UNIVERSITY OF CALIFORNIA, SAN DIEGO

Coevolutionary Search Among Adversaries

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Christopher Darrell Rosin

Committee in charge:

Professor Richard K. Belew, Chairperson
Professor Ramamohan Paturi
Professor Garrison W. Cottrell
Professor John D. Batali
Professor J. Andrew McCammon

1997

The dissertation of Christopher Darrell Rosin is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

1997

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Thanks to my advisor Rik Belew, who is a collaborator on all of this work. Section IV.D is joint work with David Goodsell, Art Olson, and Garrett Morris of the Scripps Research Institute.

Thanks to Mark Land, Thomas Kammeyer, Filippo Menczer, Mohan Paturi, and CCSRG for helpful discussions about this work. Thanks to Apple Computer, NSF, and ACISE for financial and computational support.

Thanks also to my family, Dan Clouse, Cindy Deutsch, Hal Duncan, B.J.'s, BCSD, and everyone else on my support team.

VITA

September 20, 1971	Born, Milwaukee, Wisconsin
1992	B.S., California Institute of Technology
1994	M.S., University of California, San Diego
1995-1997	Teaching Assistant, Department of Computer Science and Engineering, University of California, San Diego
1997	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

Rosin, C.D., Halliday, R.S., Hart, W.E., and Belew, R.K. (1997). "A Comparison of Global and Local Search Methods in Drug Docking." Proceedings of the Seventh International Conference on Genetic Algorithms.

Rosin, C.D. and Belew, R.K. (1996). "New Methods for Competitive Coevolution." Evolutionary Computation 5:1.

Rosin, C.D. and Belew, R.K. (1996). "A Competitive Approach to Game Learning." Proceedings of the Ninth Annual ACM Conference on Computational Learning Theory.

Rosin, C.D. and Belew, R.K. (1995). "Methods for Competitive Co-evolution: Finding Opponents Worth Beating." Proceedings of the Sixth International Conference on Genetic Algorithms.

Niebur, E., Koch, C., and Rosin, C.D. (1993). "An Oscillation-Based Model for the Neuronal Basis of Attention." Vision Research 33:18.

FIELDS OF STUDY

Major Field: Computer Science

Studies in Genetic Algorithms

Professor Richard K. Belew

ABSTRACT OF THE DISSERTATION

Coevolutionary Search Among Adversaries

by

Christopher Darrell Rosin

Doctor of Philosophy in Computer Science

University of California, San Diego, 1997

Professor Richard K. Belew, Chair

Competitive coevolution is a biologically-inspired search technique that uses a genetic algorithm in two competing populations. During search, individuals in each population are evaluated by direct competition against members of the opposing population. A class of *adversarial problems* is defined, to which competitive coevolution can be applied. In such problems, it is important to actively choose difficult test cases to accurately evaluate candidate solutions. Competitive coevolution evaluates candidate solutions and test cases against one another while searching for both, so that each drives the improvement of the other.

Tools from computational learning theory are used to obtain positive theoretical results on the efficiency of simplified coevolutionary algorithms. Lower bounds show the importance of several parameters for describing performance of these algorithms. This work gives insight into necessary and sufficient conditions for successful coevolutionary search.

Experimental work uses competitive coevolution as a heuristic method for adversarial problems. Several measurement techniques are used to understand the behavior of coevolution. These allow identification of several flaws in simple forms of coevolution. Coevolution can be stalled by the loss of important niches, by coevolutionary cycles, or by a lack of balance between populations. New methods are described that overcome these flaws and make coevolution more efficient. Competitive fitness sharing preserves important niches, the hall of fame encourages long-term progress, the phantom parasite maintains balance, and shared sampling allows efficient fitness testing. With these new methods, coevolution is able to solve several game learning test problems that cannot be efficiently solved without them.

Several applications are discussed. The first uses a simulation of patient blood pressure during surgery to evaluate drug delivery controllers against patients; the goal is a controller that successfully controls all possible patients. The second application uses a sorting network design problem to explore the relationship between theoretical concepts of testability and experimental coevolutionary behavior. The third application is the design of strategies for the game of Go. Coevolution produces competitive players that push the strategy representation to its limits. Finally, preliminary work is presented on the design of drugs that are robust across possible drug resistance mutations.

Chapter I

Introduction

I.A Adversarial Problems

Many problems may be formulated as the search for a *solution* that is correct over a large space of *test cases*. In such problems, accurate evaluation of candidate solutions is difficult; there are too many test cases to evaluate candidates on all of them, and a random sample of test cases can be uninformative. Instead, accurate and efficient evaluations can be obtained by actively searching for difficult test cases that expose flaws in candidate solutions.

As a concrete example, consider the problem of constructing sorting networks. A sorting network sorts sequences of fixed length with a collection of compare-exchange gates. Sorting networks need to sort any possible sequence correctly. There are, however, an exponentially large number of sequences, making exhaustive testing expensive when searching for large networks. The testing problem may be substantially reduced by searching for a smaller set of difficult-to-sort test sequences that still serve as an adequate test suite [44].

As a rather different example, consider the search for strategies in a game. Here, the problem is to discover a strategy that defeats any possible opponent, without testing exhaustively against the large space of possible opponents. Again, by searching for strong opponents, we may achieve our goal with substantially less computational effort [80].

We may give a general definition of this sort of problem: given $f(x,t)$, where f is an efficiently computable function that determines whether candidate solution x is correct on test case t, find a solution correct on all test cases. We call such problems *adversarial problems*, since $f(x,t)$ can be thought of as defining the structure of a competition between solutions and test cases.

Our general approach to adversarial problems continually searches for new candidate solutions that correctly handle test cases found by earlier search, and for new test cases that expose flaws in candidate solutions found by earlier search. Such a procedure may start with randomly chosen candidate solutions and test cases, and bootstrap its way to progressively better solutions and more difficult test cases.

I.A.1 Differences from Other Familiar Problems

Adversarial problems are fairly distinct from those attacked by familiar methods in AI. First, note that there are similarities to inductive learning problems: $f(x,t)$ in this case might describe whether x matches the target concept on example t. Typically, in inductive learning, one is given a fixed set of examples, or a source that supplies them randomly from a fixed distribution. The function $f(x,t)$ itself is not usually available (since having f would amount to knowing the target function). In this case, f cannot be used to guide a search for difficult test cases.

Optimization algorithms are designed to find points that optimize some given, efficiently computable cost function. In adversarial problems, $f(x,t)$ is efficiently computable, but to create a sufficient cost function of candidate solutions x, $f(x,t)$ would have to be computed over *all* test cases. Since this is usually infeasible, there is no way to directly apply optimization algorithms.

I.B Related Work

I.B.1 Related Problems and Methods

Classical game theory [96], the theory of differential games [64], and economic game learning theory [37] address similar problems to those described here. The focus in this literature is typically on problems simple enough to be understood analytically. When general-purpose methods for solving games are explored, they are typically efficient only for very small games. Here, we are more concerned with heuristic methods for large, difficult problems. Some of the general theory, however, may be usefully applied to adversarial problems; methods in Chapter II have some connections to this theory.

In machine learning, some of the issues in active learning [24, 25, 74] are similar to those here. Active learning tries to select informative examples from which to learn. For example, Cohn [24] makes the assumption that examples are available *without labels*. Labels may be obtained via some expensive method, so the number of labels requested is to be minimized. A version-space learner is used, and examples selected for labelling are those on which the most general and most specific remaining hypotheses disagree. No search over examples is done, though; the learner simply watches incoming examples from some distribution, and chooses which to label. In related work, Cohn shows how to choose new examples optimally in a problem-dependent way for certain simple concept representations [25]. Plutowski [74] minimizes the number of examples used for learning, but does no search over examples: instead, examples are selected from a fixed set on the basis of a criterion that tells how informative they will be.

Query learning [19] is a theoretical model that explores inductive learning problems in which the learner may obtain examples from oracles more powerful than a random example source. Query learning with a membership oracle falls into the framework outlined above, but in a specific concept-learning context: an oracle is available that tells whether hypothesis x matches the target on example t, and a hypothesis that matches the target on every example is sought. Much of computational learning theory explores concept classes that may be learned in such models. Chapter II discusses the application of computational learning theory to adversarial problems.

For simple adversarial problems, analytic solutions may be possible with specialized techniques such as combinatorial game theory [18]. Also, for reasonably small games, minimax versions of dynamic programming and tree search can be used to compute exact solutions.

In general, larger problems require heuristic methods. Heuristic forms of dynamic programming exist. These are used, for example, to train neural networks via reinforcement learning. This method has been successfully applied to backgammon [95] and othello [97]. Dynamic programming may apply when the complexity of the problem lies in its extension in time. In such problems, dynamic programming can begin by building evaluations that are accurate for states arising later in time, and work backwards to obtain accuracy for states arising earlier in time. When a problem is not extended in time, dynamic programming becomes less useful. For example, many problems require the two competing sides to make a single simultaneous choice from a large set. This happens in function induction problems such as sorting network design, where one side chooses the function and the other chooses the test point. In such problems, dynamic programming cannot be usefully applied because it has no temporal structure to work with. We seek a more general approach to adversarial problems.

