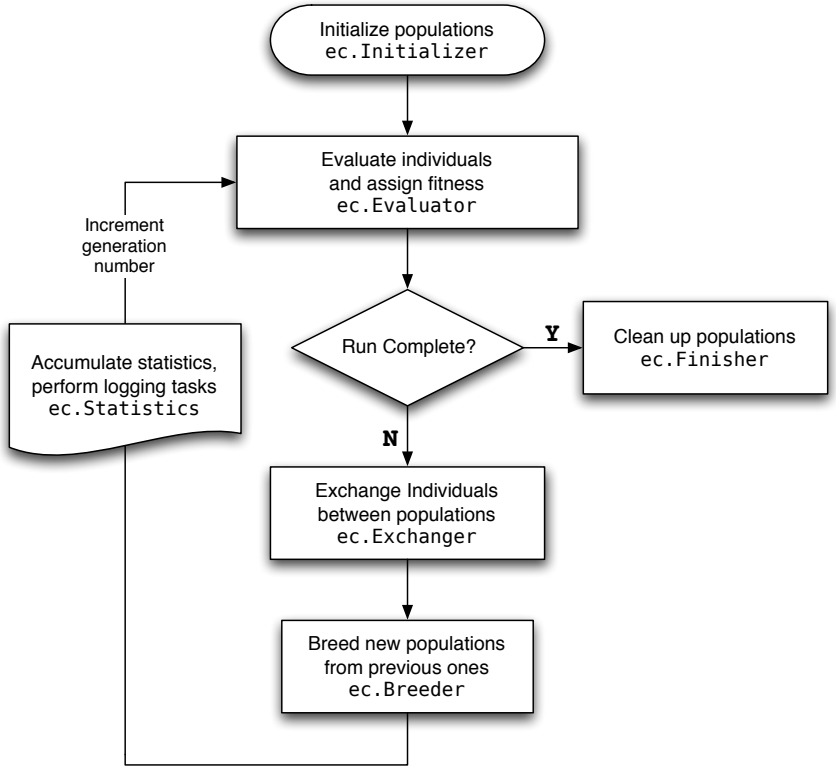


Fig. 4.1: Evolutionary process in ECJ

of evolution. However, most of the classes presented in figure are abstract and only define their functions but do not implement them. Basic implementations of these methods providing simple, non-coevolutionary, generational evolution are contained in `ec.simple` package.

4.1.2 ECJ Class Diagram

A simplified class diagram of the main classes in the ECJ system is presented in Figure 4.2. Note that the colors on this diagram represent implemented interfaces as it is explained in the legend. An important fact to observe is that besides instances of the core `Singleton` classes, the `EvolutionState` contains also a single `Population` object and a few utilities which are described in Section 4.1.4.

A population is initialized by the `Initializer` and it consists of one or more `Subpopulation` objects which store evolving `Individual`s and an information about `Species` to which they belong. An individual represents a solution to the given problem while its species defines features of the individual – specifies how it breeds (`BreedingPipeline` class) and what type of measure can be used to compare its quality (`Fitness` class).

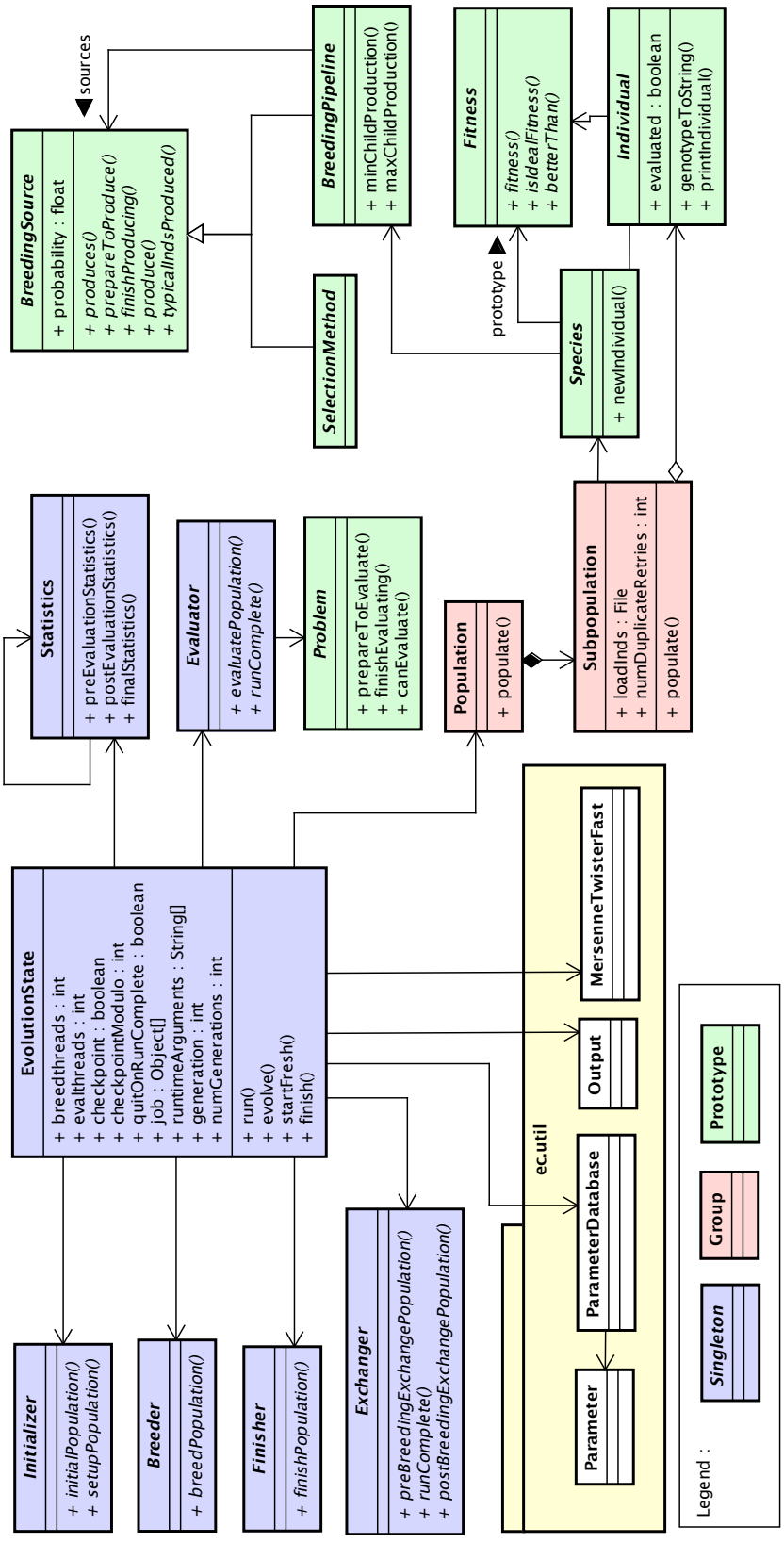


Fig. 4.2: ECJ simplified class diagram – only the core classes and their most important methods are shown; arguments and return types are not illustrated.

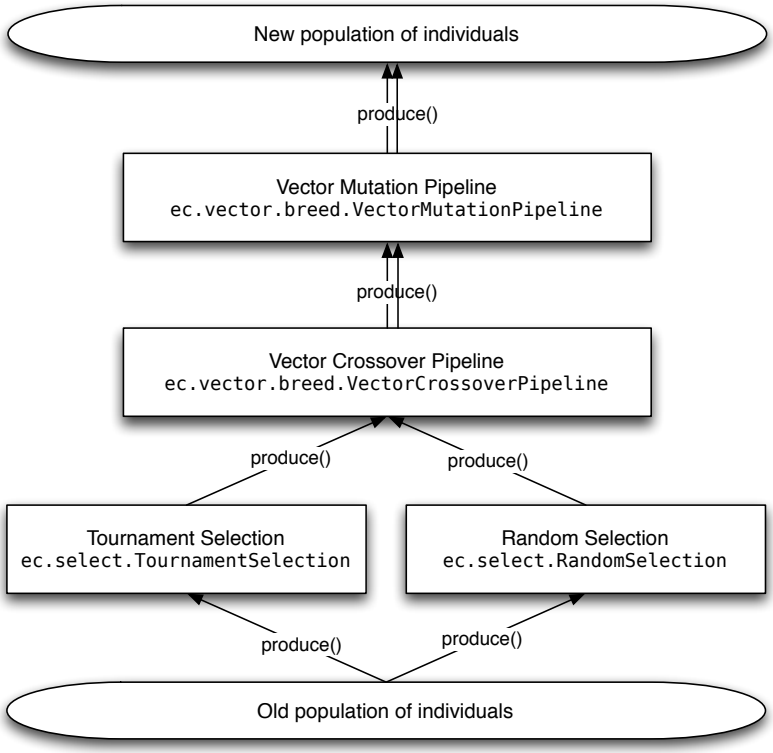


Fig. 4.3: ECJ breeding tree mechanism – example of usage

4.1.3 Breeding Mechanism

The breeding mechanism is one of the most important parts of the ECJ system, that is responsible for making the actual evolutionary change by creating a new population on the basis of the previous one. In evolutionary algorithms breeding is realized by selecting the most promising individuals and exposing them to genetic operations characteristic to a certain species. To achieve this goal in ECJ, a proper `SelectionMethod` should be used as a `BreedingSource` producing individuals for another `BreedingPipeline` realizing particular genetic operations.

It is worth pointing out that ECJ's breeding classes implement the Composite design pattern [Gamma 95], whose intent is to arrange objects into tree-like structures. In this case the leaves of the tree are `SelectionMethod`s responsible for picking individuals from the old population while the branch nodes are `BreedingPipeline`s operating on individuals gathered from its subtrees and passing it further to their parent nodes. The `Species` object of each subpopulation knows only the root of this breeding tree but it is sufficient to propagate individuals through the whole structure by recursively invoking `produce` method on children nodes. A sample breeding pipeline tree is presented in Figure 4.3. After picking two individuals from the old population using different selection methods, they are crossed over, mutated and finally placed in the new population. Note that this is only an example of using breeding mechanism and in practice the pipeline tree may take a variety of forms.

4.1.4 ECJ Utilities

Among the utilities contained in `ec.util` package the major one is undoubtedly the `ParameterDatabase` class. This class is responsible for reading and managing runtime parameters from parameter files provided by users as a command line argument to the application. Besides simple parameters of primitive types such as a number of generations, parameter file can provide names of concrete classes to be instantiated at runtime, during the course of evolution. This can be achieved thanks to reflection feature of Java language. A format of parameter file is very similar to Java properties style. Sample parameter files can be found in the appendix B.

The `ec.util` package contains also a high-quality pseudorandom number generator (an implementation of the Mersenne Twister) and output logging facilities, which are all held by the singleton `EvolutionState` object. Since it is passed as an argument to the majority of class methods of the whole system, generating pseudorandoms and message logging can be performed almost everywhere. Finally, there is also a checkpointing utility which allows for writing current evolution state to a file and restoring it later. This feature can be used in order to increase the fault-tolerance of time-consuming computation tasks.

4.2 cECJ Extensions

As it was mentioned in Chapter 2, coevolutionary algorithms differ from standard evolutionary ones in at least a few aspects. The most important difference concerns the way in which population members are evaluated. The other dissimilarities are connected with using archives as a remedy to some coevolutionary pathologies. Archives can influence the way the evaluation and breeding are performed. Moreover, the main role of archives is to represent gathered knowledge and approximate desired solution concept. All these differences form a starting point for the core functionality of cECJ extension.

4.2.1 Extended Evaluation

Evolutionary algorithms use a problem-dependent evaluation function which is capable of "objectively" measuring the quality of candidate solutions of given problem. In coevolution, on the other hand, individuals representing solutions interact with each other and according to results of these interactions the final fitness is calculated. The scope of such interactions can be inter- or intra-specific and inter- or intra-population, i.e., interaction partners can be chosen from the same or other population. In the latter case the species of the other population can be different from the species of evaluated individual.

Figure 4.4 presents the way coevolutionary evaluation extends the evaluation stage from the conventional evolutionary loop (cf. workflow diagram in Figure 4.1).

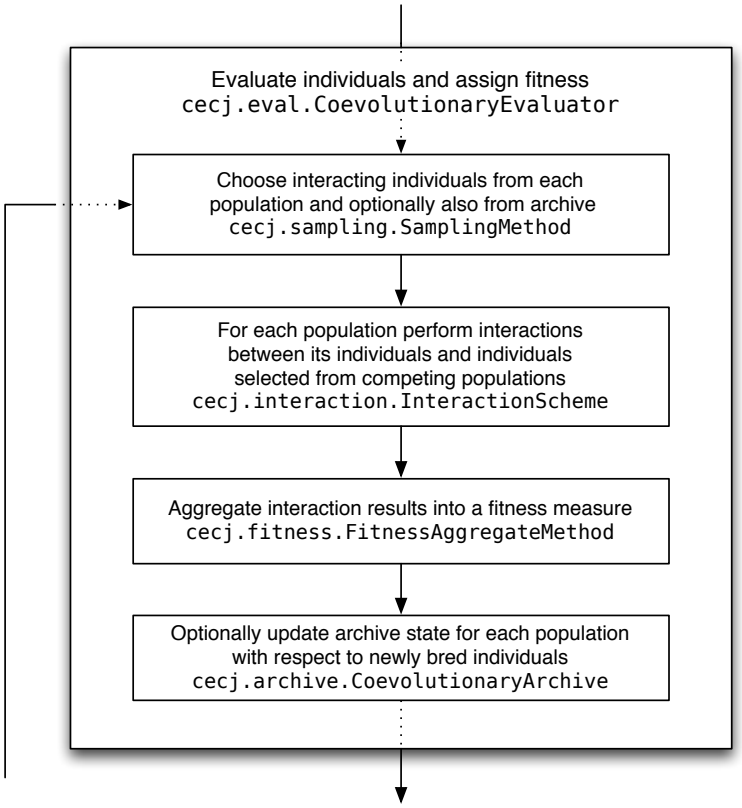


Fig. 4.4: Evaluation in cECJ

Notice also base classes responsible for particular steps. Different ways of realizing abstract methods from the workflow diagram are defined by concrete subclasses presented in Chapter 5.

4.2.2 Archive Mechanisms

Apart from expanding the evolutionary loop by a much more complex evaluation stage, the second major extension of the basic ECJ functionality is incorporating archives into evolutionary process. Archive mechanisms play three important roles:

- provide genetic material for future generations – an archival individual can be selected as a parent of a new generation's offspring,
- improve accuracy of the fitness evaluation of individuals in the population – their interaction partners can be sampled from the archive,
- approximate desired solution concept – while population performs exploration of the solution space, archive is aimed at storing the most promising individuals with respect to the implemented solution concept.

Since archive mechanism is an intrinsic part of the majority of modern coevolutionary algorithms, they have significantly influenced the design of cECJ library.