Several authors have discussed general issues surrounding the proper choice of opponents for the learning of robust game strategies [32, 91, 8]. The common theme is that diverse strong opposition is needed to learn good strategies. A single expert opposing program will often be unavailable, and even if it is available it may have a simple flaw that can be exploited by an otherwise weak strategy. Random opponents tend to be far too weak to use for learning. These difficulties suggest that opponents should be chosen by search.

I.B.2 The Genetic Algorithm

The genetic algorithm (GA) is the search technique at the core of our practical computational methods for adversarial problems. When using the GA, candidate solutions to some problem are specified in a chosen representation (often finite-length bit strings). A fitness function maps candidate solutions to a numerical evaluation. The GA uses ideas from natural evolution to improve the fitness of a population of candidate solutions over successive generations.

Evolution begins with a population of randomly chosen candidate solutions. Each generation, the fitness of each member of the population is evaluated. Individuals are selected from the population for reproduction into the next generation; population members with higher fitness are more likely to be selected. After reproduction, individuals may undergo problem-dependent crossover and mutation operations. Crossover combines elements of two parents to produce a child, and mutation makes a (usually small) change to a single individual. Over successive generations, selection chooses the best novel individuals generated by crossover and mutation, so that fitness tends to improve.

Extensive work on genetic algorithms discusses many variations on this basic outline, and a wide variety of applications (for example [33, 17]). The GA has a natural coevolutionary extension to adversarial problems, described below.

I.B.3 Biological Motivation

The genetic algorithm is inspired and informed by properties of biological evolution. Similarly, coevolutionary methods for approaching adversarial problems are inspired by competitive coevolution in nature. In biological coevolution, the fitness of individuals in one species is determined by interaction with individuals from another species. From the perspective of solving adversarial problems, the most encouraging feature of biological coevolution is the evolutionary *arms race*, in which competing species are engaged in a feedback loop; an adaptive change by one creates a new challenge for the other, leading to an adaptive change on its part and so on [29].

A classic example of an evolutionary arms race occurs between carnivores and their herbivorous prey. The table below shows brain size (normalized for body size) increases in these two groups of species over a 60 million year time span, based on fossils from the northern hemisphere [47]. Similar increases have been observed for running speed (derived from bone size measurements) [13]. We have some confidence that the brain size increase is due to the pressures of competition; there were no significant predators in South America during most of this time span, and no such increase in brain size was seen.

Table I.1: Evolutionary Increase in Normalized Brain Size

Time	**Carnivores**	**Herbivores**
Archaic (60-35 million years ago)	0.44	0.18
Paleogene (35-22.5 million years ago)	0.61	0.38
Neogene (22.5-2.5 million years ago)	0.76	0.63
Recent	1.10	0.95

From the point of view of solving adversarial problems, this is encouraging. Ideally, we would like to see the two competing sides of the search drive each other towards ever-greater levels of performance, generality, and useful complexity. The heuristics described below are partially rooted in this biological motivation.

I.C Competitive Coevolution

The main work related to the methods proposed here is that on competitive coevolution, in the GA literature. The technique involves the simultaneous evolution of two populations[1], with the fitness of one based on the outcome of competition with the other. This forms the basis for the main experimental methods used in this thesis.

[1]Occasionally, an adversarial problem may be completely symmetric between solutions and test cases. In such cases, there only needs to be one population, whose members compete amongst themselves. For example, this symmetry holds in the Prisoners' Dilemma[12].

Part of the motivation for using population-based search methods to solve adversarial problems comes from biology, as described above. We envision better solutions driving the evolution of more difficult test cases, and more difficult test cases driving the evolution of better solutions. Eventually, this should bootstrap its way to perfect solutions to the adversarial problem. Another motivation is that a population can serve both as the basis for future search, and as a set of test cases. This dual use seems to make competitive coevolution a particularly efficient heuristic approach to adversarial problems.

The original work on coevolution in a GA context was done by Axelrod [12] on the Prisoners' Dilemma. Since then, a number of coevolutionary experiments have been done on this problem. The goals of these experiments are typically in modelling the emergence of cooperation, which is fairly orthogonal to the concerns of adversarial problems.

Hillis coevolved sorting networks and difficult sequences to sort [44]. Coevolution produced better solutions than random testing. We revisit this application in Section IV.B.

Coevolution can be naturally applied to finding strategies for zero-sum games. Angeline and Pollack applied a coevolutionary form of genetic programming to tic-tac-toe [8], with moderate success. We have solved this simple game with a different representation and several extensions to competitive coevolution (Chapter III). Sims has had great success coevolving simulated 3-D robots to participate in a simple zero-sum competition in a simulated environment [88]. Both controllers and structure of these robots were evolved, and a variety of complex successful strategies emerged. Several researchers have had success applying competitive coevolution to versions of the pursuer-evader differential game [23, 78]. Recently, a very simple form of coevolution has been successfully applied to backgammon [75].

Sebald and Schlenzig discuss a different approach to adversarial problems, and its application to controller design [86]. This method uses a nested evolutionary programming procedure. The outer loop searches for controllers, while the inner loop produces a fitness evaluation by searching for the worst test case for each controller. We discuss this application further in Section IV.A.

Paredis uses a steady-state coevolutionary genetic algorithm, that preserves individuals and accumulates fitness over long time scales [73]. This helps give coevolution some memory, similar to the method discussed in Section III.C.3. Paredis uses competitive coevolution to evolve solutions while actively modifying a distribution over a fixed set of test cases.

In addition, Paredis discusses "symbiotic" (or "cooperative") coevolution, in which a problem is decomposed into components that may be improved separately in distinct cooperating populations. This can play a variety of roles in decomposable optimization and machine learning tasks [76, 73]. The goals of these cooperative systems are quite different from the goals of competitive methods for adversarial problems.

I.D Categories of Applications

To computationally solve an adversarial problem, we generally must be able to evaluate a candidate solution on any chosen test case. This is a rather specialized requirement, and practical examples of adversarial problems seem to fall into a few narrow categories. The first category uses simulations to approximately evaluate real-world performance. The design of controllers that are robust across a wide range of plants uses a simulation in this way. We discuss applications to controller design (Section IV.A) and drug design (Section IV.D) that fit into this category.

A second category of applications involves the design of novel implementations of functions or other behaviors. We need to be able to test whether a candidate implementation of the function is correct on a chosen point in the domain. Given this, we can search for an implementation that is correct over the whole domain. For example, Hillis [44] used coevolution to simultaneously search for sorting networks and test sequences, to obtain an implementation of sorting that was suitable for hardware (see Section IV.B for further experiments on sorting networks).

The final category includes games and other simulated competitive environments that have value in artificial intelligence and artificial life research. Games are attractive as test problems because outcomes are efficiently computable, and the simple nature of the domains makes results easy to interpret. Despite this simplicity, perfect solutions to many games are likely to be out of reach for practical purposes. Such competitive environments hold promise for allowing open-ended adaptations that go beyond the restrictive close-ended search required in static optimization problems (see Section IV.C for experiments on the coevolution of Go strategies).

I.E Overview

Simple forms of coevolution do not always work well. Rather than continually producing stronger solutions and more difficult test cases in a progressive arms race, it can end up cycling among poor solutions and test cases. We seek a theoretical description of conditions under which coevolution-like methods will succeed or fail. For practical applications, we seek heuristic coevolutionary methods that produce reliable arms races for solving a range of adversarial problems.

Chapter II describes a theory of simplified coevolution-like processes, and gives positive results that show the bootstrapping idea behind coevolution can solve adversarial problems efficiently under the right conditions. Chapter III presents experiments using competitive coevolution as a practical heuristic method for solving adversarial problems. We use simple games as test problems, and investigate the mechanisms underlying coevolutionary performance. We show that simple forms of coevolution suffer from several problems, and describe new methods for overcoming these problems. Chapter IV describes applications of these new coevolutionary methods to robust controller design, learning of strategies for the game of Go, sorting network design, and drug design. Chapter V suggests possible implications of these computational experiments for the understanding of biological coevolution, and concludes with a summary and directions for future research.

A preliminary version of Chapter II was published in the 1996 COLT proceedings [81]. Much of Chapter III will appear in the Evolutionary Computation Journal [82]. Small portions of Chapter III and Section IV.C appeared in an ICGA conference paper [80].

Chapter II

Theory of Coevolutionary Methods

Competitive coevolution searches for solutions and test cases that defeat earlier test cases and solutions. The hope is that this approach can bootstrap its way to robust solutions and difficult test cases. In this chapter we use a simplified theoretical model to show how procedures like coevolution can efficiently bootstrap their way to solutions to adversarial problems.

Game theory and computational learning theory are the tools most relevant to our theoretical approach to adversarial problems, and simple games are the adversarial problems most amenable to complete theoretical analysis. Because of this, the definitions and results in this chapter are described in terms of games and game learning. This also allows us to consider a variety of relevant game learning methods (and gives the results appeal for a broader theoretical audience). The definition of "game" used, however, is equivalent to the definition of adversarial problems, so results on exact game learning apply directly to arbitrary adversarial problems.

Our framework for game learning relies on the existence of a *strategy learning algorithm* able to learn strategies which defeat a given, small set of opponents. This might correspond, for example, to a single successful "bootstrapping" step by coevolution: search produces a new individual that defeats a sample of prior opponents. The *competitive algorithm* then repeatedly uses a strategy learning algorithm to discover strong strategies for the game. We seek a competitive algorithm capable of learning perfect strategies for any game in polynomial time.

In Section II.A we give details of our model of game learning and describe its connection to familiar models of concept learning. Section II.B motivates the consideration of both worst-case and randomized strategy learning algorithms, and gives necessary parameters for measuring competitive algorithm performance in each case. We then examine several competitive algorithms motivated by methods used in practice. Section II.C presents two simple competitive algorithms, and shows examples on which they can fail to learn perfect strategies in polynomial time. A competitive algorithm that meets our performance goals with both worst-case and randomized strategy learning algorithms is given in Section II.D. Examples of its use are described, including an application to concept learning with a new kind of counterexample oracle. Section II.E explores the computational complexity of game learning. Section II.F describes a theoretical extension to approximate game learning using mixed strategies.

II.A Preliminaries

II.A.1 Definition of Games

A game is a function G which maps two inputs h and x (first and second player strategies) to an outcome $G(h, x)$. The first-player strategy comes from a set of possible first-player strategies, $h \in \mathcal{H}$. Similarly, the second-player strategy comes from a set of possible second-player strategies, $x \in \mathcal{X}$.[1] For most results presented here, the basic unit of learning is the entire strategy. A game consists of both players presenting a strategy and obtaining an outcome. No further structure to game play

[1]Many board games are largely symmetric. The main reason for making a distinction between first-player and second-player strategies for such games is the existence of a perfect strategy for one player but not the other.

(sequential rounds of play, etc.) is assumed. This allows a simple, unified, inclusive view of games.

The game outcome may take on many values, usable by a strategy learning algorithm, but our framework only considers one bit of outcome information: which player was the winner. The competition is assumed to be deterministic. For simplicity, no ties are allowed.

The notation $a \succ b$ indicates that strategy a defeats strategy b. This is also extended to sets of strategies: $a \succ B$ means that $\forall b \in B, a \succ b$. $A \succ B$ is defined as $\forall b \in B, \exists a \in A$ such that $a \succ b$.[2]

II.A.2 Framework for Game Learning

For most of our results, it is necessary to assume that there is a perfect strategy (either for the first player or the second player) that defeats all possible opposing strategies. For notational convenience, it is always assumed to be the first player that has a perfect strategy when it is necessary to make this distinction.[3] For the most part, we consider exact learning of this perfect strategy.

Consideration of exact learning simplifies matters, and seems a good way to start. Section II.F considers one possible extension to approximate learning using mixed strategies.

Structure of the Learning Model

There are two main components to the game learning systems we consider. We assume that first and second player *strategy learning algorithms* are available, that can find a strategy capable of defeating a given set A of opponents. Strategy learning

[2] $\succ$ is only meaningful in the context of a particular game G and should be subscripted $\succ_G$. Whenever we use this notation, the game is clear from context, so the subscript is dropped. This is also true for several other definitions.

[3] With this assumption, first-player strategies correspond to candidate solutions of an adversarial problem, and second-player strategies correspond to test cases.

algorithms may do this using heuristic search or some other appropriate method; we leave the details of this unspecified. Distinct strategy learning algorithms are required for the two players because different demands are placed on them, and because they may use different sets of strategies. The strategy learning algorithms are denoted L_1 for the first player, and L_2 for the second player. Since there is no perfect second-player strategy, no single strategy may be capable of defeating every member of A for L_2. So, we require that it return a set B of strategies, such that $B \succ A$ and $|B| \leq k'$ for some specified constant k'.

The *competitive algorithm* is the outer loop, and uses the strategy learning algorithms to produce new strategies. In this model, the *only* methods available to the competitive algorithm for producing new strategies are the strategy learning algorithms. For example, the competitive algorithm is not allowed to use domain-specific knowledge to modify strategies. Starting points for the competitive algorithm are obtained by calling the strategy learning algorithms on the empty set.

More formally, competitive algorithms may be described in the following terms. The competitive algorithm operates over multiple steps. Let F_i and S_i be the sets of first and second player strategies, respectively, that have been observed at the end of step i. F_0 and S_0 are set to the empty set. On step $i+1$ of the competitive algorithm:

1. F_{i+1} is initialized to F_i, and S_{i+1} is initialized to S_i.

2. Some subset $A_S \subseteq S_{i+1}$ is chosen.

3. L_1 is called on A_S, and the returned strategy is added to F_{i+1}.

4. Some subset $B_F \subseteq F_{i+1}$ is chosen.

5. L_2 is called on B_F, and the returned strategies are added to S_{i+1}.

Termination occurs when (5), in the above procedure, fails. This failure can only occur if B_F contains a perfect strategy, so termination occurs when a perfect strategy has been found.

Notes on the Learning Model

Several points should be made about this framework. First, though learning takes place through competition, the strategies chosen by one player inform learning done by the other. We are not trying to model an extreme competitive situation in which, for example, players intentionally present the weakest strategy possible to their opponents in order to be uninformative and to slow opponent learning.

As a concrete example of how this framework might be applied, consider Samuel's original work on learning evaluation functions for checkers from self-play[84]. Games were played between a fixed *Beta* player and a learning *Alpha* player. Alpha would learn from these games via Samuel's reinforcement learning algorithm; this corresponds to the strategy learning algorithm. When Alpha was finally able to defeat Beta, Beta was replaced by Alpha. The competitive algorithm uses the strategy learning algorithm to find a new (Alpha) strategy capable of defeating the current (Beta) strategy, then moves to this new strategy (makes Beta equal to Alpha).

In some empirical work on game learning, the competitive algorithm is not explicitly defined. For example, Tesauro's backgammon learning system[95] uses reinforcement learning without explicitly checking whether new strategies defeat old ones, as a competitive algorithm would. Given the continual improvement Tesauro observed in his strategies, it is likely that this condition was met implicitly.[4] So, our framework might be used to explain the performance of this system.

For the most part, we assume that a strategy learning algorithm is available. This is a fairly strong assumption; we make it for several reasons. Strategy learning is an optimization problem with a fixed, efficiently-computable cost function. This type of optimization problem has been well studied. Several researchers have had empirical success in strategy learning against fixed opponents [32, 85, 91]. The focus of the theory presented here is on the difficulties in game learning that arise *even though* a strategy learning algorithm exists. For example, it is not clear that Samuel's

[4]Similar results were obtained when this condition *was* checked explicitly by sampling the outcome of several games between new and old strategies [75].

bootstrapping procedure described above will converge rapidly on good strategies, even if new successful Beta strategies are continually found. As another example, we may ask whether competitive coevolution is certain to make long-term progress if search is usually locally successful in defeating current opposition. The definition of the model allows us to address such questions more easily, in isolation from the details of search for any particular application. Finally, this framework allows theoretical progress through the application of techniques from computational learning theory.

Time and Strategy Set Sizes

"Time" for a competitive algorithm refers to the total number of strategies considered by it. Polynomial time for a competitive algorithm will actually be polynomial clock-time as long as the strategy learning algorithm requires polynomial clock-time.

Bounds on learning time will be sought that are polynomial in $\lg(|\mathcal{H}|)$ and $\lg(|\mathcal{X}|)$. Such bounds will be most meaningful in the context of complex games if $\mathcal{H}$ and $\mathcal{X}$ are restricted to strategies that are compactly representable in some particular scheme. For example, strategies might be represented as neural net evaluation functions [85, 95, 97, 75]. The framework can also be used to consider the problem of learning to defeat particular classes of simple opponents, rather than arbitrary all-powerful opponents.

While $\mathcal{H}$ and $\mathcal{X}$ will not typically include all possible strategies for a game, they will usually be large enough that it is infeasible to consider all or most strategies in them. We need to generalize from an examination of a very small fraction of the strategy sets.

II.A.3 Correspondence with Concept Learning

Our model for game learning shares similarities with the typical model of concept learning. The first-player strategy space $\mathcal{H}$ corresponds to concept learning's hypothesis space; the assumption that it has a perfect strategy is similar to assuming that the hypothesis space contains the target concept. The first-player strategy learning algorithm corresponds to a hypothesis learner, and the second-player strategy learning algorithm can be viewed as a counterexample oracle. The competitive algorithm corresponds to various learning protocols that use a hypothesis learner multiple times (for example, alternating equivalence queries and calls to an algorithm providing consistent hypotheses).

Having a hypothesis h consistent with the target on an example x corresponds to $h \succ x$. An important difference between concept learning and game learning is that concept learning is primarily concerned with discovering how the target classifies new examples, whereas game learning requires the actual "target" strategy. In this sense, game learning is a more restrictive model. Positive results from game learning should carry over to concept learning (an example of this will be seen below) and negative results from concept learning should carry over to game learning.

II.B Competitive Algorithm Performance

Performance of competitive algorithms will always depend on $\lg(|\mathcal{H}|)$ and $\lg(|\mathcal{X}|)$. This type of dependence is familiar from concept learning, and shouldn't be prohibitive for reasonable representations of strategies. In this section, we establish necessary additional parameters for studying competitive algorithm performance.

II.B.1 Specification Number

For a particular game G with strategy spaces $\mathcal{H}$ and $\mathcal{X}$, define a *teaching set* T for G as a subset of $\mathcal{X}$ such that for any imperfect strategy $h \in \mathcal{H}$, $\exists x \in T$ such that $x \succ h$. Define the *specification number* k for G to be the size of the smallest teaching set. These definitions follow the corresponding ones for concept learning [10, 42].