4.2.3 cECJ Class Diagram

A simplified class diagram of the most essential classes in cECJ extension is depicted in Figure 4.5. Once again the legend in the left bottom corner explains the meaning of the white-colored with respect to ECJ's interfaces introduced in the previous section. The left part of this diagram presents hierarchy of evaluators extending `ec.Evaluator` abstract class. Except `TournamentCoevolutionaryEvaluator` which is a dedicated solution for a special type of intra-population interaction scheme, simple and archiving evaluators make extensive use of helper classes shown on the right side. These classes are white-colored since they do not implement any of the original ECJ's core interfaces.

The helper classes were already mentioned in Figure 4.4 because they are responsible for particular steps of evaluation in coevolutionary algorithms. Extracting each of these simple functions as a separate interface allows for a lot of flexibility during configuration of experimental setup. Different implementations of particular steps can be mixed together in a number of ways without any limitations.

Additionally, a few of the original ECJ's classes were subclassed specially for coevolutionary purposes. `ArchivingSubpopulation` (extending `ec.Subpopulation`) is used by archives and corresponding evaluators to store archival individuals. Keeping all individuals (from population and archive) in one place allows for mixing individuals from both sources by the breeding mechanism. Obviously, this can result in producing more promising offspring. Another class is `TestBasedProblem` (a subclass of `ec.Problem`) which plays a role of a marker interface for problems applicable to coevolution. Furthermore, this class adds a dedicated method for computing an interaction result between two individuals among which one is considered as a candidate solution while another one as a test. More detailed description of particular classes and their exact implementations can be found in Chapter 5.

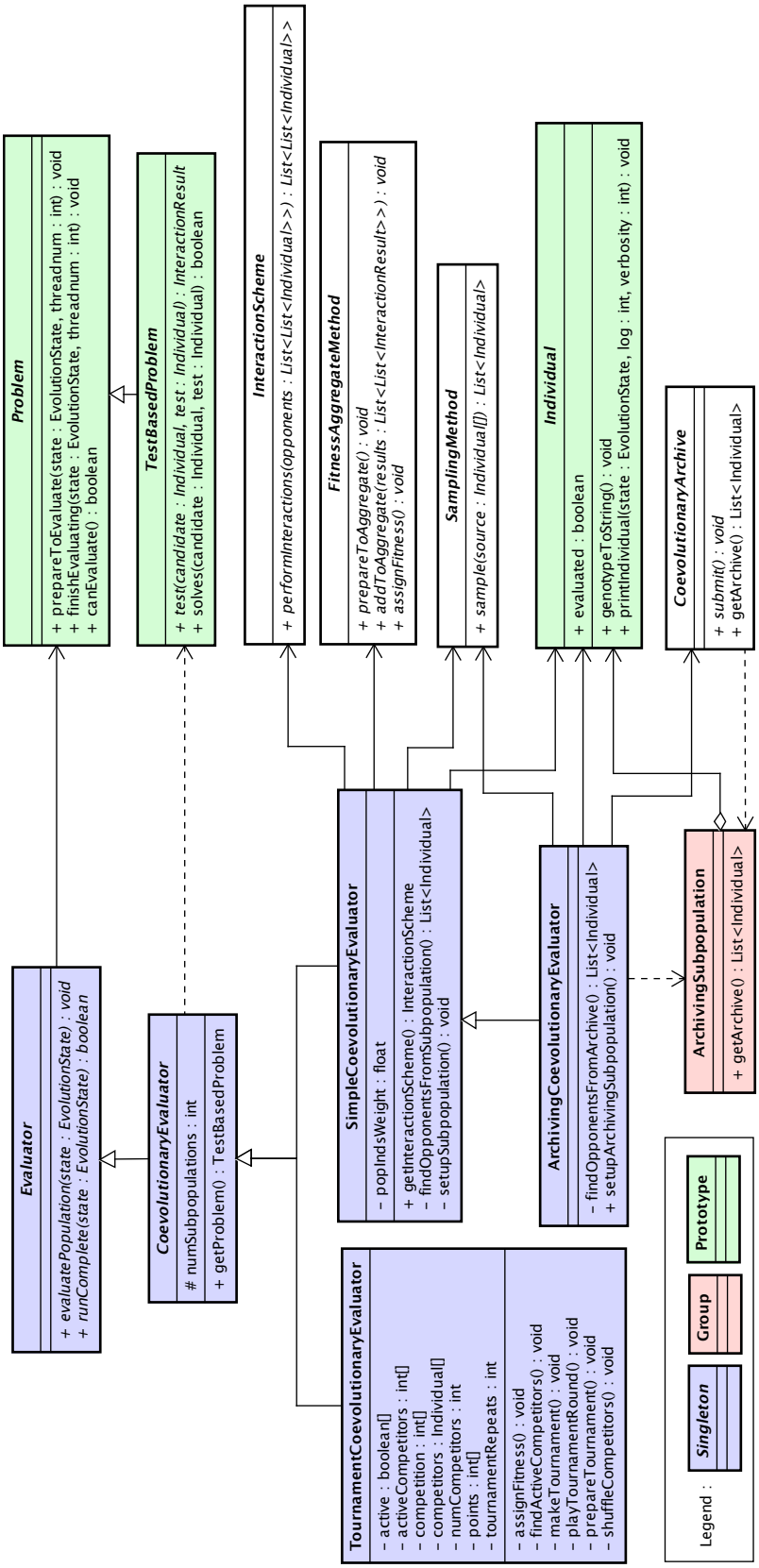


Fig. 4.5: cECJ simplified class diagram – only the most important classes and interfaces are illustrated; not all methods are precisely modeled.

Chapter 5

cECJ Implementation

This chapter reviews in detail the whole implemented cECJ library. Particular sections correspond to cECJ packages and specify how different stages of the coevolutionary run are realized. Every implemented class which defines one method of taking certain step of the coevolutionary process is shortly presented. Therefore, this chapter can be viewed as a documentation useful for preparing parameter files and defining custom experiments.

Recall that one of the main features of the original ECJ system is flexibility in defining an experimental setup without necessity of source code modifications. This was achieved by employing the concept of parameter files that is reused and extended by the cECJ library in the following way. Every step taken in the coevolutionary process can be performed in a number of ways, but to avoid implementing each of possible combination as a separate class, delegation mechanism was utilized as figure 4.5 illustrates. For example, `SimpleCoevolutionaryEvaluator` delegates evaluation subtasks like sampling or fitness assignment to dedicated interfaces. The parameter file's role is to specify their concrete realizations.

5.1 Evaluators

The most significant change in the evolutionary process, comparing to ECJ, concerns `ec.Evaluator` implementation; indeed, this is the only one of the core `ec.Singleton` classes maintained by `ec.EvolutionState` that must be changed in coevolutionary setup. Consequently, the `cecj.eval` package containing evaluators implementation is the main cECJ package – the hierarchy of created classes is illustrated in Figure 5.1. The abstract `CoevolutionaryEvaluator` class just verifies if a given problem is applicable to coevolution and overrides the `runComplete` method; this method always returns `false` because coevolutionary algorithms are not able to determine if an ideal solution has been found. Before we explore particular subclasses, note that the enhanced evaluation procedure shown in Figure 4.4 is realized in its entirety only by `ArchivingCoevolutionaryEvaluator` while the other ones fulfill it partially.

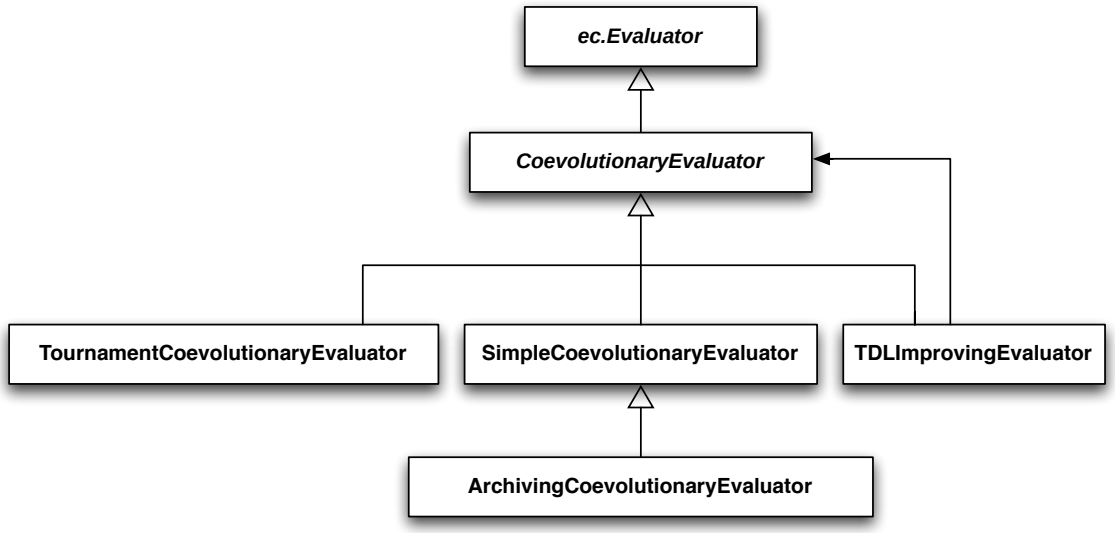


Fig. 5.1: Evaluators hierarchy

- `TournamentCoevolutionaryEvaluator` is an evaluator dedicated to so-called Single Elimination Tournament (SET) described in greater detail in Chapter 2.2.2.1. This class is completely different from the other evaluator types because of two reasons. First, it is compatible only with single population coevolution. Second, the interactions between individuals must be performed in a certain order because it depends on the outcome of previous interaction if the individual can compete further. Since it would be hard to extend this evaluator with generic archiving or fitness sharing, only the simplest settings are available. To reduce the inherent noise of the tournament evaluation scheme, a few rounds can be played. The number of rounds is specified by a `repeats` parameter which is equal to 1 by default.
- `SimpleCoevolutionaryEvaluator` is the simplest implementation of conventional coevolutionary evaluation where interactions between individuals can be performed in an arbitrary order. However, the character and the scope of interactions can be different – it is defined by instantiating appropriate `InteractionScheme` subclass (see Section 5.3.2). The evaluation proceeds as follows. First of all, a reference set of opponent individuals is selected from each subpopulation. This task is handled by a `SamplingMethod` realization. Distinct sampling methods can be used by different subpopulations – available types are precisely described in Section 5.3.1. Next, each subpopulation individuals are confronted with previously selected opponents from subpopulations pointed by the concrete `InteractionScheme` class. Finally, `FitnessAggregateMethod` is responsible for aggregating outcomes of these confrontations into a single fitness measure which is used later during selection stage of the evolutionary process.

- `ArchivingCoevolutionaryEvaluator` extends the simple evaluation process with an archiving mechanism. The role of archives is presented in Section 4.2.2, whereas their implementation details are discussed in Section 5.2. In this class, the evaluation procedure is realized in the following manner. Firstly, after taking simple evaluation steps mentioned above, additional opponents are selected among archival individuals (an archive is maintained for each subpopulation). Outcomes of the interactions with such opponents are added to results obtained by the superclass and aggregated together. Eventually, subpopulation individuals are submitted to the archive which decides if any of them is worth keeping. While interaction scheme and fitness aggregation method are inherited from the `SimpleCoevolutionaryEvaluator`, archival sampling methods must be defined separately for each of the archives. Often, opponents sampled from the population are less competent than these from the archive; to address this issue, an additional parameter was created that specifies the importance of opponents from the particular source.
- `TDLImprovingEvaluator` implements the Decorator pattern [Gamma 95] and thus acts as a wrapper for any other `CoevolutionaryEvaluator`. The only role of this class is improving each individual of the population by running a specific temporal difference learning (TDL) algorithm before the evaluation. Since the exact implementation of this algorithm depends on the problem, evaluator delegates the learning task to the provided `TDLImprover` interface realization. At the beginning of evolutionary process individuals may require some preparation for running TDL. For this reason, appropriate interface methods are invoked before the first evaluation. Clearly, not every problem can be approached by reinforcement learning paradigm so this class has also a limited scope of applicability. Note, however, that this evaluator realizes the Coevolutionary Reinforcement Learning idea introduced in Chapter 3.

5.2 Archives

Archives form a remarkable part of every modern coevolutionary algorithm. Theoretical issues concerning archives and their roles are discussed in Sections 2.2.2.2 and 5.2. In this section we focus on the implementation details. Figure 5.2 illustrates the hierarchy of created archive classes contained in the `cecj.archive` package. Once again there is an abstract class at the top of the hierarchy (`CoevolutionaryArchive`) that checks the compatibility with the rest of the configuration. Nevertheless, the main functionality is left to be defined in the `submit` method within subclasses. This method is called by `ArchivingCoevolutionaryEvaluator` after evaluating the population and its role is to add promising individuals from the population to the archive.

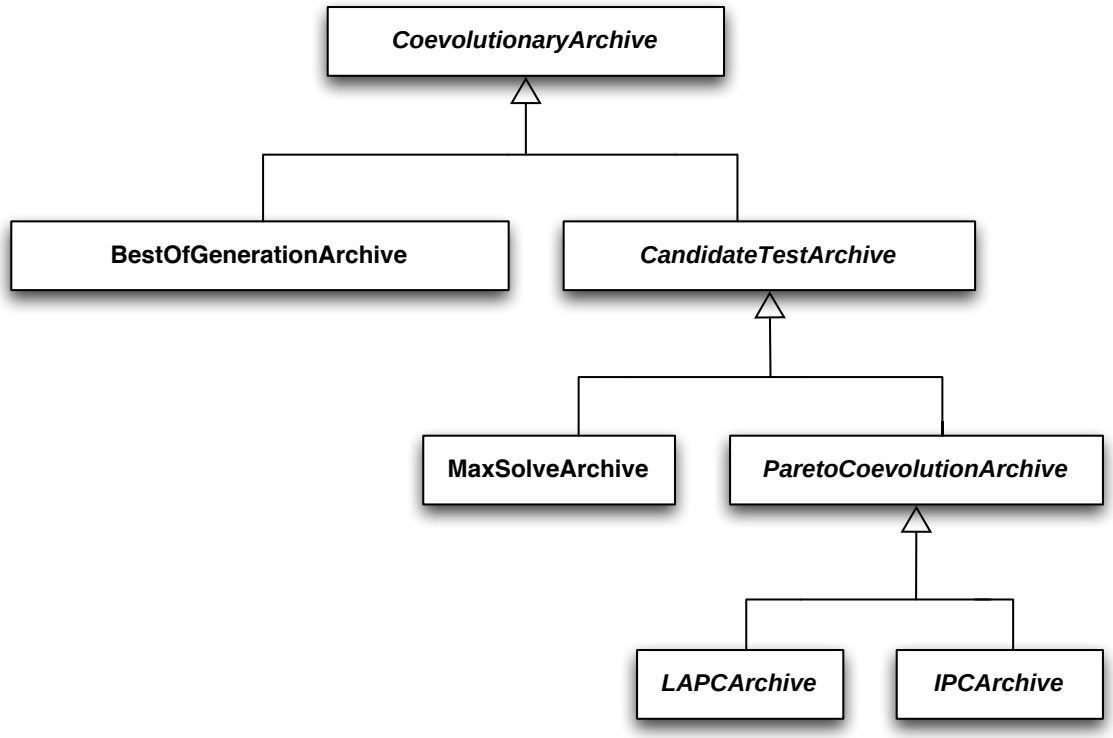


Fig. 5.2: Archives hierarchy

- `BestOfGenerationArchive` (BoG) is the simplest archive type which just finds the best individual from the submitted population and appends it to the list of individuals found in previous generations. Note that, by default, the archive size is unbounded and grows steadily over time because no archival individuals are removed. By setting the `archive-size` parameter value to x, only the best competitors from last x generations are maintained.
- `CandidateTestArchive` is an abstract class representing an archive dedicated for learner-teacher paradigm. Thus, it checks if appropriate interaction scheme (i.e., `LearnerTeacherInteractionScheme`) is used and finds out the role of each subpopulation. Submitting a population to this archive results in extracting candidate and test individuals and passing them to subclass methods.
- `MaxSolveArchive` is a straightforward implementation of the algorithm proposed by de Jong [de Jong 05]. During the first step of the `submit` method, new candidates and tests are added to corresponding archives; then duplicate candidate solutions are eliminated (two individuals are considered equal if for each test in the archive they have identical outcomes). After sorting candidates by number of solved tests, the best `archive-size` individuals are kept in the archive. The last step is elimination of unsolved and duplicated (with respect to the outcomes against archival candidates) tests. Since duplicates removal is a common task, a purpose-built `EquivalenceComparator` interface is used for defining custom equivalence criteria.