Since we are considering exact learning, we seek bounds on learning time that are polynomial in k, as well as in $\lg(|\mathcal{H}|)$ and $\lg(|\mathcal{X}|)$. The following lemma shows that the dependence on k is necessary.

Lemma 1 *For games G with at most (a constant) c perfect strategies and with specification number k, there exists a randomized first-player strategy learning algorithm L_1 such that any competitive algorithm using it takes expected time $\Omega(k)$ to learn a perfect strategy.*

Proof: For arbitrary A, let $L_1(A)$ choose a strategy uniformly at random from the set $\{h \in \mathcal{H} | h \succ A\}$; this choice of L_1 will become important in Section II.B.3. Let T be the minimal teaching set for G; $|T| = k$. Each member of T must defeat at least one member of $\mathcal{H}$ that no other member of T defeats (otherwise there would be no reason to include it and T would not be minimal). So, for any A with $|A| < |T|$, there must be at least $k - |A|$ members of $\mathcal{H}$ not defeated by A. The probability of $L_1(A)$ returning a perfect strategy is at most $\frac{c}{k-|A|}$.[5] The probability that a competitive algorithm finds a perfect strategy by time t (for $t < k$) is at most $t\frac{c}{k-t}$, since at most t calls were made to $L_1(A)$, each with $|A| \leq t$. By time $t = \frac{k}{2c+1}$, the probability of finding a perfect strategy is at most $\frac{1}{2}$, giving a total expected time of at least $\frac{k}{4c+2}$, which is $\Omega(k)$. $\Box$

As an example, consider a class of games G^n_{hard}. G^n_{hard} has n first-player strategies, only one of which is perfect, and $n-1$ second-player strategies, each defeating only one imperfect first-player strategy. The specification number for this game is $k = n-1$. Any competitive algorithm using the first-player strategy learning algorithm described above requires $\Omega(k) = \Omega(n)$ time to learn a perfect strategy for this game; this is not polynomial in $\lg(|\mathcal{H}|)$ and $\lg(|\mathcal{X}|)$.

This indicates that time bounds for competitive algorithms need to depend on k, as well as $\lg(|\mathcal{X}|)$ and $\lg(|\mathcal{H}|)$. For game learning to be practical, k should be fairly small; this is the case for specific examples described below.

The second-player strategy learning algorithm requires $k' \geq k$ to ensure that it can always succeed. We assume that k' is polynomial in k, $\lg(|\mathcal{H}|)$, and $\lg(|\mathcal{X}|)$.

[5]This shows the necessity of bounding the number of perfect strategies by a constant. If c were allowed to grow with k, this probability might always be large.

II.B.2 Worst-case Strategy Learning Algorithms

In order to demonstrate the power of competitive techniques, we would like a competitive algorithm that can solve games in time polynomial in $\lg(|\mathcal{X}|)$, $\lg(|\mathcal{H}|)$, and k with *worst-case* choices of L_1 and L_2. Unfortunately, the following lemma shows that this is not possible. First, a definition is needed:

Define a *transitive chain* of length ℓ in a game to be a sequence of pairs (h_i, X_i) $(i = 1, 2, \ldots \ell)$, with $h_i \in \mathcal{H}$ and $X_i \subseteq \mathcal{X}$, $|X_i| \leq k'$, such that:

1. $\forall i > j, h_i \succ X_j$

2. $\forall i \geq j, X_i \succ \{h_j\}$

Lemma 2 *For games G with a transitive chain of length ℓ, there exist L_1 and L_2 such that any competitive algorithm using these strategy learning algorithms requires $\Omega(\ell)$ time to learn a perfect strategy.*

Proof: Assume G has a transitive chain $(h_1, X_1) \ldots (h_\ell, X_\ell)$.

If L_1 and L_2 are passed the empty set, let them return h_1 and X_1, respectively.

If A contains only members from the transitive chain, let $L_1(A)$ return h_i, where $A = \{X_{i_1}, X_{i_2}, \ldots X_{i_{|A|}}\}$ and $i = (\max_j i_j) + 1$. Otherwise, let L_1 return the perfect strategy.

Similarly, if B contains only members from the transitive chain, let $L_2(B)$ return X_i, where $B = \{h_{i_1}, h_{i_2}, \ldots h_{i_{|B|}}\}$ and $i = (\max_j i_j) + 1$. Otherwise, let L_2 return a teaching set.

The first call to L_1 or L_2 must pass the empty set (since the competitive algorithm has no strategies available yet), so that the first strategies available to the competitive algorithm are h_1 or X_1. At least the next ℓ calls return only strategies from the transitive chain, with each successive call increasing the distance into the transitive chain by at most 1. Since none of the strategies in the transitive chain may be perfect except for the last one, the competitive algorithm cannot learn the perfect strategy in less than ℓ steps. $\square$

As an example, consider the following class of games: $\mathcal{H}$ consists of the numbers $1 \ldots n$ and $\mathcal{X}$ consists of the numbers $0 \ldots n-1$. $h \succ x$ iff $h > x$. There is a perfect first player strategy and $k = 1$. But, there is a transitive chain of length n: $(0, \{0\}), (1, \{1\}), \ldots (n-1, \{n-1\})$. So, with worst-case strategy learning algorithms, there is no competitive algorithm that solves these games in time polynomial in $\lg(|\mathcal{H}|)$, $\lg(|\mathcal{X}|)$, and k.

Due to Lemma 2, upper bounds for competitive algorithms using worst-case strategy learning algorithms must depend on transitive chain length ℓ, as well as $\lg(|\mathcal{H}|)$, $\lg(|\mathcal{X}|)$, and k. Unfortunately, sensitivity to transitive chains is not just a property of a few perverse games. The example below shows that at least some classes of strategies for a real board game (Go) can also contain long transitive chains.

Example 1 In the game of Go, players "black" and "white" alternate placing stones of their respective color on intersections of an n by n grid in an attempt to capture territory. Once placed, stones are never moved. Each turn, almost every empty intersection on the grid is a legal move. See [56] for details about the game.

The transitive chain is constructed from an exponentially large number of strategies, each of which has a unique numeric label. Strategies spend the initial moves of the game communicating this label, then cooperate so that the strategy with the numerically larger label wins the game.

Consider the following set of strategies for the game of Go on an n by n board. Define the 2 vertical lines on the left edge of the board as the "white side" and the 2 vertical lines on the right edge of the board as the "black side"; each of these areas is an n by 2 rectangle of intersections. Assume each player's strategy space includes a set of 2^n strategies, each with a unique n bit label. During the first n moves of a game, these strategies "communicate labels," by successively playing n moves along the appropriate side for the player's color. Each of these moves is on the first line (on the edge of the board) to communicate a 0, and the second line to communicate a 1.

When two of these strategies play one another, after each has communicated all n bits, the strategies work together to produce an outcome such that the strategy with the numerically greater label wins; this is very easy to do when the two strategies cooperate to produce this outcome. This yields a transitive chain in which a strategy with label Z (when treated as an n bit binary number) defeats all strategies with labels less than Z and loses to all strategies with labels greater than Z. This yields a transitive chain with length 2^n. The operations required to form this transitive chain are not very complex; many natural, powerful classes of strategies for Go will contain such a long transitive chain. □

This example depends on unusual strategies that intuitively seem cooperative rather than reasonably competitive. It should be possible to modify the example to use strategies that are nominally competitive and meet some simple definition of "uncooperative" strategies. Given our simple framework, however, Example 1 is adequate to show long transitive chains may exist within powerful classes of strategies for Go.

This example could be easily extended to N by N checkers [34]. This suggests that we should go beyond worst-case strategy learning algorithms, to eliminate the need for dependence on ℓ.

II.B.3 Randomized Strategy Learning Algorithms

In practice, it seems unlikely that simple, natural strategy learning algorithms will be "adversarial", producing uninformative near-worst-case strategies. To obtain positive results for competitive algorithms without using ℓ as a parameter, we need to restrict, in some meaningful way, the strategy learning algorithms that we consider. Note that it would be uninformative to assume *best-case* strategy learning algorithms. Since we allow the strategy learning algorithm unlimited access to the game, any game with a solution has a first-player strategy learning algorithm that immediately produces a perfect strategy.

Another typical approach for going beyond worst-case algorithms is to use randomized learning algorithms [60]. We consider randomized strategy learning algorithms that, when passed a set A of opponents, choose from a distribution over the set of strategies (or strategy sets) that defeat A. Randomization is most helpful when this distribution is required to be uniform, and we will typically assume "randomized strategy learning algorithm" to mean this below; Theorem 4 relaxes the uniform-distribution condition somewhat, however. The importance of randomization is that it does not allow strategy learning algorithms to always choose from a few bad strategies that make the competitive algorithm fail.

Recall that Lemma 1 used a randomized strategy learning algorithm. Hence, competitive algorithm time bounds will depend on $\lg(|\mathcal{H}|)$, $\lg(|\mathcal{X}|)$, and k when randomized strategy learning algorithms are used.

II.C Two Simple Competitive Algorithms

Two simple competitive algorithms are described in this section that make few demands on the strategy learning algorithms. We then motivate a more complex competitive algorithm by showing natural examples where the simpler competitive algorithms may fail to solve games in polynomial time, even with randomized strategy learning algorithms.

II.C.1 Each Defeats the Last

A simple method of learning is to have the competitive algorithm obtain an initial first-player strategy s, then find a second-player strategy t with $t \succ s$. Then, find $s' \succ t$, $t' \succ s'$, and so on. In the framework given in Section II.A.2, A_S is always chosen to be the single strategy most recently added to S_i, and similarly for B_F. This is essentially the competitive algorithm used in Samuel's checkers learning system, and is very similar to that used in a recent backgammon learning system [75].

The main problem with this competitive algorithm is that intransitivity may exist, and a particular learner may simply keep choosing strategies in a cycle. Intransitivity has been observed when using this competitive algorithm for backgammon

[75]. Even a randomized learner may get stuck in such a cycle for a long time. The following example demonstrates this.

Example 2 (Small Game Trees) This example considers games represented by game trees. The time required by the strategy learning algorithm is polynomial in the number of nodes in the tree, so this example could be reasonably extended to familiar simple games like tic-tac-toe.

Let T_d be the complete binary tree of depth d; T_d has $n = 2^{d+1} - 1$ nodes and $l = 2^d$ leaves. Let G_d be the class of games with game tree T_d, with leaves labelled with binary outcomes in all possible 2^l ways. An outcome of 1 indicates a first-player win, while an outcome of 0 indicates a second-player win. Let $\mathcal{H}$ consist of all possible first-player strategies (all possible sets of choices at the first-player nodes). Let $\mathcal{X}$ consist of all possible second-player strategies. Note that $\lg(|\mathcal{H}|)$ and $\lg(|\mathcal{X}|)$ are $O(n)$. Each game in G_d must either be a first player win or a second player win with perfect play, so let $G'_d \subset G_d$ be the set of games in G_d that are a first player win. Since $\mathcal{H}$ contains all possible strategies for G'_d, it must contain a winning strategy for each element of G'_d.

These games have $k \leq n$. For a particular $g \in G'_d$, for each imperfect $h \in \mathcal{H}$, there must be a leaf of g with label 0 such that h may reach this leaf, given appropriate second-player play. For each leaf, there is a strategy $x \in \mathcal{X}$ that will always reach that leaf, if it is reachable given the opposing first player. Let T contain one strategy from $\mathcal{X}$ for each leaf in g labelled 0. Now, for any imperfect $h \in \mathcal{H}$, there exists $x \in T$ with $x \succ h$. So, T is a teaching set, and is of size $|T| \leq n$.

The maximum length of a transitive chain in these games is $\ell \leq n$. Each set of second-player strategies X_i in a chain must defeat h_i via a leaf l_0 unreachable by any previous second-player strategy in the chain; such a previous strategy that could reach l_0 could defeat h_i, which contradicts the definition of a transitive chain. Since each step in the transitive chain must contain a second-player strategy that can reach a leaf no previous second-player strategy could reach, the length of the longest transitive chain is bounded by the number of leaves, which is bounded by n.

A randomized strategy learning algorithm may be constructed for these games. For a particular opponent O, all that needs to be specified to defeat it is the moves along a single path through the game tree, such that O is consistent with all the choices along the path and loses the game. If O is a first player, each such path requires specification of $\lfloor \frac{d}{2} \rfloor$ moves. If O is a second player, each such path requires specification of $\lceil \frac{d}{2} \rceil$ moves. A list of all such winning paths can be constructed in time polynomial in l, since there is only one possible path for each leaf. To make the uniform choice, a random choice is made from this list and the irrelevant bits are then set to 1 with probability $\frac{1}{2}$ and to 0 with probability $\frac{1}{2}$.

The competitive algorithm described in this section will typically fail to solve such games in polynomial time using this learning algorithm. This is easy to see: the learner always sets most of the strategy bits randomly. For example, consider games where the first player must always choose the right branch to win. Considering all possible second-player responses, at least $2^{\lceil \frac{d}{2} \rceil} - 1$ bits must be correctly set to have a perfect first-player strategy, only $\lceil \frac{d}{2} \rceil$ of which will actually be specified (not set randomly) in any strategy produced by the learning algorithm. Correctly guessing the remaining bits will require time doubly exponential in d, which is exponential in n. $\square$

For this example, the competitive algorithm fails to learn a perfect strategy in time polynomial in $\lg(|\mathcal{H}|)$, $\lg(|\mathcal{X}|)$, k, and ℓ, even using randomized strategy learning algorithms.

Adding finite memory to this competitive algorithm, by always choosing a strategy that defeats the last h opponents, is of limited usefulness; it can still fail on cycles with period greater than h. The next competitive algorithm uses a memory of *all* previous second-player strategies.

II.C.2 Single Counterexamples

In exact concept learning, one type of model considers equivalence queries. Given a hypothesis, an equivalence query returns "yes" if the hypothesis is the target, and provides a counterexample if it is not. The concept learner queries with a hypothesis consistent with all examples seen; if its hypothesis is wrong, it chooses a new one consistent with all examples including the newly received counterexample. Translated into a game learning context, a counterexample is a second-player strategy chosen to defeat a proposed first-player strategy. The first-player strategy learning algorithm is now required to choose strategies that defeat *all* second-player strategies that have been seen during learning. In terms of the definition from Section II.A.2, B_F is chosen to be the single strategy most recently added to F_{i+1}, but A_S is chosen to include all strategies from S_{i+1}. This is a natural extension to the competitive algorithm in the previous subsection.

This competitive algorithm is sufficient to learn perfect strategies for the games in Example 2, in time polynomial in n using *any* strategy learning algorithms. For games in G_d, each counterexample must be capable of reaching a particular leaf l_0 that is a first-player loss. Subsequently chosen first-player strategies cannot make the first-player moves leading to l_0, or else they would be defeated by this counterexample. These first-player strategies must be defeated by leaves other than l_0. So, each new counterexample wins via a leaf by which no previous counterexample could win. This limits the total number of steps of the competitive algorithm to l.

Counterexamples are not sufficient to learn all games, though. Angluin has shown that for some classes in concept learning, an adversary may choose uninformative counterexamples so that the target cannot be exactly learned in polynomial time [9]. Below, we show the existence of a game for which counterexamples do not ensure polynomial learnability, with either worst-case strategy learning algorithms or randomized strategy learning algorithms.

Example 3 (Generalized Guessing Games) Let $\mathcal{H}$ consist of all natural numbers in the range $0 \ldots n-1$. Each member of $\mathcal{X}$ is a union of at most I intervals in this range. A game g_y is defined by a particular number y in this range, and the particular choice of unions of intervals in $\mathcal{X}$. Every $x \in \mathcal{X}$ not containing y loses all games in g_y. Other than this, $h \in \mathcal{H}$ defeats x iff x contains h. To ensure that the specification number is 1, the single interval containing only y is included in $\mathcal{X}$.

As a worst-case example, assume that $\mathcal{X}$ consists of all unions of at most two intervals that exclude a single number, and the interval containing only y. The second-player strategy learning algorithm chooses the interval excluding h as a counterexample to the first-player strategy h (assuming $h \neq y$). The first-player strategy learning algorithm makes a choice uniformly at random from the intersection of all passed second-player strategies. Since each counterexample eliminates only one first-player strategy, this requires $\Omega(n)$ expected time to learn the perfect strategy. Since ℓ is $O(1)$ (the only second-player strategy defeating more than one first-player strategy is the interval containing only y), this time bound is not polynomial in $\lg(|\mathcal{H}|)$, $\lg(|\mathcal{X}|)$, k, and ℓ.

For an example where randomized strategy learning algorithms fail, let the first-player strategy learning algorithm be defined as above and let the second-player strategy learning algorithm choose uniformly from all members of $\mathcal{X}$ containing the passed first-player strategy. Let $\mathcal{X}$ contain all unions of at most two intervals that exclude only elements $i \ldots (i + \lfloor\sqrt{n}\rfloor) \bmod n$, for each i. $\mathcal{X}$ also contains the interval consisting only of y, but the probability of choosing this informative counterexample is only $\frac{1}{\lfloor\sqrt{n}\rfloor+1}$ each step of the competitive algorithm. After t steps, the probability of the first-player strategy learning algorithm choosing y is at most $\frac{1}{n-t(\lfloor\sqrt{n}\rfloor+1)}$. So, the expected time to find y is not polynomial in $\lg n$. $\square$

II.D The Covering Competitive Algorithm

In this section, we present the *covering competitive algorithm*, which uses the idea of covering all previous opposition to provide polynomial-time learnability. After proving an upper bound on the number of steps required by this competitive algorithm, we give a lower bound to show the upper bound is nearly tight. We then give a more complex competitive algorithm with improved running time. We conclude the section with several examples that use the covering competitive algorithm.

Section IV.D gives an additional example of its use.

II.D.1 Covering all First and Second Player Opponents

Counterexamples to single first-player strategies fail to provide polynomial learnability for all games. Note that the first-player strategy learning algorithm, at every step, finds a strategy that defeats all second-player strategies already seen. It seems natural to produce new second-player strategies in a similar way. Unless the specification number is 1, it may not always be possible to provide a single second-player strategy that defeats all first-player strategies already seen, but as long as the perfect first player has not yet been chosen, it is always possible to choose a set of second players of size k that covers all first players already seen.

This competitive algorithm will be called the *covering competitive algorithm*. Each step, the first-player strategy learning algorithm is called on all previous second-player strategies, then the second-player strategy learning algorithm is called on all previous first-player strategies (including the new one from the current step). In terms of the definition from Section II.A.2, B_F is always chosen to contain all strategies from F_{i+1}, and A_S is always chosen to contain all strategies from S_{i+1}.