- `ParetoCoevolutionArchive` is another abstract class that is useful in Pareto-Coevolution paradigm where each test is viewed as an objective in the sense of Multi-Objective Optimization. The class provides methods for comparing individuals on the basis of their interactions outcomes considered in the context of Pareto dominance relation (methods `dominates`, `isDominated`). According to the results of these comparisons a test can be found *useful* if it proves that a particular candidate solution is not dominated by any other individual in the archive. Such usefulness can be verified using the methods of this class (`isUseful`, `findUsefulTest`).
- `IPCArchive` (Incremental Pareto-Coevolution Archive) is an implementation of the archiving algorithm introduced by de Jong [de Jong 04b]. For each of the submitted candidates it is checked if any *useful* test exists in the archive or currently submitted population, that proves the candidate is non-dominated. If such test is found, the considered individual is added to the archive while all individuals that it dominates are removed. The implementation relies heavily on the methods provided by the superclass – `PareroCoevolutionArchive`.
- `LAPCArchive` (Layered Pareto-Coevolution Archive) is a modified version of the IPCA archive described above (see also [de Jong 04a] and [de Jong 07]). While the original one can grow indefinitely (tests are never removed from the archive), this type of archive limits the maximum number of stored individuals. However, this goal is achieved for the price of reducing the reliability of the algorithm. After appending non-dominated candidate solutions and useful tests to appropriate archives, `maintainLayers` and `updateTestArchive` methods are invoked in order to decrease the amount of used memory. The first one checks which candidate solutions belong to the first `num-layers` Pareto layers and keeps them in the archive. The second retains only these tests which make distinctions between neighboring layers.

5.2.1 Archiving Subpopulation

The above archive description raises the question of where archival individuals should be stored. There are a few possible places including the `CoevolutionaryArchive` subclasses. For breeding purposes, however, the best place is near all the rest of individuals – in the population. Thus, a dedicated `ArchivingSubpopulation` class was created with an additional list of individuals forming an archive. This class overrides `emptyClone` method called by `ec.Breeder` to create a storage for future generation. In contrast to the array of population individuals which is allocated but not filled until breeding starts, the list of archival individuals is entirely copied from the last generation. Note that `CoevolutionaryArchive` instances gain access to this list by calling the `getArchivalIndividuals` getter method.

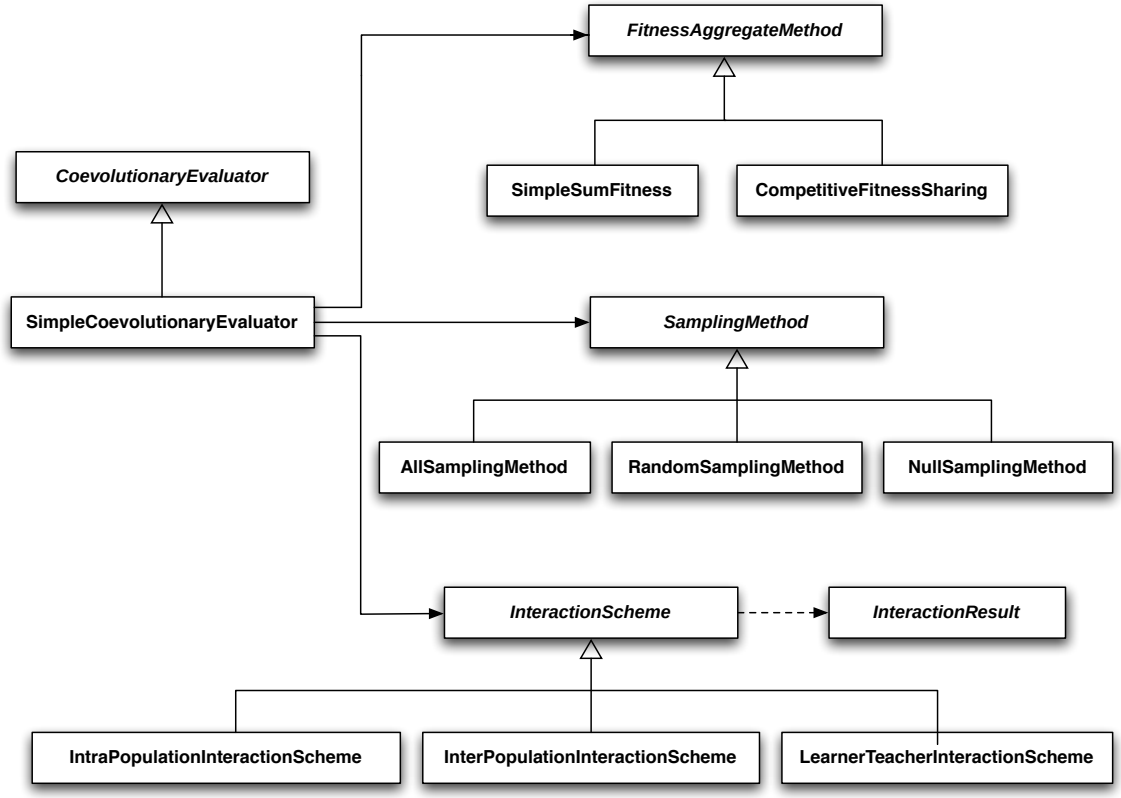


Fig. 5.3: Interaction schemes hierarchy

5.2.2 Archive as a Breeding Source

One of the main roles of an archive is to provide genetic material for future generations. This concept is realized by `ArchiveRandomSelection` class which extends `ec.SelectionMethod` and allows for using archive as a breeding source. Individuals produced by this class are selected randomly from the `ArchivingSubpopulation`. In addition, parameter `size` can be used to limit the scope of selection to last `size` individuals recently appended to the archive.

5.3 Evaluating Infrastructure

Since the evaluation stage in coevolutionary algorithms is relatively sophisticated, a few helper interfaces were extracted. These interfaces address particular subproblems of the evaluation process as it is illustrated in Figure 4.4. In the following subsections particular interfaces and their example realizations will be described in the order in which they are employed by evaluator classes. All these entities are depicted in Figure 5.3. Note that the overall evaluation process is largely influenced by choosing convenient implementations among these classes.

5.3.1 Sampling Methods

Before interactions can be performed, each subpopulation must be sampled in order to choose a reference set of individuals representing this group. This task is delegated to the `SamplingMethod` interface and its `sample` method. The only role of this method is to select a subset of individuals from the source set provided in the form of an array or `java.util.List`. Clearly, the number of selected individuals has a great impact on the amount of time spent on evaluating the whole population. Only a few easy sampling methods were implemented, but it would be interesting to add Shared Sampling presented in [Rosin 95], for instance. Classes representing particular methods form `cecj.sampling` package.

- `AllSamplingMethod` is a trivial implementation which just returns the unchanged source as a list of sampled individuals. This type of sampling results in the largest number of interactions and thus it is the most time-consuming.
- `RandomSamplingMethod` selects randomly a combination of individuals (with repetitions allowed). The class can be customized by setting `sample-size` parameter to control the number of sampled individuals. By default, it is equal to 1. This method is known in the literature as *k-random opponents*.
- `NullSamplingMethod` was implemented instead of adding another parameter or specialization of evaluator class which does not take into account interactions with particular sources of individuals (i.e., subpopulations or archives). This method can be viewed as a variation of the Special Case design pattern.

5.3.2 Interaction Schemes and Interaction Results

The second delegate interface utilized by `CoevolutionaryEvaluator` subclasses to evaluate a population is `InteractionScheme`. Classes implementing this interface specify between which subpopulations interactions take place and how to obtain their outcomes. Particularly, they should know what method of the given problem definition is capable of deciding about a result of particular interactions. The only interface method (`performInteractions`) returns the list of `InteractionResult` objects. Examples of classes representing such results are `WinDrawLossResult` and `RealValuedResult`. The exact character of an outcome is problem-dependent and the interaction scheme may abstract from it. All the interactions-related classes belong to `cecj.interaction` package.

- `IntraPopulationInteractionScheme` defines a basic interaction scheme. Each individual in the particular subpopulation competes with a sampled reference set of opponents from the same subpopulation. Since the payoff matrix of the problem can be asymmetric, the competition between two individuals is repeated twice – each individual plays both roles. This class requires that problem implements `SymmetricCompetitionProblem` interface.

- `InterPopulationInteractionScheme` is a scheme where all possible combinations of two subpopulations are considered. Certainly, there is $\binom{n}{2}$ such subsets, where n is a number of subpopulations. However, every combination implies two possibilities of interactions because competing populations may have distinct roles. For this reason, for each subpopulation pop_1 we examine every other subpopulation pop_2. Each such pair in the form $< pop_1, pop_2 >$ implies that all individuals from pop_1 are confronted with a reference set of competitors sampled from pop_2.
- `LearnerTeacherInteractionScheme` is a specific kind of interactions between different subpopulations intrinsic to the learner-teacher paradigm. In contrast to `InterPopulationInteractionScheme` described above, not every pair of subpopulations can interact. In order to distinguish pairs that should be confronted, the `role` parameter was introduced – populations marked with `LEARNER` role interact with `TEACHER` populations.

5.3.3 Fitness Aggregation Methods

The last step of the evaluation procedure is an aggregation of gathered interaction results into a single fitness measure associated with the `ec.Individual` object. This issue is addressed by the `FitnessAggregateMethod` interface, whose `addToAggregate` method allows for aggregating outcomes coming from many sources. Note that one of the arguments of this method is `weight` which specifies how important certain results are. For example, `ArchivingEvaluator` combines results of interactions with population and archival individuals. Furthermore, it uses `pop-inds-weight` and `archive-inds-weight` parameters to determine an impact of interactions with individuals from particular source (population and archive, respectively) on the final fitness value.

Only two fitness aggregation methods were implemented – they are contained in `cecj.fitness` package. Both of them assume that individuals are configured to use `ec.simple.SimpleFitness` that is consisting of a single floating-point value. Pareto dominance rank is potentially another method that could be added – it is quite common in Evolutionary Multi-Objective Optimization.

- `SimpleSumFitness` just adds numeric values of all the interactions results associated with a certain individual and treats the sum as a fitness value.
- `CompetitiveFitnessSharing` is an implementation of popular method proposed by Rosin and Belew [Rosin 97]. Each opponent individual is treated as a unit resource that can be shared among individuals which are able to overcome it. The accurate amount of this resource awarded to particular individuals depends on the outcome of their interaction with defeated opponent.

5.4 Test-based Problems

Obviously, not every single problem is applicable to coevolutionary approach. For this reason, there must be a way to check if the given problem is a proper one. `CoevolutionaryProblem` is a marker interface that has to be implemented by a concrete `ec.Problem` subclass in order to indicate that it is prepared to be used with coevolutionary evaluators. The role of a problem definition in this context is to describe how interaction proceed, what kind of individuals interact with each other and what type of interaction result is returned.

The most popular arrangement in coevolutionary algorithms is known as the *learner-teacher* or *test-based* paradigm (see Chapter 2). Typically, this results in two coevolving populations with different roles and search spaces. For purposes of this paradigm, the abstract `TestBasedProblem` class was created.

5.4.1 Caching Evaluation Results

Often, during a single coevolutionary run, millions of interactions are performed. If the number of possible individuals genotypes is limited (i.e., search space is finite) then it may happen that identical individuals meet many times within different generations. Thus, it would be useful to cache the interaction results between such individuals and in case they meet again, just find the result of their previous interaction. `TestBasedProblemCachingDecorator` plays a role of such cache manager. The class wraps the original `TestBasedProblem` instance and asks it for a particular interaction outcome if it is encountered for the first time. Using of such utility can result in a significant efficiency improvement. Note, however, that it cannot be used if an interaction result is not deterministic. For instance, backgammon game is based on a roll dicing, so caching game result between certain players is impossible.

Regarding implementation details, it is worth pointing out that internally the caching decorator class employs `java.util.HashMap`. Additionally, to deal with unlimited memory requirements of the cache manager, simple LRU (*least recently used*) algorithm was implemented. The algorithm is fired up when the size of cached interaction results exceeds the value specified by the `cache-size` parameter. After sorting map entries with respect to times of last access, the older half is removed.

5.4.2 Sample Problems – Numbers Game and Othello

Two sample test-based problems were implemented for experimental purposes. The first one is a well-known Numbers Game which was introduced by Watson [Watson 01] and extended later by de Jong [de Jong 04c] who creates Compare-On-One and Compare-On-All versions of the game. The second problem is the board game of Othello (see Chapter 3). The source code of both problems definitions can be found in corresponding subpackages of `cecj.app` package.

5.5 Objective Fitness

In standard evolutionary algorithms the fitness of individuals is computed by an objective fitness function which does not change during evolutionary process. This function can provide some measure of individual's performance in the context of the given task. For example, considering Traveling Salesman Problem, the fitness function could be naturally defined as a length of the Hamiltonian cycle in the weighted complete graph (the order of vertices forming a cycle is controlled by an individual's genotype). In coevolution, there is still a fitness function – returning a result of aggregating outcomes of interactions between individuals – but it is no longer objective. Certainly, the fitness value assigned to an individual by a coevolutionary algorithm depends on randomly chosen opponents from competing species. The same individual can obtain diametrically different evaluations because of the randomness of the evaluation procedure. Even if we decide to choose all the population individuals as opponents, the fitness is still subject to the state of coevolving populations.