The idea of measuring coverage of prior opponents has been discussed in the context of empirical game learning systems [23].

II.D.2 Using Worst-case Strategy Learning Algorithms

This competitive algorithm performs as well as possible with worst-case strategy learning algorithms.

Theorem 3 *Assume a game G has maximum transitive chain length ℓ. The covering competitive algorithm learns a perfect strategy for G in at most $\ell(k'+1)$ strategies.*

This is true simply because this competitive algorithms explicitly constructs transitive chains. This gives at most ℓ steps, each of which obtains at most one first-player and k' second-player strategies. Since k' is polynomial in k, $\lg(|\mathcal{H}|)$, and $\lg(|\mathcal{X}|)$, this time bound is polynomial in the relevant parameters.

II.D.3 Using a Randomized Strategy Learning Algorithm

For this positive result, it is desirable to relax the uniform-choice condition for randomized strategy learning algorithms. Define the (p, q)-*randomization criterion* for sampling from an arbitrary set Y, as follows. The choice must be made from a distribution D over Y, with the property that there exist constants p and q ($0 < p, q \leq 1$) such that there are at least $p|Y|$ elements of Y to which D assigns a probability of at least $\frac{q}{|Y|}$.

At each step of this competitive algorithm, denote by X the set of remaining feasible sets of second-player strategies (the sets of at most k' second-player strategies that defeat all previously chosen first-player strategies). Similarly, call the remaining set of feasible first-player strategies H.

Theorem 4 *Assume that the first-player strategy learning algorithm, at each step, makes a choice from H that satisfies the (p, q)-randomization criterion for some fixed p and q. Assume that the second-player strategy learning algorithm, on each step, chooses at most k' strategies with k' polynomial in k, $\lg(|\mathcal{H}|)$, and $\lg(|\mathcal{X}|)$. Then, the perfect strategy is found with the expected number of strategies chosen bounded by a polynomial in $\lg(|\mathcal{H}|)$, $\lg(|\mathcal{X}|)$, k, $\frac{1}{p}$, and $\frac{1}{q}$.*

Proof: The idea will be to consider sequences of m steps, during which a "sample" of m first-player strategies are chosen from the remainder of $\mathcal{H}$. By a PAC-like argument, the second-player strategies chosen at the end of the m steps to defeat this sample must eliminate a large fraction of the remaining strategies in $\mathcal{H}$. The only real complication in the proof comes from the fact that successive members of the sample may be chosen from somewhat different distributions, due to the (p, q)-randomization criterion and the fact that H may shrink over the course of the m steps.

Consider a sequence of $m = \frac{2+k' \lg(|\mathcal{X}|)}{-\lg(1-\frac{pq}{4})}$ steps of the competitive algorithm. Let H_0 be H at the start of this sequence, and let H_t be H after t steps into the sequence. Similarly, let X_0 be X at the start of this sequence, and let X_t be X t steps into the sequence. The goal is to show that a fraction of at least $\frac{p}{4}$ of H_0 will be eliminated after these m steps are completed, with high probability.

The last set of second players, x, chosen in the sequence must defeat the m first players already chosen. Consider two cases. In the first case, assume that $|H_{m-1}| \leq (1-\frac{p}{4})|H_0|$ (note that H_{m-1} is the set of remaining first-player strategies just before x is chosen). In this case, the condition on this sequence of steps is already satisfied.

In the second case, assume that this is not true. Then, at least $1-\frac{p}{4}$ of the original set of first players, H_0, remain (and were also present each prior step of the sequence). Consider a particular $x_{bad} \in X_0$ that is "bad": defeats less than $\frac{p}{4}$ of H_0. Each step t of the sequence of m, at most $\frac{p}{2}$ of the strategies in H_0 fail to defeat x_{bad}, or have already been eliminated. On each such step, there was a probability of at least $\frac{(2p-p^2)q}{4}$ that a first-player strategy other than one of these was chosen. This is because at least $p|H_t| - \frac{p}{2}|H_0| \geq p(1-\frac{p}{4})|H_0| - \frac{p}{2}|H_0| = \frac{2p-p^2}{4}|H_0|$ first players must: have not yet been eliminated, defeat x_{bad}, and have a probability of at least $\frac{q}{|H_t|} \geq \frac{q}{|H_0|}$ of being chosen. After m first players are chosen[6], the probability that x_{bad} remains in X_t when x is chosen is at most $(1-\frac{(2p-p^2)q}{4})^m$. At worst, every member of X_0 is bad, so $|X_0|(1-\frac{(2p-p^2)q}{4})^m$ is an upper bound on the probability that any bad second player remains at the last step.

Assuming that the probability of the first case is 0 (worst case), the maximum probability of failure to eliminate at least $\frac{p}{4}$ of H_0 this sequence of m steps is $|X_0|(1-\frac{(2p-p^2)q}{4})^m$. This is at most $|X_0|(1-\frac{pq}{4})^m$. We want

$$|X_0|(1-\frac{pq}{4})^m < \delta$$

$$\lg(|X_0|) + m\lg(1-\frac{pq}{4}) < \lg(\delta)$$

[6]These choices are not fully independent. But the events that we consider here are independent.

$$m > \frac{\lg(\frac{|X_0|}{\delta})}{-\lg(1-\frac{pq}{4})}$$

Since $|X_0| \leq |\mathcal{X}|^{k'}$, this will be satisfied with $\delta = \frac{1}{2}$ if

$$m > \frac{1 + k' \lg(|\mathcal{X}|)}{-\lg(1-\frac{pq}{4})}.$$

The choice of m ensures this. Each m steps, with probability at least $\frac{1}{2}$, H will lose at least a fraction $\frac{p}{4}$ of its size; this gives an expected decrease of at least a fraction $\frac{p}{8}$. This means that the algorithm terminates in (expected) $O(\frac{\lg(|\mathcal{H}|)}{-\lg(1-\frac{p}{8})})$ sequences of m steps. Since each step adds at most k' second players (and 1 first player) to the sets, an expected total of

$$O((\frac{\lg(|\mathcal{H}|)}{-\lg(1-\frac{p}{8})})(\frac{2 + k' \lg(|\mathcal{X}|)}{-\lg(1-\frac{pq}{4})})(k'+1))$$

strategies will be added to the sets before termination. This is

$$O(\frac{k'^2}{p^2 q} \lg(|\mathcal{H}|) \lg(|\mathcal{X}|)).$$

Since k' was assumed to be polynomial in k, $\lg(|\mathcal{H}|)$, and $\lg(|\mathcal{X}|)$, the covering competitive algorithm chooses an expected number of strategies polynomial in $\lg(|\mathcal{H}|)$, $\lg(|\mathcal{X}|)$, k, $\frac{1}{p}$, and $\frac{1}{q}$. □

The second-player strategy learning algorithm has a fairly difficult task: it must find sets of strategies that cover all first-player opponents. Since this theorem allows arbitrary second-player strategy learning algorithms, we can reduce this task somewhat. Assume that an algorithm L_2' is available that takes as input a set A of first-player strategies, and returns the second-player strategy covering as many of these as possible. Since a set of size k exists that covers all of A, at least one member of this set must cover $\lceil \frac{1}{k} \rceil$ of A. Using L_2' repeatedly, a set-covering approximation can be used to create a second-player strategy learning algorithm that returns sets of size at most $k' = k \log |\mathcal{H}|$ [22].

If a randomized strategy learning algorithm is available that can find k' second-player strategies all at once, satisfying the (p, q)-randomization criterion in the space of remaining feasible sets of k' second-player strategies, then the roles in Theorem 4 may be reversed, giving the following corollary.

Corollary 5 *Assume that the second-player learning algorithm, at each step, makes a choice from* X *that satisfies the* (p,q)*-randomization criterion. Then, the perfect strategy will be found by the covering competitive algorithm with the expected number of strategies chosen bounded by a polynomial in* $\lg(|\mathcal{H}|)$, $\lg(|\mathcal{X}|)$, k, $\frac{1}{p}$, *and* $\frac{1}{q}$.

II.D.4 A Matching Lower Bound

We show here that the dependences on k', $\lg(|\mathcal{H}|)$, and $\lg(|\mathcal{X}|)$ described above are nearly tight, for the covering competitive algorithm. We construct strategy spaces and strategy learning algorithms such that the time required by the covering competitive algorithm is close to (i.e. within small log factors of) the upper bound given in the proof of Theorem 4.

$\mathcal{H}$ and $\mathcal{X}$ are partitioned in d levels; the covering competitive algorithm progresses through these levels one by one. The construction depends on chosen values for k (the specification number), and parameters b and d. The strategy space sizes depend on these parameters as described below; an adequate range of variation in the strategy space sizes can be achieved through appropriate choices of these parameters.

Beyond the d levels, $\mathcal{H}$ has a separate perfect strategy (consider this to be in level $d+1$). In each other level it contains k sets of b blocks each, where each block consists of identically-behaving strategies.[7] We require that $b \geq 2$. The block size is 1 strategy in level d, (kb) strategies in level $d-1$, $(kb)^2$ in level $d-2$, and so on up to $(kb)^{(d-1)}$ strategies per block in level 1. This is $(kb)^{(d-i)}$ strategies per block in level i, for a total of $(kb)^{d-i+1}$ strategies in level i.

$\mathcal{X}$ has d levels as well, but each is the same size. Each level consists of k sets, each containing 2^b strategies. These strategies defeat/lose to the b distinct strategies in the corresponding set of the same level of $\mathcal{H}$ in all possible 2^b ways; sets of k of these can defeat any subset of the blocks of strategies from the same level of $\mathcal{H}$. Strategies in level i defeat all strategies in levels $j < i$ of $\mathcal{H}$, and lose to all

[7] We need these duplicate strategies to get the right distribution of outcomes when choosing randomly among strategies from $\mathcal{H}$.

strategies in levels $j > i$ of $\mathcal{H}$.

The strategy learning algorithm for $\mathcal{X}$, when passed a set of strategies from $\mathcal{H}$ with maximum level i, returns k strategies from level i that defeat only the blocks of level i to which the passed strategies belong (along with defeating all strategies from previous levels). Note that $k' = k$.

The strategy learning algorithm for $\mathcal{H}$, when passed a set of opponents with maximum level i, chooses uniformly at random from the strategies in levels i and $i+1$ that defeat all opponents passed. Note that, for all levels j, level $j+1$ is at most half of the size of level j. Therefore, the total number of strategies in levels higher than $i+1$ is less than the number of strategies in level $i+1$. Since the randomized choice made by this strategy learning algorithm comes from a set that at least includes all strategies in level $i+1$, it satisfies the (p, q)-randomization criterion with $p \geq \frac{1}{2}$ and $q = 1$.[8]

After s steps in level i, s blocks from level i of $\mathcal{H}$ have been eliminated, so that $kb - s$ blocks remain. If a new strategy from level i of $\mathcal{H}$ is chosen, the block to which it belongs is eliminated by the new strategies from $\mathcal{X}$. Level $i+1$ of $\mathcal{H}$ has the same size as a single block of level i, so the probability of choosing a strategy from it is $\frac{1}{kb-s+1}$. This gives an $\Omega(kb)$ expected time spent in level i.

Learning ends when the perfect strategy is chosen. Since learning passes through d levels before this point, the total expected number of choices is $\Omega(kbd)$. With $k+1$ strategies per choice this is $\Omega(k^2bd)$ strategies. $|\mathcal{H}|$ is $O((kb)^d)$ and $|\mathcal{X}|$ is $O(dk2^b)$. The bound from Theorem 4 thus gives $O(d(\lg(k)+\lg(b))(b+\lg(k)+\lg(d))k^2)$, which only differs from the lower bound by small log factors.

[8]For large values of b, p becomes very close to 1.

II.D.5 A Competitive Algorithm with an Improved Upper Bound

The covering competitive algorithm can be modified so that the extra factor of k' in the upper bound of Theorem 4 can be removed. The proof of Theorem 4 only relies on the effect of the k' second-player strategies chosen on the *last* step of a sequence of m steps (where m is defined as in the proof). The second-player strategies chosen earlier in the sequence of m steps only serve to complicate the proof by potentially changing the distribution from which the first-player strategies are chosen during this time.

Define an *improved competitive algorithm* that begins by computing m (this is possible given knowledge about the strategy set sizes and the strategy learning algorithms). In each step (which corresponds to m steps of the covering competitive algorithm), it calls the first-player strategy learning algorithm m times on the set of all second-player strategies that have been seen. It then calls the second-player strategy learning algorithm once, to obtain k' second-player strategies that cover all first-player strategies that have been seen.

The proof from Theorem 4 showing elimination of a substantial fraction of H each m steps, applies here to a *single* step. So, the improved competitive algorithm terminates in an expected $O(\frac{\lg(|\mathcal{H}|)}{-\lg(1-\frac{p}{8})})$ single steps. Each single step of the improved competitive algorithm chooses m first-player strategies and k' second player strategies. Since $k' < m$, the extra factor of k' disappears from the bound.

Theorem 6 *The total expected number of strategies chosen by the improved competitive algorithm is:*

$$O(\frac{k'}{p^2 q}\lg(|\mathcal{H}|)\lg(|\mathcal{X}|)).$$

This result is of theoretical interest, but the improved competitive algorithm is more complex and requires detailed knowledge of the first-player strategy learning algorithm and second-player strategy set size. Theorem 4 is more intuitively appealing because it provides a positive result for a simpler, more natural competitive algorithm.

II.D.6 Examples

Generalized Guessing Games

By Theorem 3, use of the covering competitive algorithm gives us our desired worst-case performance on the generalized guessing games of Example 3. Also, the randomized strategy learning algorithms described for it satisfy the (p, q)-randomization criterion with $p = q = 1$, allowing the covering competitive algorithm to solve the game in time polynomial in n.

Concept Learning

A game-learning approach can also be applied to concept learning, using a new kind of counterexample oracle.

Example 4 (Learning Teachable Concepts) Assume a concept class $\mathcal{H}$ (serving as hypothesis and target class) and a space of examples, $\mathcal{X}$. A "game" g_t is associated with each $t \in \mathcal{H}$, with $g_t(h, x)$ being 1 (victory for h) if t is consistent with h on x, and 0 otherwise. An oracle is assumed available to provide *counterexample sets*. If the specification number (of a particular g_t) is k, the oracle, when given a set of (imperfect) hypotheses H, returns a set X of at most k' examples such that for each $h \in H$, $g_t(h, x) = 0$ for some $x \in X$. Assume that k' is polynomial in k, $\lg(|\mathcal{H}|)$, and $\lg(|\mathcal{X}|)$. Each step, the learner finds a hypothesis consistent with the current set of counterexamples, and adds it to the current set of hypotheses. Then, the oracle provides a set of examples that cover the current set of hypotheses; the members of this set are added to the current set of counterexamples. The target will be learned exactly in an expected number of examples polynomial in $\lg(|\mathcal{H}|)$, $\lg(|\mathcal{X}|)$, k, $\frac{1}{p}$, and $\frac{1}{q}$ if at least one of the following two conditions holds:

1. The oracle chooses a cover of k' examples from the space of possible covers consisting of k' examples, such that this choice satisfies the (p, q)-randomization criterion.

2. The learner chooses consistent hypotheses from $\mathcal{H}$, such that this choice satisfies the (p, q)-randomization criterion.

For the second possibility, the counterexample oracle can be reduced to an oracle that returns a counterexample to as many of the current hypotheses as possible; the set covering approximation can then be used. This result shows that, with such a powerful counterexample oracle, any teachable class of hypotheses can be learned. It has previously been shown that any finite concept class may be efficiently learned with simple counterexamples and powerful computational abilities, by querying with hypotheses outside of $\mathcal{H}$ [19]. Our result is distinct from this, because it relies only on hypotheses from $\mathcal{H}$.

Note that fixing k' is crucial. If the counterexample oracle were allowed to provide as many counterexamples as passed hypotheses, it could simply provide a separate counterexample to each hypothesis. The learning model would have no more power than equivalence queries. In this sense, a small set of counterexamples is more informative than a large set of counterexamples. □

II.D.7 Other Strategy Learning Algorithms

There are some games which, despite the existence of long transitive chains, have strategy learning algorithms that allow the covering competitive algorithm to learn perfect strategies in polynomial time without meeting the (p, q)-randomization criterion. The following example gives a strategy learning algorithm that uses a particular form of evaluation function to learn strategies for deterministic, terminating, perfect-information, 2-player games with alternating turns. The representation used permits fairly powerful strategies; this example begins to approach a practical learning algorithm that might be applied to a complex game.

Example 5 We define a first-player strategy space and strategy learning algorithm. We then show that, using these with *any* second-player strategy space and strategy learning algorithm, the covering competitive algorithm succeeds in a polynomial number of steps.

Consider a first-player strategy space consisting of evaluation functions built out of predicates over positions. Assume there are P predicates, with strategies given as a binary vector of length P, indicating which predicates should be included in the strategy. Evaluations are binary, with each position evaluated as 0 (loss for first player) or 1 (win for first player). A position is evaluated as 1 iff at least one of the predicates in the strategy is `true` for the position; the evaluation function is the disjunction of the predicates. It is assumed that the correct evaluation function, giving the game-theoretic value of each position, can be represented by some disjunction of these predicates.

The predicates could take on any form. As a simple example, positions might be labelled with binary vectors, and the set of predicates might be the set of conjunctions of v literals over this vector. Strategies are then represented as v-DNF expressions. As another example, a fixed database of positions could be used as the basis of a nearest-neighbor representation; the fixed database effectively tiles the space into a number of cells equal to the number of positions in the database. Each of these cells corresponds to a single predicate giving the evaluation for positions in that cell.

The strategy learning algorithm maintains a current strategy; this state is kept across calls from the competitive algorithm. The initial strategy contains all predicates. After that, predicates may be eliminated but are never added back into the current strategy. The current strategy is maintained such that a position that is evaluated as 0 is always correctly labelled.

When called from the competitive algorithm, the strategy learning algorithm repeatedly loops through the given second-player opponents and plays each one. To make a move during a game using an evaluation function, the position resulting from each possible move is evaluated with the evaluation function. The first such move (in some fixed ordering) evaluated as 1 is chosen.

If no moves are evaluated as 1, the position is lost. So, the prior position (to which the most recent first-player move was made) was incorrectly labelled with a 1. This is also true if the game is lost after the prior position was reached, before another first-player move could be made. All predicates which include this mislabelled position are removed. After this point in the game, arbitrary choices may be made and no further learning is done.