5.5.1 Objective Fitness Calculator

Since we seek to optimize the objective fitness which allows us to compare solutions obtained by different algorithms, we have to define some objective measure of the individual's quality. For test-based problems, however, there is often no natural way to accurately evaluate candidate solutions. One of the most intuitive methods of approximating performance of an individual is to simulate interactions with random or expert player. Obviously, the exact algorithm computing an objective fitness is dependent upon the problem at hand. `ObjectiveFitnessCalculator` is an interface that has to be implemented by such algorithm to enable it to be used with statistics classes presented below.

5.5.2 Objective Fitness Statistics

Statistics information in standard ECJ's distribution is gathered by `ec.Statistics` subclasses. They are prepared to log the traditional fitness which is associated with each individual and guides the selection stage. Nevertheless, in coevolutionary algorithms this fitness is subjective and it usually does not provide precise information about real individual's quality. Thus, it cannot be used to compare individuals evolved in different environments nor even to monitor a progress of the population performance. To address these issues `ObjectiveFitnessStatistics` class was defined. Every `frequency` generations, it calculates the objective fitness of each individual in the population using provided `ObjectiveFitnessCalculator` instance specified by `fitness-calc` parameter. On the basis of calculated fitnesses, statistical measures like average and standard deviation are computed and written together in the `fitness-file` log.

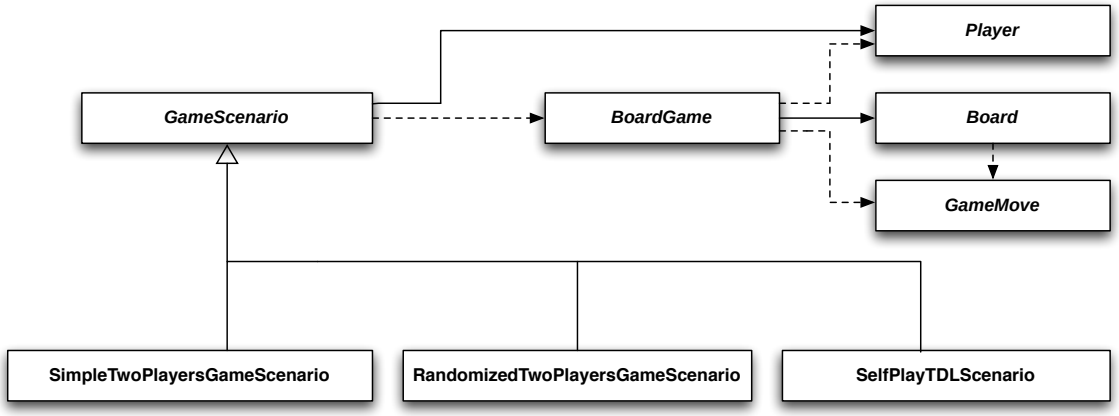


Fig. 5.4: Games interfaces interaction diagram

Additionally, `ObjectiveFitnessChartingStatistics` was implemented that integrates with ECJ's GUI utilities. This class allows for plotting the chart of an average objective fitness of the population.

5.6 Board Games Interfaces

Because board games are one of the main applications of coevolutionary algorithms, a set of interfaces was defined to ease the future development of such games. These interfaces and other game-related classes are contained in `games` package. Figure 5.4 illustrates interactions inside this package. The upper part of the picture is occupied by `Player`, `BoardGame`, `Board` and `GameMove` interfaces whose role is to describe the game rules including the following aspects:

- verifying if the game is finished,
- determining what legal moves are available in a certain game state,
- specifying how the board state changes after a particular move is made,
- evaluating particular moves according to the player's knowledge,
- getting the final outcome of the game.

Moreover, the `GameScenario` interface was introduced to represent different ways of playing a game. Apart from deterministic `SimpleTwoPlayersGameScenario`, `RandomizedTwoPlayersGameScenario` is defined where each player can be forced to make a random move. The proportion of random moves is controlled by the probabilities given as arguments to scenario constructor. The last type of scenario depicted in the figure is `SelfPlayTDLScenario` which concerns single player's self-play. During the game, the player's evaluation function is updated according to temporal difference (*TD(0)*) algorithm. In this scenario forcing random moves is a typical method to explore the game tree.

The game we devote the large part of our experiments (see Chapter 6) is Othello that is described precisely in Chapter 3. The implementation of this game is contained in `cecj.app.othello` package, but it uses interfaces from `games` package.

Chapter 6
Experiments and Results

In this chapter we present several computational experiments and their result. These experiments were performed to verify our hybrid approach to the problem of learning Othello strategy and compare it to preexisting methods. We begin by describing a detailed setup of all tested algorithms and the way the players are evaluated (see Section 6.1). Next, in Section 6.2 we present a set of experimental results including direct comparison of all methods and round-robin tournament between the best players of every run. Additionally, we investigate the issue of appropriate intensity of individually performed temporal difference learning. Finally, we present some minor findings concerning influence of the initialization on the speed of learning. All the following experiments were implemented using our *co-Evolutionary Computation in Java* (cECJ) library described in detail in Chapters 4 and 5. Some of the parameter files used in these experiments can be found in Appendix B.

6.1 Experimental Setup

We conducted several experiments to test different algorithms of learning Othello strategy, that were introduced in Chapter 3 – Temporal Difference Learning (TDL), Coevolutionary Learning (CEL), and a hybrid Coevolutionary Temporal Difference Learning (TDL + CEL = CTDL) based on two previous methods. Additionally, CEL and CTDL setups were also tested with a simple Hall of Fame (HoF) archive that maintain the set of best-of-generation individuals [Rosin 97].

To avoid precise tuning of parameters for this specific problem, we used the most typical and intuitive configuration of given algorithms. However, at the same time some settings were based on previous research [Lucas 06] which we want to consult regarding our findings. In order to compare different methods fairly, for all experiments we set the number of generations (CEL) or steps (TDL) so that the total number of played games did not exceed 4.5 million. For statistical significance, each experiment was repeated 30 times with different pseudorandom generator seed values. In the next section we give detailed information about settings of each method involved in comparison.

6.1.1 Methods

6.1.1.1 TDL — Temporal Difference Learning

TDL is a straightforward implementation of gradient-descent temporal difference algorithm – *TD(0)* described in Section 3.2.2. This setup represents an autonomous reinforcement learning approach. All WPC weights are initially set to 0 and the learner is trained solely through self-play. Random moves are forced with probability 0.1 and the value of learning rate $\alpha = 0.01$. An important issue is that directly after making a random move learning does not occur and, consequently, the WPC is not updated.

6.1.1.2 CEL — Coevolutionary Learning

CEL uses a generational coevolutionary algorithm with a population of 50 individuals. Each individual is represented by an array of 64 doubles which is used as a WPC vector. Initially, all weights are set to 0 whereas during mutation they are limited to the range $[-1, 1]$. In evaluation phase, a round-robin tournament is played between all individuals. Like in many sports leagues, *three points for a win* awarding standard is used (win – 3 points, draw – 1, loss – 0). The sum of the scores achieved in this tournament serves as a final fitness value. The evaluated individuals are selected using standard tournament selection (tournament size 5), and then, with probability 0.03, each of their weights undergo Gaussian mutation ($\sigma = 0.25$). Next, they mate using one-point crossover, and the resulting offspring is the only source for the subsequent generation (there is no elitism). As each generation requires 50×50 games, the run lasts for 1800 in order to attain 4.5 millions of training games.

Parameter file for this setup is presented in Appendix B.1.

6.1.1.3 CEL + HoF — Coevolutionary Learning with Hall of Fame

This setup extends the previous one with the Hall of Fame archive consisting of best-of-generation individuals encountered during the course of evolution. In evaluation phase, each individual plays games with all 50 individuals from the population (including itself) and, additionally, with 50 randomly selected archival individuals. According to the outcomes of these games, the best individual is selected and copied into the archive. The archive serves also as a source of genetic material for breeding purposes, as the first parent of crossover is randomly drawn from it with probability 0.2. Since in each generation $50 \times (50 + 50)$ games are played, 900 generations are required to reach necessary number of games.

Parameter file for this setup is presented in Appendix B.2.

6.1.1.4 TDL + CEL = CTDL — Coevolutionary Temporal Difference Learning

CTDL combines TDL and CEL as it was described in Section 3.3.1, with the TDL phase parameters adapted from 6.1.1.1 and the CEL phase parameters – from 6.1.1.2. CTDL starts with a population of players with weights set to 0 and alternately repeat TDL phase and CEL phase until the total number of 4.5 million games is attained. The only new parameter introduced here is the intensity of individually performed learning, which is expressed by a number of TDL games played per each TDL-CEL cycle. This parameter determines the exact number of generations. For example, 10 TDL games played each time lead to a total of 3000 games per generation (including round-robin tournament) and run length of $1,500$ generations.

Parameter file for this setup is presented in Appendix B.3.

6.1.1.5 TDL + CEL + HoF = CTDL with Hall of Fame

The last setup represents the most complex method combining setups 6.1.1.3 and 6.1.1.4. It does not involve any extra parameters.

6.1.2 Strategy Evaluation

Before we discuss particular experiments and obtained results, we need to consider an important issue of how to measure the quality of evolved players. As learning game strategy is an example of a *test-based problem*, accurate objective function cannot be easily defined. Ideally, objective evaluation of an individual should take into account games with all possible players and be based on a particular solution concept [Ficici 04]. This approach is impossible to implement in practice due to inconceivably high number of possible strategies for Othello. Thus, following [Lucas 06], we rely on two computationally feasible approximate quality measures which are presented below.

In order to monitor the progress in an objective way, 50 times per experimental run (approximately every $90,000$ games) we assess the quality of the best-of-generation individual. Both of defined measures estimate individual's quality by playing $1,000$ games (500 as black and 500 as white) against certain opponent(s) and calculating the probability of winning. Note that the evaluation games, similarly to learning ones, are also played at 1-ply.

- *Playing against a set of random players:* Arguably, this is the most common method employed in such circumstances, which tests how well the player fares against a wide variety of opponents. The evaluation is based on playing with player that for each board state chooses a random move. Notice that this quality measure approximates the solution concept of *Maximization of Expected Utility* introduced in [Ficici 04].

- *Playing against a standard heuristic player:* This measure tests how well the player copes with moderately competent opponent whose WPC is depicted in Figure 3.2. Since Othello is a completely deterministic game, we force both players to make random moves with probability $\epsilon = 0.1$ to diversify their behavior and make the estimated values more continuous. Though this essentially leads to a different game definition, we assume that the ability of playing such a randomized game is highly correlated with playing the original Othello.

6.1.3 Choosing Final Solutions

Another important question is: how to choose a final solution from the population of strategies obtained at the end of evolution? Obviously, this problem does not concern a pure TDL method because there is only one learning entity. There are a few potential methods of tackling this issue:

- *Choosing a strategy with the highest subjective fitness:* This is probably the best thing we could do in practice if we do not have an access to any objective fitness measure. We just need to take into account the fitness based on intra-population interactions, assuming that it is correlated with an objective quality of the individual. This approach is used during our experiments, the individual selected in this way is denoted as *best-of-generation* individual.
- *Choosing a best strategy with respect to some estimation of objective fitness:* This is an interesting idea, but we need to evaluate the whole population what may be very time-consuming; reducing the amount of time spent on evaluation could lead to increasing the bias of the estimation.
- *Choosing a random strategy from the population:* Strategy chosen in this way is expected to have performance equal to average playing ability of the population individuals. Clearly, this is the easiest approach.

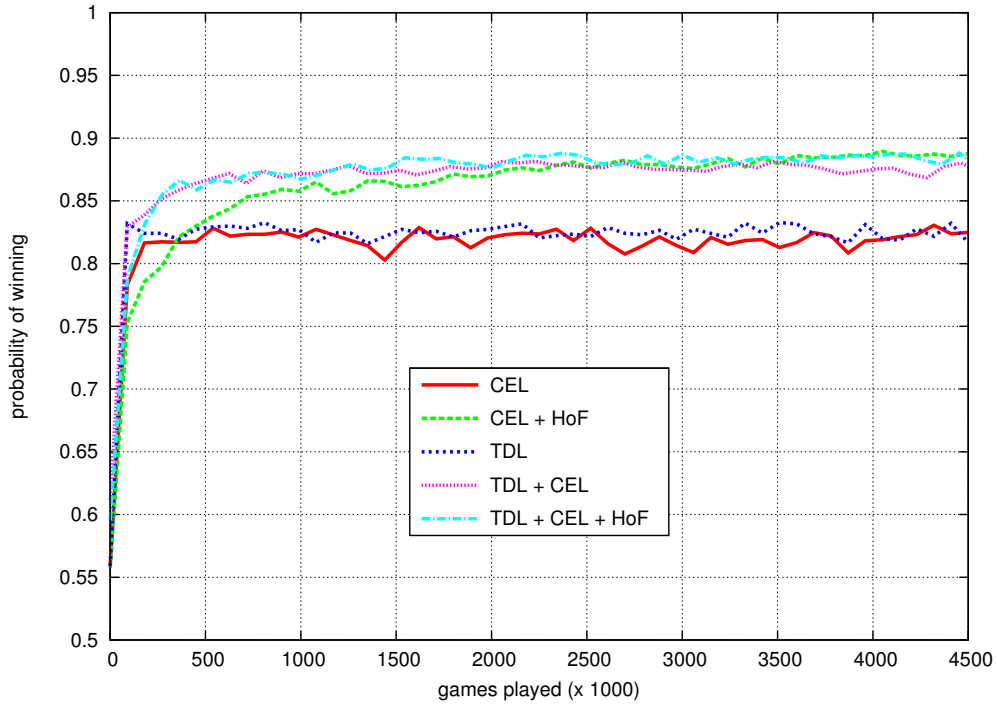


Fig. 6.1: Average performance of the best-of-generation individuals measured as a probability of winning against a **random** player, plotted against the number of training games played.

6.2 Main Results

In this section we present results of different experiments conducted using setups described in Section 6.1.1. We start with a basic comparison of all examined methods and the round-robin tournament between the best players found by each of them. Next, we consider more specific experiments.

6.2.1 Basic Comparison

In the first experiment, we compared all the five methods. Figure 6.1 illustrates how strategies produced by these algorithms perform on average against a pure random player. Note that for population-based methods (i.e., all except pure TDL), each point of the graph represents the objective quality (probability of winning) of a best-of-generation individual averaged over 30 runs. For TDL, the graphs show the average performance of the single solution maintained by this method. It is interesting to observe that the algorithms can be divided into two groups with respect to the performance they eventually achieve. The simplest non-hybrid methods (CEL and TDL) are in the long run significantly worse than hybrid ones. As expected, in the beginning, the quality of individuals produced by TDL-based algorithms is higher than those produced by other methods.

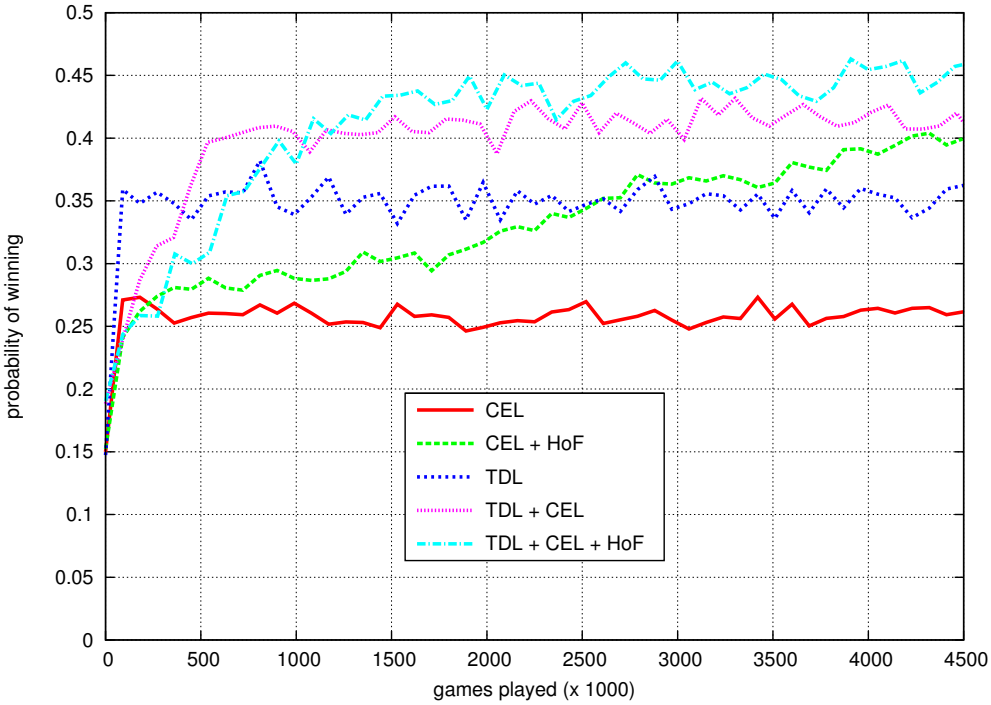


Fig. 6.2: Average performance of the best-of-generation individuals measured as a probability of winning against a standard **heuristic** player (both players were disturbed by random moves with probability equal to 0.1), plotted against the number of training games played.

Basing on the graph 6.1 alone, we can see that both methods TDL + CEL and TDL + CEL + HoF are decent choice, as they quickly achieve good performance and they are best in the long run. Interestingly, the line generated by CEL + HoF achieves eventually similar level to the hybrid TDL + CEL (+ HoF) approaches, but the learning proceeds slower.

Figure 6.2 illustrates the progress of the same set of methods but measured as the quality of the best-of-generation individuals versus a standard heuristic player. The most striking observation is that hybrid approaches once again outperform the original algorithms, this time even more remarkably. TDL + CEL + HoF is the best, then TDL + CEL, TDL, CEL + HoF and CEL, which is the worst here. It can also be observed that the HoF archive, despite its simplicity, help both CEL and TDL + CEL methods to achieve a higher level of play. Nevertheless, regarding CEL + HoF setup, it needs approximately 10 times more games than the simplest hybrid approach (TDL + CEL), to reach comparable performance. Another important remark is that TDL learns rapidly indeed, but the progress is observable only during the first several thousands of games and then it stagnates.

Team	Games	Wins	Draws	Defeats	Points
TDL + CEL + HoF	7200	4918	203	2079	14957
TDL + CEL	7200	4171	209	2820	12722
CEL + HoF	7200	3726	225	3249	11403
TDL	7200	2873	206	4121	8825
CEL	7200	1787	207	5206	5568

Tab. 6.1: Best-of-run tournament results.

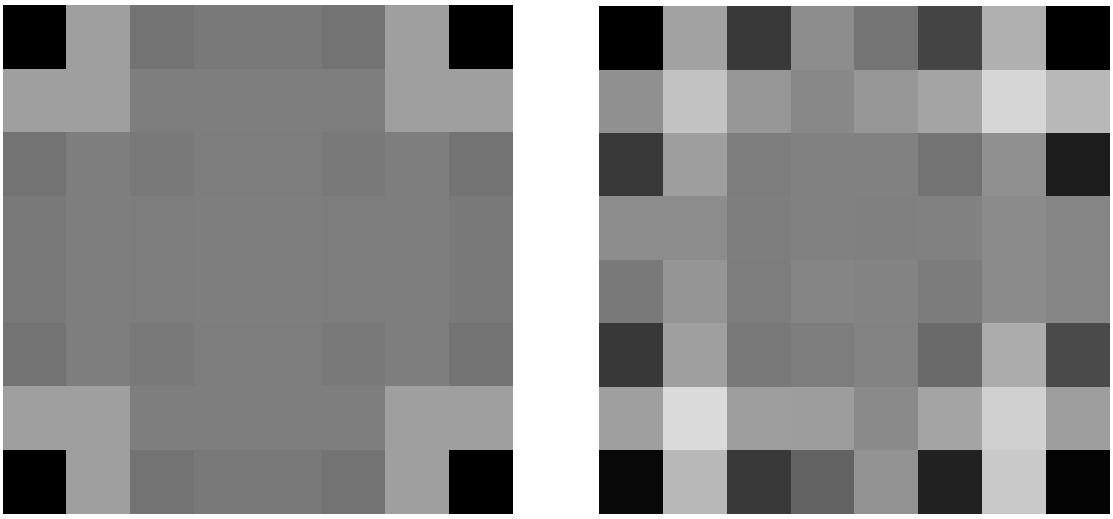
```
 1.02 -0.27  0.55 -0.10  0.08  0.47 -0.38  1.00
-0.13 -0.52 -0.18 -0.07 -0.18 -0.29 -0.68 -0.44
 0.55 -0.24  0.02 -0.01 -0.01  0.10 -0.13  0.77
-0.10 -0.10  0.01 -0.01  0.00 -0.01 -0.09 -0.05
 0.05 -0.17  0.02 -0.04 -0.03  0.03 -0.09 -0.05
 0.56 -0.25  0.05  0.02 -0.02  0.17 -0.35  0.42
-0.25 -0.71 -0.24 -0.23 -0.08 -0.29 -0.63 -0.24
 0.93 -0.44  0.55  0.22 -0.15  0.74 -0.57  0.97
```

Fig. 6.3: WPC vector of the best player in the tournament.

6.2.2 Best-of-Run Tournament

In order to confirm our results, we conducted a second experiment in which we investigate the relative ordering of the methods with respect to their results. We performed a round-robin tournament between all best-of-run individuals, i.e., the best-of-generation individuals from the last generation. We created 5 teams, one for each method, each one composed of 30 best-of-run individuals (one per run). Next, a round-robin tournament was played, where each strategy played against $4 \times 30 = 120$ strategies from opponent teams for a total of 240 games (120 as white and 120 as black). The final score of a team was determined as the sum of points obtained by its players in overall $7,200$ games.

The results of this competition presented in Table 6.1 confirm the former observations. We can conclude that definitely the best method in direct comparison is TDL + CEL + HoF approach. Moreover, the obtained ranking of methods is consistent with the ranking obtained from measuring the quality against a standard heuristic player (see Figure 6.2). This may be explained by the fact that in this tournament, an individual faces exclusively well-performing strategies, so what matters here is the ability to play against a competent opponent and not against a random one. It is noteworthy that the CEL method, although it is the worst in the field, can become much more successful thanks to employing any additional mechanism – TDL or HoF. Indeed, both TDL + CEL and CEL + HoF teams gained more than twice as many points as the simplest coevolutionary algorithm.



(a) Standard heuristic player **(b)** Winner of the best-of-run tournament

Fig. 6.4: WPC vectors illustrated as Othello boards with locations shaded accordingly to corresponding weights.

The WPC vector of the best scoring player, who is a member of the winner team TDL + CEL + HoF, is shown in Figure 6.3. Additionally it was also presented in a more illustrative way by means of a weight-proportional grayscale in Figure 6.4b (darker squares denote larger weights, i.e., more desirable locations on the board). An important observation is that the WPC matrix is quite symmetric. Similarly to the heuristic player's strategy (see Figure 6.4a), the corner locations are the most desirable while their immediate neighbors have very low weights. However, in contrast to the heuristic player, the edge locations having distance 2 from the corners get very high weights.

6.2.3 TDL Intensity

Preliminary experiments have shown that TDL intensity, i.e., the number of self-played TD games per generation, is an important parameter of CTDL. Technically, it is natural and convenient to interlace the CEL and TDL phases one-to-one. However, CEL learns only from the final game outcome, while TDL exploits information from every single move and, as the above results demonstrate, learns faster than CEL.

We have investigated this issue by running the best algorithm (TDL + CEL + HoF) for different TDL intensities – 1, 2, 5, 10, and 50 TDL games per each TDL-CEL cycle. The probability of winning of the best-of-generation individual against the random player for different TDL intensities, presented in Figure 6.5, proves that CTDL performance indeed depends on TDL-CEL ratio and that TDL-CEL ratio greater than 1 can be destructive. This figure demonstrates also the tradeoff between the learning speed and the ultimate player quality.

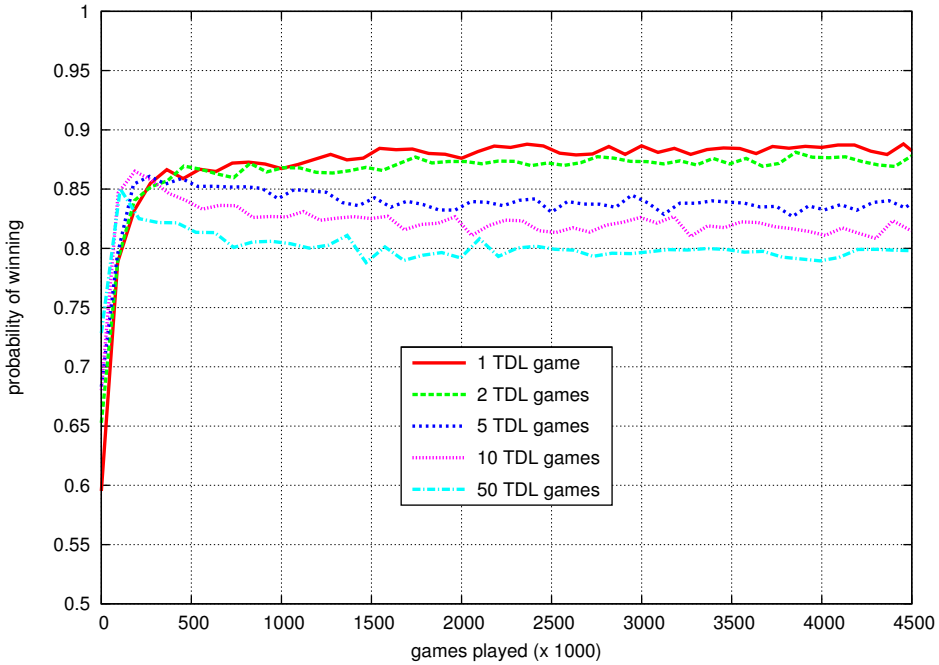


Fig. 6.5: Average performance of the best-of-generation individuals found by TDL + CEL + HoF method for different TDL intensities, measured as a probability of winning against a **random** player, plotted against the number of training games played.

In Figure 6.6, which illustrates the performance of produced strategies against a standard heuristic player, different TDL intensities cause only slight differences in the long run. This was confirmed by playing another best-of-run tournament between setups with different number of TDL games. The tournament ended ended with each team scoring a very similar number of points.

6.2.4 Negative Learning Rate

As we have seen above, the hybrid approach was able to learn remarkably better strategies than the non-hybrid methods. An interesting question is whether the purposeful (meaning: driven towards greater probability of winning) character of changes brought in by TDL is essential, or TDL plays the role of a mere mutation. To verify this, we compared regular TDL + CEL + HoF to TDL + CEL + HoF with learning rate $\alpha = -0.01$, which implies that TDL in the latter method deteriorates the strategies found by CEL. The results, shown in Figure 6.7, prove that a purposeful TDL is a key factor explaining the success of the hybrid approach.

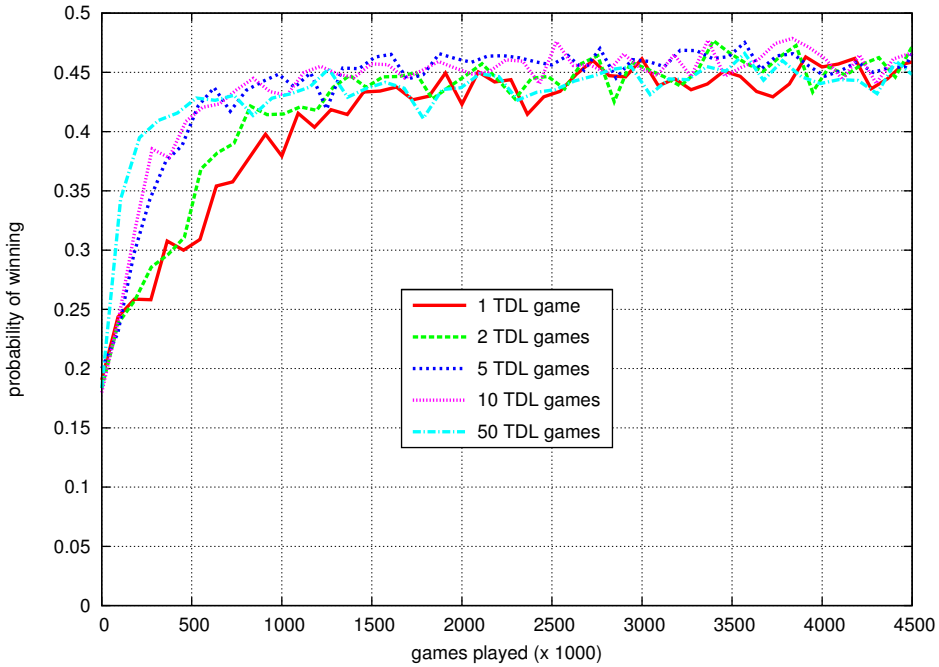


Fig. 6.6: Average performance of the best-of-generation individuals found by TDL + CEL + HoF method for different TDL intensities, measured as a probability of winning against a standard **heuristic** player (forcing random moves with probability 0.1), plotted against the number of games played.

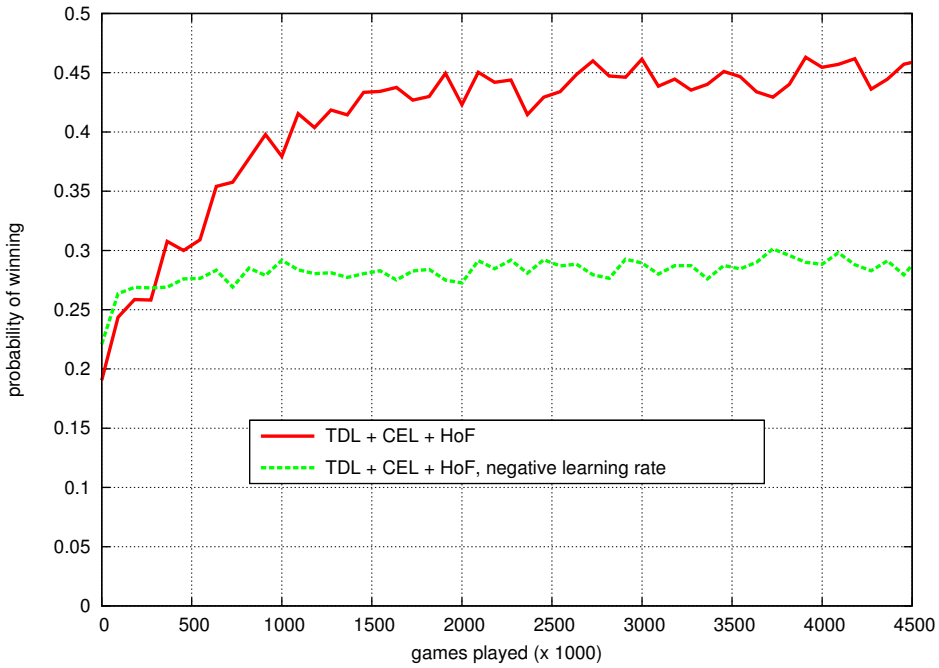


Fig. 6.7: Average performance of the best-of-generation individuals found by TDL + CEL + HoF method for different learning rate values of TDL, measured as a probability of winning against a standard **heuristic** player (both players were disturbed by random moves with probability equal to 0.1), plotted against the number of games played.

6.3 Minor Findings

In this section we present additional experiments that were brought about by observations made during the preliminary tests. Particularly, it was noticed that the way the strategies are initialized may have a substantial impact on the learning process. Recall from Section 6.1 that in case of TDL-based approaches initialization procedure sets all weights to 0. This is adapted from the previous work [Lucas 06] where it is reported that such initialization leads to the best results. However, in coevolution, similarly to traditional evolutionary approaches, random initialization is the most commonly used method for generating initial population as it results in covering different areas of the search space. Surprisingly, the following results illustrate, that for this problem, even population-based algorithms should start with zero-initialized candidate solutions.

Figures 6.8 and 6.9 show the results of TDL + CEL + HoF and CEL + HoF methods for two different initialization procedures. *Random-init* draws each weight uniformly from the range $[-1, 1]$ while *0-init* just sets all weights to 0 for all the individuals. The main observation is that proper initialization can largely affect the convergence speed of the learning process. Clearly, 0-init results in much faster convergence for both algorithms. However, regardless of the initialization procedure, our hybrid TDL + CEL + HoF algorithm outperforms conventional coevolutionary approach in the long run. This is especially visible in Figure 6.9. It is also interesting to observe, that our method is less sensitive to the choice of initialization type – it achieves the same quality of the final solutions no matter how the population is initialized. Such robustness, that is not observed in case of CEL + HoF, can be viewed as another benefit of this approach.

Since we found quite unexpected the fact that it is favorable to initialize the entire population with the same strategies, we tried to explain it by examining in detail the evolution of Othello strategies. Figures 6.10 and 6.11 show how the board evaluation function changes throughout the learning by TDL + CEL + HoF method, for different initialization types. Although the final results in both figures are almost identical, 0-init leads to obtaining solution similar to the last one much faster than random-init. This corresponds to differences in the convergence speed of these methods. Another important remark is that the key characteristic of a well-playing WPC strategy for this problem are small weight values for the majority of locations on the board. The 0-initialized strategy naturally meets this requirement, in contrast to the random one which has larger weight variation. Figure 6.11 illustrates that the evolutionary effort during the learning process is focused exactly on reducing weight values for the middle locations.

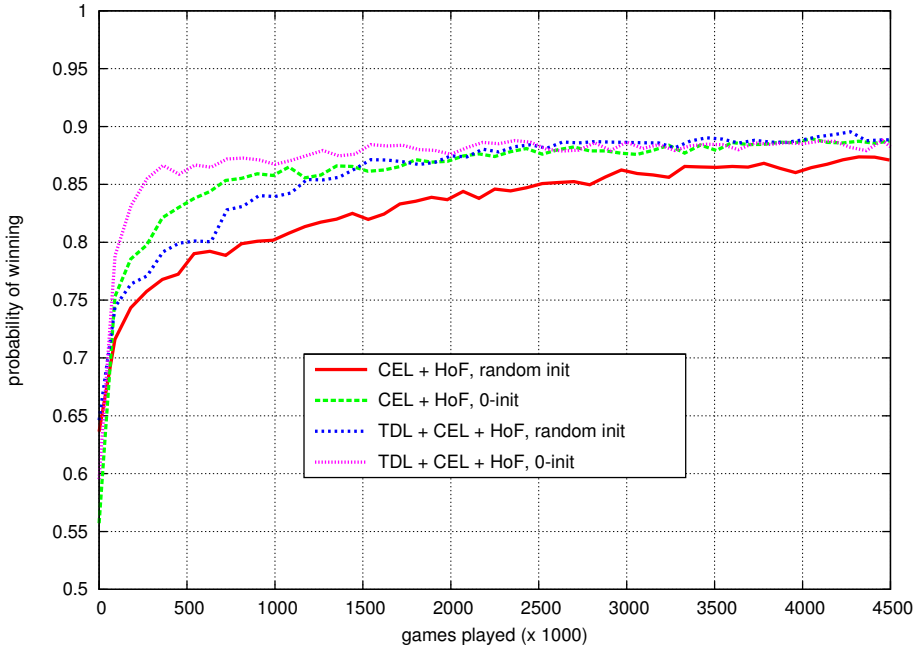


Fig. 6.8: Average performance of the best-of-generation individuals found by TDL + CEL + HoF and CEL + HoF methods for different initialization procedures, measured as a probability of winning against a **random** player, plotted against the number of games played.

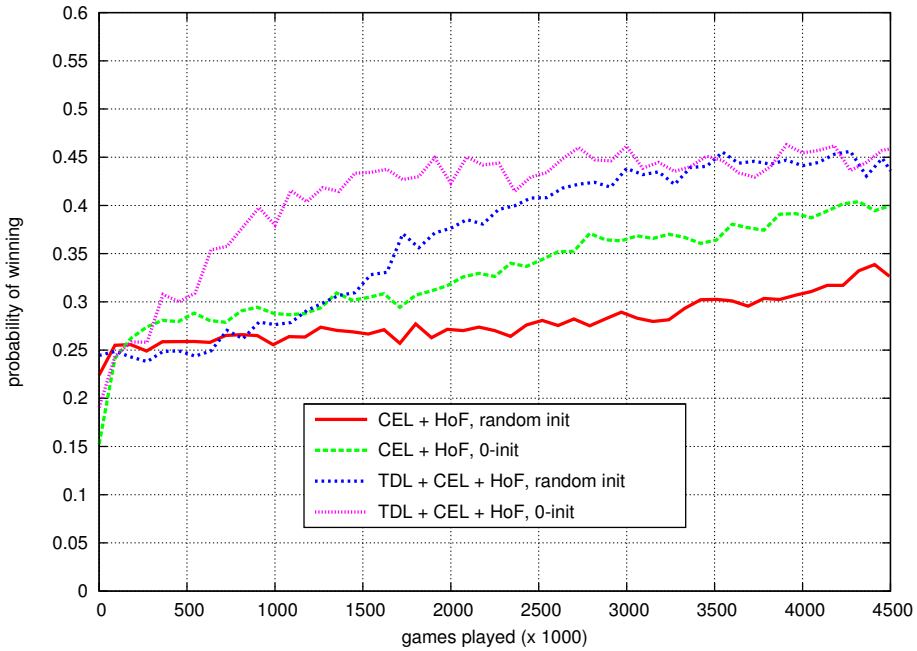


Fig. 6.9: Average performance of the best-of-generation individuals found by TDL + CEL + HoF and CEL + HoF methods for different initialization procedures, measured as a probability of winning against a standard **heuristic** player (both players were disturbed by random moves with probability equal to 0.1), plotted against the number of games played.

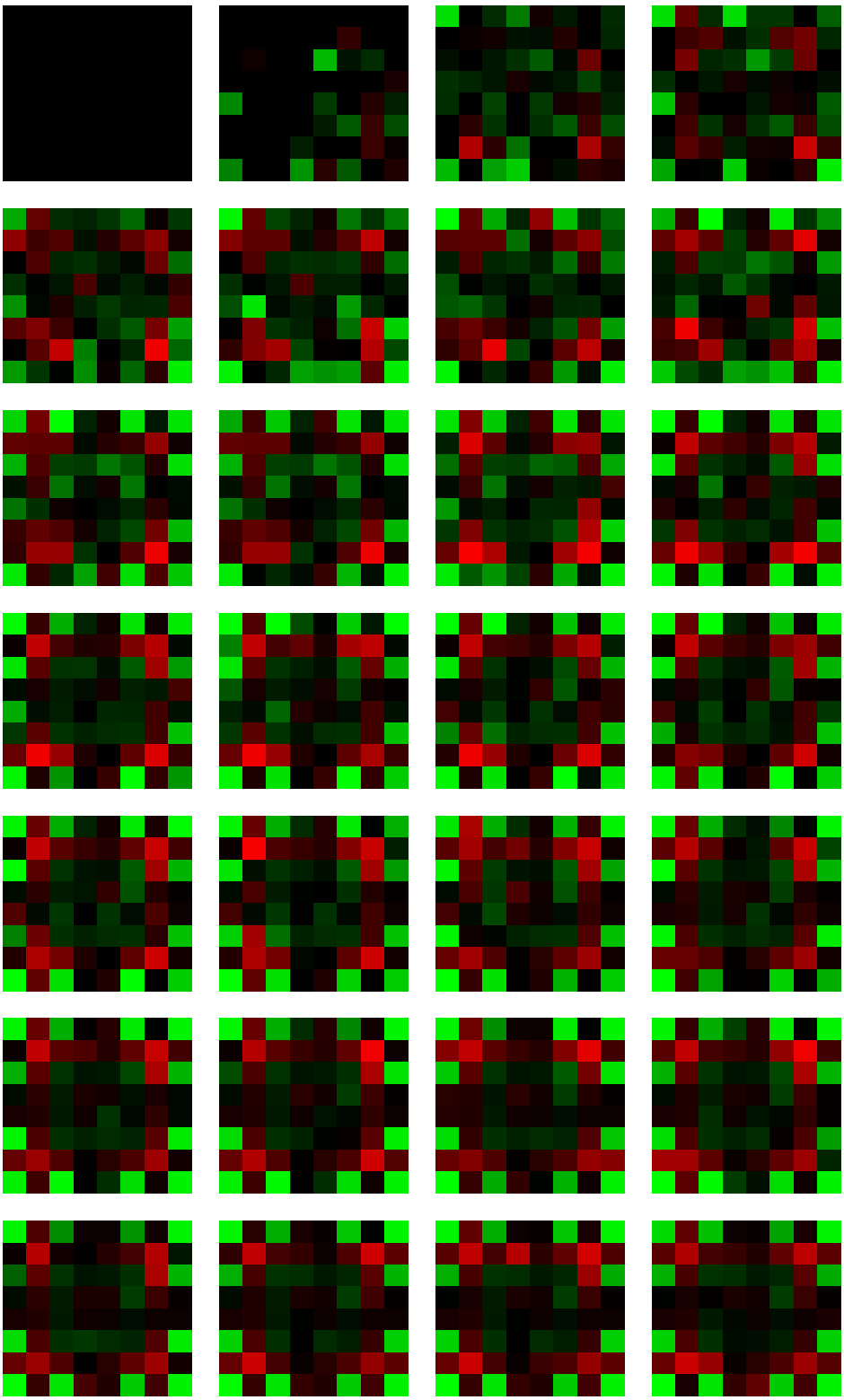


Fig. 6.10: Evolution of **zero-initialized** individual illustrated as a sequence of Othello boards (one drawing per 30 generations) colored accordingly to corresponding WPC weights (light red $= -1$, light green $= 1$, black $= 0$).

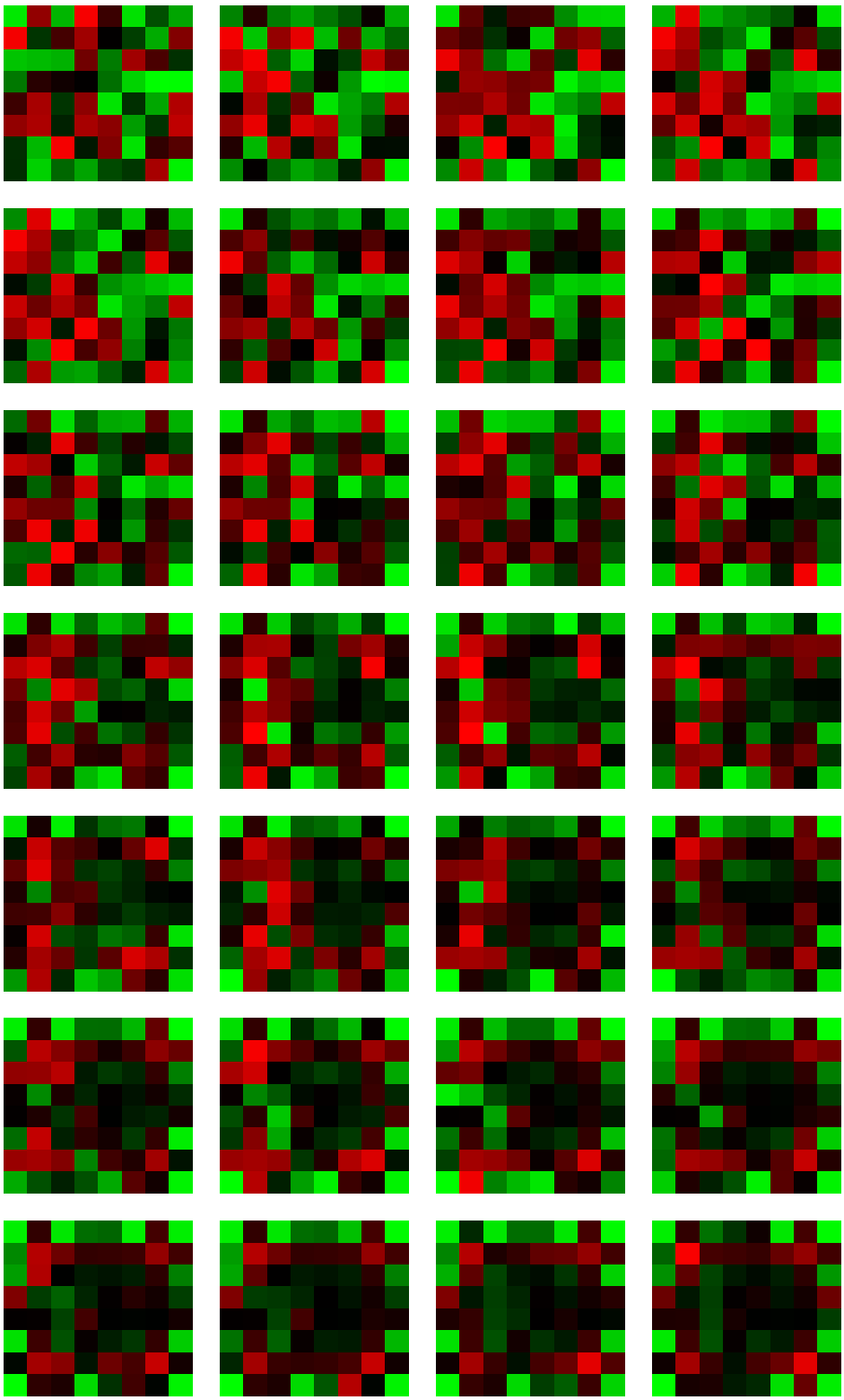


Fig. 6.11: Evolution of **randomly initialized** individual illustrated as a sequence of Othello boards (one drawing per 30 generations) colored accordingly to corresponding WPC weights (light red $= -1$, light green $= 1$, black $= 0$).

Chapter 7
Summary and Conclusions

In this study we presented a novel method of machine learning that hybridizes coevolutionary search with reinforcement learning. This approach is particularly interesting because of its unsupervised nature that makes it useful when the knowledge of the problem domain is unavailable or expensive to obtain. We introduced the Coevolutionary Temporal Difference Learning algorithm and drew an analogy to Lamarckian evolution theory that explains its biological inspirations. The algorithm was applied to the exemplary problem of learning Othello-playing strategy and compared with previous approaches to this problem.

The experimental results demonstrate that the proposed hybrid algorithm is capable of exploiting mutually complementary characteristics of both constituent methods. The evolved learners achieve better performance in the long run and the convergence of the learning process is faster. It is interesting to note that the constructed Othello strategies reveal also basic "understanding" of the game, such as the importance of the board corners. We investigated in detail the initialization issue which plays an important role in the context of this specific problem.

Additionally, another benefit of this work is development of the co-Evolutionary Computation in Java library which was thoroughly documented and tested. The library is composed of the most important modern coevolutionary methods including coevolutionary archives for different solution concepts and competitive fitness sharing. Moreover, it was designed to be easy to extend and to allow flexible experiment definition. The software integrates with a well-known ECJ system and, thereby, may be helpful for many ECJ users.

7.1 Future Work

There are many aspects of the proposed approach that are worth further investigation. These aspects concern the tuning of the algorithm for the particular problem of learning Othello strategy, as well as extensions of the coevolutionary reinforcement learning approach that may be useful in general. Let us point out a few possible directions of the future work:

- The simplest WPC strategy representation seems to limit the learning process for this problem. Therefore, it would be useful to employ a more complex architecture such as, for instance, a multi-layer nonlinear neural network.
- Apart from initialization, the details concerning other evolutionary process stages may have also a great impact on the quality of the evolved Othello strategies. Interesting possibilities include using geometric crossover during breeding a new population or competitive fitness sharing in the the fitness assignment stage.
- Using coevolutionary reinforcement learning with two-population coevolution learner-teacher paradigm, with solutions and tests bred separately, would open the possibility of using more advanced archive methods such as LAPCA and IPCA [de Jong 07] and potentially obtaining better results.
- From the Lamarckian evolution perspective, our reinforcement learning procedure simulates learning that occurs throughout an organism's lifetime and accordingly modifies the genotype to pass the acquired traits on to offspring. An alternative idea is to perform such learning solely in order to change the phenotype – this would influence the evaluation process only. Clearly, this idea is consistent with the Baldwin effect described in Section 3.3.3.
- As we discussed in Section 3.3.1, the aims of coevolutionary and temporal difference learning phases are slightly conflicting. As a result, the CEL phase may damage the effort of the preceding TDL phase. We reduce this clash by setting an appropriate TDL intensity, but another idea is to perform some sort of scaling procedure after each TDL-CEL cycle. This procedure should normalize the weight values to make them compatible with the prediction function worked out by TDL.

In summary, this work shows the potential of coevolutionary reinforcement learning method and points to the need of its further investigation in the context of other challenging problems.

Appendix A
DVD Content

The DVD attached to this thesis contains:

- The software environment described in Chapters 4 and 5.
- Raw results of the experiments described in Chapter 6.
- The PDF version of this thesis.

Appendix B

Sample Parameter Files

B.1 Othello Single Population Coevolution

```
verbosity = 0
breedthreads = 1
evalthreads = 1
seed.0 = 2009
print-unused-params = true

pop = ec.Population
state = ec.simple.SimpleEvolutionState
init = ec.simple.SimpleInitializer
finish = ec.simple.SimpleFinisher
breed = ec.simple.SimpleBreeder
stat  = ec.simple.SimpleStatistics
exch = ec.simple.SimpleExchanger
eval = cecj.eval.SimpleCoevolutionaryEvaluator

generations = 1800
checkpoint = false
prefix = ec
checkpoint-modulo = 1

pop.subpops = 1
pop.subpop.0 = ec.Subpopulation
pop.subpop.0.size  = 50
pop.subpop.0.duplicate-retries  = 0

pop.subpop.0.species  = ec.vector.FloatVectorSpecies
pop.subpop.0.species.ind = ec.vector.DoubleVectorIndividual
pop.subpop.0.species.fitness  = ec.simple.SimpleFitness
pop.subpop.0.species.genome-size = 64
pop.subpop.0.species.min-gene = -1
pop.subpop.0.species.max-gene = 1
pop.subpop.0.species.crossover-type = one
pop.subpop.0.species.mutation-prob = 0.03
pop.subpop.0.species.mutation-type = gauss
```

```
pop.subpop.0.species.mutation-stdev = 0.25

pop.subpop.0.species.pipe = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.0 = ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.source.0 = ec.select.TournamentSelection
pop.subpop.0.species.pipe.source.0.source.1 = ec.select.TournamentSelection
pop.subpop.0.species.pipe.source.0.source.0.size  = 5
pop.subpop.0.species.pipe.source.0.source.1.size  = 5

eval.problem = cecj.app.othello.Othello
eval.interaction-scheme = cecj.interaction.IntraPopulationInteractionScheme
eval.subpop.0.sampling-method = cecj.sampling.AllSamplingMethod
eval.subpop.0.fitness-method = cecj.fitness.SimpleSumFitness

stat.file = $out.stat
stat.num-children  = 2
stat.child.0  = cecj.statistics.AverageObjectiveFitnessStatistics
stat.child.0.fitness-file = $fitness_random.stat
stat.child.0.fitness-calc  = cecj.app.othello.OthelloRandomPlayer2
stat.child.0.fitness-calc.play-both  = true
stat.child.0.fitness-calc.repeats   = 500
stat.child.0.frequency = 36

stat.child.1  = cecj.statistics.AverageObjectiveFitnessStatistics
stat.child.1.fitness-file = $fitness_heuristic.stat
stat.child.1.fitness-calc  = cecj.app.othello.OthelloHeuristicPlayer
stat.child.1.fitness-calc.play-both  = true
stat.child.1.fitness-calc.repeats   = 500
stat.child.1.fitness-calc.evaluated-randomness = 0.1
stat.child.1.fitness-calc.evaluator-randomness = 0.1
stat.child.1.frequency = 36
```

B.2 Othello Coevolution with Archive

```
verbosity = 0
breedthreads = 1
evalthreads = 1
seed.0 = 2009

pop = ec.Population
state = ec.simple.SimpleEvolutionState
init = ec.simple.SimpleInitializer
finish = ec.simple.SimpleFinisher
breed = ec.simple.SimpleBreeder
stat  = ec.simple.SimpleStatistics
exch = ec.simple.SimpleExchanger
eval = cecj.eval.ArchivingCoevolutionaryEvaluator

generations = 900
checkpoint = false
prefix = ec
checkpoint-modulo = 1

pop.subpops = 1
pop.subpop.0 = cecj.archive.ArchivingSubpopulation
pop.subpop.0.size  = 50
pop.subpop.0.duplicate-retries  = 0

pop.subpop.0.species  = ec.vector.FloatVectorSpecies
pop.subpop.0.species.ind = ec.vector.DoubleVectorIndividual
pop.subpop.0.species.fitness  = ec.simple.SimpleFitness
pop.subpop.0.species.genome-size = 64
pop.subpop.0.species.min-gene = -1
pop.subpop.0.species.max-gene = 1
pop.subpop.0.species.crossover-type = one
pop.subpop.0.species.mutation-prob = 0.03
pop.subpop.0.species.mutation-type = gauss
pop.subpop.0.species.mutation-stdev = 0.25

pop.subpop.0.species.pipe = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.0 = ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.source.1 = ec.select.TournamentSelection
pop.subpop.0.species.pipe.source.0.source.1.size  = 5

pop.subpop.0.species.pipe.source.0.source.0 = ec.select.MultiSelection
pop.subpop.0.species.pipe.source.0.source.0.num-selects = 2
pop.subpop.0.species.pipe.source.0.source.0.select.0 = cecj.archive.ArchiveRandomSelection
pop.subpop.0.species.pipe.source.0.source.0.select.0.prob = 0.2
pop.subpop.0.species.pipe.source.0.source.0.select.0.size = 50
pop.subpop.0.species.pipe.source.0.source.0.select.1 = ec.select.TournamentSelection
pop.subpop.0.species.pipe.source.0.source.0.select.1.prob = 0.8
pop.subpop.0.species.pipe.source.0.source.0.select.1.size = 5
```

```
eval.problem = cecj.app.othello.Othello
eval.archive = cecj.archive.BestOfGenerationArchive
eval.interaction-scheme = cecj.interaction.IntraPopulationInteractionScheme
eval.subpop.0.sampling-method = cecj.sampling.AllSamplingMethod
eval.subpop.0.fitness-method = cecj.fitness.SimpleSumFitness
eval.subpop.0.archive-sampling-method  = cecj.sampling.RandomSamplingMethod
eval.subpop.0.archive-sampling-method.sample-size = 50

stat.file = $out.stat
stat.num-children  = 2
stat.child.0  = cecj.statistics.AverageObjectiveFitnessStatistics
stat.child.0.fitness-file = $fitness_random.stat
stat.child.0.fitness-calc  = cecj.app.othello.OthelloRandomPlayer2
stat.child.0.fitness-calc.play-both  = true
stat.child.0.fitness-calc.repeats   = 500
stat.child.0.frequency = 18

stat.child.1  = cecj.statistics.AverageObjectiveFitnessStatistics
stat.child.1.fitness-file = $fitness_heuristic.stat
stat.child.1.fitness-calc  = cecj.app.othello.OthelloHeuristicPlayer
stat.child.1.fitness-calc.play-both  = true
stat.child.1.fitness-calc.repeats   = 500
stat.child.1.fitness-calc.evaluated-randomness = 0.1
stat.child.1.fitness-calc.evaluator-randomness = 0.1
stat.child.1.frequency = 18
```

B.3 Othello Coevolutionary TD Learning

```
verbosity = 0
breedthreads = 1
evalthreads = 1
seed.0 = 1987

pop = ec.Population
state = ec.simple.SimpleEvolutionState
init = ec.simple.SimpleInitializer
finish = ec.simple.SimpleFinisher
breed = ec.simple.SimpleBreeder
stat  = ec.simple.SimpleStatistics
exch = ec.simple.SimpleExchanger
eval = cecj.eval.TDLImprovingEvaluator

generations = 1500
checkpoint = false
prefix = ec
checkpoint-modulo = 1

pop.subpops = 1
pop.subpop.0 = ec.Subpopulation
pop.subpop.0.size  = 50
pop.subpop.0.duplicate-retries  = 0

pop.subpop.0.species  = ec.vector.FloatVectorSpecies
pop.subpop.0.species.ind = ec.vector.DoubleVectorIndividual
pop.subpop.0.species.fitness  = ec.simple.SimpleFitness
pop.subpop.0.species.genome-size = 64
pop.subpop.0.species.min-gene = -1
pop.subpop.0.species.max-gene = 1
pop.subpop.0.species.crossover-type = one
pop.subpop.0.species.mutation-prob = 0.03
pop.subpop.0.species.mutation-type = gauss
pop.subpop.0.species.mutation-stdev = 0.25

pop.subpop.0.species.pipe = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.0 = ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.source.0 = ec.select.TournamentSelection
pop.subpop.0.species.pipe.source.0.source.1 = ec.select.TournamentSelection
pop.subpop.0.species.pipe.source.0.source.0.size  = 5
pop.subpop.0.species.pipe.source.0.source.1.size  = 5

eval.problem = cecj.app.othello.Othello
eval.inner-evaluator = cecj.eval.SimpleCoevolutionaryEvaluator
eval.inner-evaluator.problem = cecj.app.othello.Othello
eval.inner-evaluator.interaction-scheme = cecj.interaction.IntraPopulationInteractionScheme
eval.inner-evaluator.subpop.0.sampling-method = cecj.sampling.AllSamplingMethod
eval.inner-evaluator.subpop.0.fitness-method = cecj.fitness.SimpleSumFitness
```

```
eval.tdl-improver = cecj.app.othello.OthelloTDLImprover
eval.tdl-improver.repeats  = 10
eval.tdl-improver.randomness  = 0.1
eval.tdl-improver.learning-rate = 0.01

stat.file = $out.stat
stat.num-children  = 2
stat.child.0  = cecj.statistics.AverageObjectiveFitnessStatistics
stat.child.0.fitness-file = $fitness_random.stat
stat.child.0.fitness-calc  = cecj.app.othello.OthelloRandomPlayer2
stat.child.0.fitness-calc.play-both  = true
stat.child.0.fitness-calc.repeats   = 500
stat.child.0.frequency = 24

stat.child.1  = cecj.statistics.AverageObjectiveFitnessStatistics
stat.child.1.fitness-file = $fitness_heuristic.stat
stat.child.1.fitness-calc  = cecj.app.othello.OthelloHeuristicPlayer
stat.child.1.fitness-calc.play-both  = true
stat.child.1.fitness-calc.repeats   = 500
stat.child.1.fitness-calc.evaluated-randomness = 0.1
stat.child.1.fitness-calc.evaluator-randomness = 0.1
stat.child.1.frequency = 24
```

Bibliography

[Angeline 93] Peter J. Angeline & Jordan B. Pollack. *Competitive Environments Evolve Better Solutions for Complex Tasks*. In ICGA, pages 264–270, 1993.

[Bäck 97a] Thomas Bäck, David B. Fogel & Zbigniew Michalewicz. Handbook of evolutionary computation. IOP Publishing Ltd., Bristol, UK, UK, 1997.

[Bäck 97b] Thomas Bäck, Ulrich Hammel & Hans paul Schwefel. *Evolutionary Computation: Comments on the History and Current State*. IEEE Transactions on Evolutionary Computation, vol. 1, pages 3–17, 1997.

[Baldwin 96] Mark J. Baldwin. *A New Factor In Evolution*. American Naturalist, vol. 30, pages 441–457, 1896.

[Bar-Cohen 05] Y. Bar-Cohen. Biomimetics - biologically inspired technologies. CRC Press, 2005.

[Bucci 04] Anthony Bucci, Jordan B. Pollack & Edwin de Jong. *Automated Extraction of Problem Structure*. In Genetic and Evolutionary Computation–GECCO 2004. Proceedings of the Genetic and Evolutionary Computation Conference. Part I, pages 501–512, Seattle, Washington, USA, 2004. Springer-Verlag, Lecture Notes in Computer Science Vol. 3102.

[Bucci 07] Anthony Bucci. *Emergent geometric organization and informative dimensions in coevolutionary algorithms*. PhD thesis, Waltham, MA, USA, 2007.

[Burkhardt 77] R.W. Burkhardt. The spirit of system: Lamarck and evolutionary biology. Cambridge, MA: Harvard University Press, 1977.

[Buro 95] M. Buro. *Logistello: A Strong Learning Othello Program.* In 19th Annual Conference Gesellschaft für Klassifikation e.V., 1995.

[Buro 97] Michael Buro. *Takeshi Murakami vs. Logistello*, 1997.

[Buro 02] Michael Buro. *Improving Heuristic Mini-Max Search by Supervised Learning.* Artificial Intelligence, vol. 134, pages 85–99, 2002.

[Carroll 87] Lewis Carroll. Through the looking-glass. Plain Label Books, 1887.

[Caverlee 00] James B. Caverlee. *A Genetic Algorithm Approach to Discovering an Optimal Blackjack Strategy.* In John R. Koza, editeur, Genetic Algorithms and Genetic Programming at Stanford 2000, pages 70–79. Stanford Bookstore, Stanford, California, 94305-3079 USA, June 2000.

[Chomsky 93] Noam Chomsky. Language and thought. Moyer Bell, 1993.

[Coello 98] Carlos A. Coello Coello. *A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques.* Knowledge and Information Systems, vol. 1, pages 269–308, 1998.

[Darwin 59] C. Darwin. On the origin of species by means of natural selection. John Murray, London, UK, 1859.

[Dawkins 79] Richard Dawkins & J. R. Krebs. *Arms Races between and within Species.* Proceedings of the Royal Society of London, Series *B*, vol. 205, pages 489–511, 1979.

[de Jong 04a] E.D. de Jong. *Towards a bounded Pareto-coevolution archive.* In Evolutionary Computation, 2004. CEC2004. Congress on, 2004.

[de Jong 04b] Edwin D. de Jong. *The Incremental Pareto-Coevolution Archive.* Evolutionary Computation, vol. 12, no. 2, pages 525–536, 2004.

[de Jong 04c] Edwin D. de Jong & Jordan B. Pollack. *Ideal Evaluation from Coevolution.* Evolutionary Computation, vol. 12, no. 2, pages 159–192, 2004.

[de Jong 05] Edwin D. de Jong. *The MaxSolve algorithm for coevolution.* In GECCO, pages 483–489, 2005.

[de Jong 07] Edwin D. de Jong. *A Monotonic Archive for Pareto-Coevolution.* Evolutionary Computation, vol. 15, no. 1, pages 61–93, 2007.

[Dozier 98] G. Dozier, J. Bowen & A. Homaifar. *Solving Constraint Satisfaction Problems Using Hybrid Evolutionary Search.* IEEE Transactions on Evolutionary Computation, vol. 2(1), pages 23–32, 1998.

[Eiben 03] Agoston E. Eiben & J. E. Smith. Introduction to evolutionary computing. SpringerVerlag, 2003.

[Epstein 94] Susan L. Epstein. *Toward an ideal trainer.* In Machine Learning, pages 251–277, 1994.

[Ficici 98] Sevan G. Ficici & 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.

[Ficici 04] Sevan Gregory Ficici. *Solution concepts in coevolutionary algorithms.* PhD thesis, Waltham, MA, USA, 2004. Adviser-Jordan B. Pollack.

[Ficici 08] Sevan Gregory Ficici. *Multiobjective Optimization and Coevolution.* In Multiobjective Problem Solving from Nature, pages 31–52. Springer Berlin Heidelberg, 2008.

[Floreano 08] Dario Floreano & Claudio Mattiussi. Bio-inspired artificial intelligence: Theories, methods, and technologies. The MIT Press, 2008.

[Fogel 95] David B. Fogel & Lawrence J. Fogel. *An Introduction to Evolutionary Programming.* In Artificial Evolution, pages 21–33, 1995.

[Fogel 02] David B. Fogel. Blondie24: playing at the edge of ai. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson & John M. Vlissides. Design patterns: Elements of reusable object-oriented software. Addison-Wesley Professional Computing Series, 1995.

[Gause 34] G. F. Gause. The struggle for existence. Hafner, 1934.

[Goldberg 89] David E. Goldberg. Genetic algorithms in search, optimization, and machine learning. Addison-Wesley Professional, 1989.

[Grosan 07] Crina Grosan & Ajith Abraham. Hybrid evolutionary algorithms: Methodologies, architectures, and reviews. Springer, 2007.

[Hillis 92] D. Hillis. *Co-evolving parasites improves simulated evolution as an optimization procedure.* Artificial Life II, 1992.

[Holland 62] John H. Holland. *Outline for a Logical Theory of Adaptive Systems.* J. ACM, vol. 9, no. 3, pages 297–314, 1962.

[Jaśkowski 07] Wojciech Jaśkowski, Krzysztof Krawiec & Bartosz Wieloch. *BrilliANT: The Winner Entry of the GECCO'2007 Ant Wars Contest.* Rapport technique RA-05/07, 2007.

[Juillé 96] Hugues Juillé & Jordan B. Pollack. *Co-Evolving Intertwined Spirals.* In Evolutionary Programming, pages 461–468, 1996.

[Juillé 98] Hugues Juillé & Jordan B. Pollack. *Coevolutionary Learning: A Case Study.* In ICML, pages 251–259, 1998.

[Kaelbling 96] Leslie Pack Kaelbling, Michael Littman & Andrew Moore. *Reinforcement Learning: A Survey.* Journal of Artificial Intelligence Research, vol. 4, pages 237–285, 1996.

[Katayama 00] K. Katayama, H. Sakamoto & H. Narihisa. *The Efficiency of Hybrid Mutation Genetic Algorithm for the Travelling Salesman Problem.* Mathematical and Computer Modelling, vol. 31, pages 197–203, 2000.

[Kim 07a] Kyung-Joong Kim & Sung-Bae Cho. *Evolutionary Algorithms for Board Game Players with Domain Knowledge.* In Advanced Intelligent Paradigms in Computer Games, pages 71–89. Springer. Studies in Computational Intelligence, Vol. 71., 2007.

[Kim 07b] Kyung-Joong Kim, Heejin Choi & Sung-Bae Cho. *Hybrid of Evolution and Reinforcement Learning for Othello Players.* Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on, pages 203–209, 2007.

[Koza 92] John R. Koza. Genetic programming: On the programming of computers by means of natural selection. MIT Press, Cambridge, MA, USA, 1992.

[Lee 90] Kai-Fu Lee & Sanjoy Mahajan. *The development of a world class Othello program.* Artif. Intell., vol. 43, no. 1, pages 21–36, 1990.

[Leouski 95] Anton Leouski. *Learning of Position Evaluation in the Game of Othello.* Rapport technique, 1995.

[Lubberts 01] Alex Lubberts & Risto Miikkulainen. *Co-Evolving a Go-Playing Neural Network.* In Richard K. Belew & Hugues Juillè, editeurs, Coevolution: Turning Adaptive Algorithms upon Themselves, pages 14–19, San Francisco, California, USA, 7 July 2001.

[Lucas 06] Simon M. Lucas & Thomas Philip Runarsson. *Temporal Difference Learning Versus Co-Evolution for Acquiring Othello Position Evaluation.* In CIG, pages 52–59, 2006.

[Luke 08] Sean Luke. *ECJ 18 - A Java-based Evolutionary Computation Research System.* http://cs.gmu.edu/~eclab/projects/ecj/, 2008.

[Manning 07] Edward P. Manning. *Temporal Difference Learning of an Othello Evaluation Function for a Small Neural Network with Shared Weights.* 2007.

[Margulis 02] Lynn Margulis & Dorion Sagan. Acquiring genomes: A theory of the origins of species. HarperCollins, 2002.

[Miconi 08] Thomas Miconi. *The Road to Everywhere: Evolution, Complexity and Progress in Natural and Artificial Systems.* PhD thesis, University of Birmingham, 2008.

[Miconi 09] Thomas Miconi. *Why Coevolution Doesn't "Work": Superiority and Progress in Coevolution.* In EuroGP 2009, 2009.

[Miller 94] Geoffrey F. Miller & Dave Cliff. *Protean behavior in dynamic games: arguments for the co-evolution of pursuit-evasion tactics.* In SAB94: Proceedings of the third international conference on Simulation of adaptive behavior : from animals to animats 3, pages 411–420, Cambridge, MA, USA, 1994. MIT Press.

[Mitchell 97] Thomas M. Mitchell. Machine learning. McGraw-Hill, 1997.

[Monroy 06] German A. Monroy, Kenneth O. Stanley & Risto Miikkulainen. *Coevolution of neural networks using a layered pareto archive.* In GECCO, pages 329–336, 2006.

[Moriarty 97] David E. Moriarty, Alan C. Shultz & John J. Grefenstette. *Reinforcement Learning through Evolutionary Computation.* 1997.

[Neumann 58] John von Neumann. The computer and the brain. Yale University Press, 1958.

[Nolfi 98] Stefano Nolfi & Dario Floreano. *Coevolving Predator and Prey Robots: Do "Arms Races" Arise in Artificial Evolution?* Artificial Life, vol. 4, no. 4, pages 311–335, 1998.

[Pagie 97] Ludo Pagie & Paulien Hogeweg. *Evolutionary Consequences of Coevolving Targets.* Evolutionary Computation, vol. 5, no. 4, pages 401–418, 1997.

[Panait 02] Liviu Panait & Sean Luke. *A Comparison Of Two Competitive Fitness Functions.* In GECCO, pages 503–511, 2002.

[Paredis 97] Jan Paredis. *Coevolving Cellular Automata: Be Aware of the Red Queen.* In Thomas Bäck, editeur, Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97), San Francisco, CA, 1997. Morgan Kaufmann.

[Parlett 99] David Parlett. Oxford history of board games. Oxford University Press, 1999.

[Pollack 98] Jordan B. Pollack & Alan D. Blair. *Co-Evolution in the Successful Learning of Backgammon Strategy.* Machine Learning, vol. 32, no. 3, pages 225–240, 1998.

[Potter 00] M. Potter & K. De Jong. *Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents.* Evolutionary Computation, vol. 8, no. 1, pages 1–29, 2000.

[Price 96] P. W. Price. Biological evolution. Saunders College Publishing, Philadelphia, PA, 1996.

[Rechenberg 73] Ingo Rechenberg. Evolutionsstrategie. Frommann-Holzboog, 1973.

[Rosin 95] Christopher D. Rosin & Richard K. Belew. *Methods for Competitive Co-Evolution: Finding Opponents Worth Beating.* In ICGA, pages 373–381, 1995.

[Rosin 97] Christopher D. Rosin & Richard K. Belew. *New Methods for Competitive Coevolution.* Evolutionary Computation, vol. 5, no. 1, pages 1–29, 1997.

[Runarsson 05] Thomas P. Runarsson & Simon Lucas. *Co-evolution versus Self-play Temporal Difference Learning for Acquiring Position Evaluation in Small-Board Go.* IEEE Transactions on Evolutionary Computation, vol. 9, 2005.

[Runarsson 07] Thomas Philip Runarsson & Egill Orn Jonsson. *Effect of look-ahead search depth in learning position evaluation functions for Othello.* In IEEE Computational Intelligence and Games, pages 210–215, 2007.

[Samuel 59] Arthur L. Samuel. *Some studies in machine learning using the game of checkers.* IBM Journal of Research and Development, vol. 44, no. 1, pages 206–227, 1959.

[Shannon 50] Claude E. Shannon. *Programming a computer for playing chess.* Philosophical Magazine, vol. 41, pages 256–275, 1950.

[Sims 94a] Karl Sims. *Evolving 3D Morphology and Behaviour by Competition.* In R. Brooks & P. Maes, editeurs, Artificial Life IV Proceedings, pages 28–39, MIT, Cambridge, MA, USA, 1994. MIT Press.

[Sims 94b] Karl Sims. *Evolving virtual creatures.* In SIGGRAPH, pages 15–22, 1994.

[Singer 01] Joshua A. Singer. *Co-evolving a Neural-Net Evaluation Function for Othello by Combining Genetic Algorithms and Reinforcement Learning.* In International Conference on Computational Science (2), pages 377–389, 2001.

[Stanley 05] Kenneth O. Stanley, Bobby D. Bryant & R. Miikkulainen. *Real-time neuroevolution in the NERO video game.* Evolutionary Computation, IEEE Transactions on, vol. 9, no. 6, pages 653–668, 2005.

[Sutton 88] Richard S. Sutton. *Learning to Predict by the Methods of Temporal Differences.* Machine Learning, vol. 3, pages 9–44, 1988.

[Sutton 98] Richard S. Sutton & Andrew G. Barto. Reinforcement learning: An introduction. The MIT Press, 1998.

[Tesauro 95] Gerald Tesauro. *Temporal difference learning and TD-Gammon.* Commun. ACM, vol. 38, no. 3, pages 58–68, 1995.

[Van Valen 73] Leigh Van Valen. *A new evolutionary law.* Evolutionary Theory, vol. 1, no. 1, pages 1–30, 1973.

[Viswanathan 05] Shivakumar Viswanathan. *On The Coevolutionary Construction Of Learnable Gradients*, 2005.

[Watson 01] R. A. Watson & J. B. Pollack. *Coevolutionary Dynamics in a Minimal Substrate.* In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001), pages 702–709, 2001.

[Weise 09] Thomas Weise. Global optimization algorithms - theory and application. 2009.

[Whitley 94] D. Whitley, V.S. Gordon & K. Mathias. *Lamarckian Evolution, the Baldwin Effect and Function Optimization.* In Y. Davidor, H.-P. Schwefel & R. Maenner, editeurs, Proc. Third International Conference on Parallel Problem Solving from Nature (PPSN),

volume 866 of *Lecture Notes in Computer Science*, pages 6–15, New York, 1994. Springer-Verlag.

[Wiegand 01] R. Paul Wiegand, William Liles & Kenneth De Jong. *An Empirical Analysis of Collaboration Methods in Cooperative Coevolutionary Algorithms.* pages 1235–1242, 2001.

[Zufferey 08] Jean-Christophe Zufferey. Bio-inspired flying robots: experimental synthesis of autonomous indoor flyers. CRC Press, 2008.