If, after playing each opponent, all games are won, then strategy learning is complete. If a game was lost to some opponent, then there must have been a mislabelled position discovered, and predicates must have been removed from the current strategy. Since there are only P predicates available, at most P passes through all opponents will be required for the strategy learning algorithm to successfully terminate.

Figure II.1 specifies the disjunctive evaluation function learner in detail. We prove the following lemma about this strategy learning algorithm by induction:

Lemma 7 *Any predicates removed from E by the strategy learning algorithm cannot be part of the completely correct evaluation function (and are therefore correctly removed).*

Proof: The first predicate removed must include a position that is known to be lost for one of the following reasons:

1. The position is a final state of the game that is a loss for the first player.
2. The second player can make a move from the position, to a final state that is a loss for the first player.
3. The second player made a move to another position, from which all moves are not covered by any predicate in the original set. Since the perfect evaluation function cannot include any of these positions, they are all losses for the first player.

```
/* Input:  a set A of second-player strategies to defeat.  */
If this is the first call to the learner:
    Initialize E to contain all P predicates.
Else:  start with E returned from previous call.

Repeat until all members of A are defeated:
    /* Play each opponent, removing all incorrect predicates found.  */

    For each opponent O ∈ A:
        z ← initial position of the game.  /* z holds game state.  */
        Repeat until game ends:
            If no successor s of z is covered by E:
                /* temp records the previous position covered by E.  */
                Remove from E all predicates that cover temp.
                End game.  /* A loss has been reached.  */

            /* Otherwise, continue game, making first-player move:  */
            z ← first successor of z covered by E.
            If z is a first-player loss:
                Remove from E all predicates that cover z.
                End game.  /* A loss has been reached.  */

            /* Otherwise, continue game, making second-player move:  */
            temp ← z /* Record last position covered by E.  */
            z ← O's move from z.
            If z is a first-player loss:
                /* Last position covered by E is a loss:  */
                Remove from E all predicates that cover temp.
                End game.  /* A loss has been reached.  */
Return E.
```

Figure II.1: Disjunctive Evaluation Function Learner

In each of these cases, the removed predicate covers a position that is a forced loss for the first player. So, the first predicate removed is correctly removed.

For the n-th predicate removed, assume that all predicates removed previously were correctly removed, so that any position reached that is not covered by any current predicate is correctly evaluated as 0. The n-th predicate removed either meets one of the conditions listed above, or is found to cover a position from which the second player moves to a position that has no successors covered by any predicate in the current strategy. Since the correct evaluation function also does not cover any of these successors, the n-th predicate covers a position which should be evaluated as

0, and is therefore correctly removed. □

Predicates removed are removed permanently, so the number of predicates in the current strategy can only decrease or stay the same. This is true across successive calls from the competitive algorithm since state is maintained. During learning each step of the competitive algorithm, at least one game must be lost (against one of the newly-added second-player strategies). So, at least one predicate in the current strategy is removed permanently. This bounds the total number of steps that may be taken by the covering competitive algorithm by at most P. Since there are 2^P first-player strategies, this is linear in the log of the size of the set of first-player strategies; this is true regardless of the second-player strategy space or strategy learning algorithm. □

We now address two issues suggested by this example: the construction of second-player strategy learning algorithms, and mistake-bounded game learning.

Second-player Learner

In Example 5, any second-player learner will suffice. Here, we provide a second-player strategy learning algorithm similar to that given for the first player. The second-player learner's task can be difficult. The second player is not guaranteed that it can force a win; this guarantee was used in the first-player strategy learning algorithm.

One way to get a to get a fairly general second-player strategy learning algorithm is to modify the game slightly (though the end product of the covering competitive algorithm is still a winning first-player strategy for the original game). In the modified game, the second player is allowed to choose the starting position. The second player must choose a starting position p such that:

1. It is the second player's turn to move from p.

2. p is the successor of a position that the first player evaluates as a win (is covered by a predicate in the current first-player strategy).

By the second condition, the first player still considers the proposed starting point to be a first-player win. The game begins from this position, and proceeds normally.

The second player represents strategies in a similar fashion as the first player, with a possibly different set of P_2 predicates. It is assumed that the correct evaluation function for the second player is representable by some disjunction of these predicates. During learning, the condition that is maintained for the first-player strategy learning algorithm is maintained here as well: any position not covered by any predicate is evaluated correctly as a loss for the second player.

The second-player strategy learning algorithm maintains a set of strategies, each of which proposes a different starting point. These starting points may be chosen beforehand, offline, as follows. Choose arbitrarily one starting point from each pairwise intersection $r_1' \cap r_2$, where r_2 is the set of positions covered by a second-player predicate and r_1' is the set of successors of positions covered by a first-player predicate. Any imperfect first-player strategy in our representation contains a predicate that incorrectly evaluates some positions as a first-player win. One of the representatives described above is in the successor set of this predicate; this is a starting point which a second-player strategy may legally propose and then force a win. Note that this implies that PP_2 perfect second-player strategies, with one starting from each of these representatives, form a teaching set.

For each representative, a second-player strategy is initialized (on the first step of the competitive algorithm) that proposes the representative as the starting position, and contains all predicates. On later steps each of these strategies is updated from the corresponding strategy from the previous step. Learning proceeds as for the first-player strategy learning algorithm, but first-player opponents that do not evaluate the proposed starting position as a win are ignored. For starting points that are actually first-player wins, learning may eventually remove too many predicates to allow moves to be made; choices can be made arbitrarily then. But, for any starting point that is actually a second-player win, the second-player strategy learning algorithm will be able to find a perfect strategy that defeats any current first player starting from that point, just as the first player can learn to defeat any set of second players starting from the won initial position. At least one of the PP_2 starting points must be incorrectly evaluated as a first-player win by an imperfect first-player strategy, so learning can always cover the passed first-player strategies with these starting points. On each call of the second-player strategy learning algorithm, at most P_2 passes through the opponents are required for each of the PP_2 strategies. After learning, the second-player strategy learning algorithm returns the current PP_2 strategies. Each opponent that was passed is defeated by at least one of these.

A perfect first-player evaluation function is the only strategy that can defeat every second-player strategy in this modified game. Note that the first-player strategy learning algorithm now has to deal with games that start from different positions, but the correctness of its predicate-removals still applies. Each step of the covering competitive algorithm still eliminates at least one predicate from the first-player strategy, so that the total number of steps required by the covering competitive algorithm is at most P.

Mistake-bounded Learning

The first-player strategy learning algorithm of Example 5 removes at least one predicate from its set each time it is defeated. This means that it can lose at most P times before it has an undefeatable strategy. This is true, regardless of the size of the space from which opposing strategies are being chosen, or how they are being chosen. It is even true independent of the specification number, which can get large for this class of games if second-player strategies are unrestricted. Also, this is accomplished without keeping track of all previous first and second player strategies. This is similar to mistake-bounded concept learning [59]. Since the mistake bound should be fairly small, one can even imagine doing final learning, with such a strategy learning algorithm, against expert human opponents.

This mistake bound works for this special type of representation and strategy learning algorithm. The fact that every loss forces significant learning progress is the key to proving that perfect strategies may be quickly learned. This is an unusual property of this class of strategies. For most other natural classes of strategies (for example, perceptrons that take predicate values as inputs), losses may be very uninformative. Against some opponents, the win can be an elusive "needle in a haystack" that requires exponential time to find. In such cases, efficient strategy learning is impossible. This seems to limit the possibilities for extending Example 5; different assumptions will be needed to progress towards more realistic learning algorithms.

II.E Computational Complexity

We consider the computational complexity of two versions of game learning. Unfortunately, our best efforts will still leave us in complexity classes that lie above P, but the results may be practically interesting for two reasons. First, there often exist heuristics that can solve instances of problems that are NP-hard in the worst case. Second, we may argue for the value of the techniques presented above by showing that they *reduce* the computational complexity of game learning.

II.E.1 Game Learning Without Considering k

We need to define games more carefully to understand the sort of computation that can be done on them. Assume that the first and second player strategies are n_1 and n_2 bits long, and that binary game outcome is given by a boolean formula f of length polynomial in n_1 and n_2. A decision-problem version of game learning can be formulated: is the game a first-player win? That is, if game outcome is given by $f(h, x)$, decide whether $\exists h \forall x f(h, x)$. But this is simply the canonical $\Sigma_2 P$-complete decision problem $QSAT_2$ [70], so game learning is $\Sigma_2 P$-complete. Similar problems in game theory have been related to the complexity class $\Sigma_2 P$ [58, 71].

Lemma 2 shows that, using our definition of a competitive algorithm, no competitive algorithm exists that solves every game in time polynomial in $\lg(|\mathcal{H}|)$, $\lg(|\mathcal{X}|)$, and k with worst-case strategy learning algorithms. It is natural to ask whether there exists *any* type of algorithm that can use worst-case strategy learning algorithms to solve any game in polynomial time. As a first step, remove the dependence on k; as described above, game learning is then $\Sigma_2 P$-complete. But, if there is an algorithm that uses worst-case strategy learning algorithms to solve games in time polynomial in $\lg(|\mathcal{H}|)$ and $\lg(|\mathcal{X}|)$, game learning would be in $\Delta_2 P$ (since the strategy learning algorithms could be replaced by an NP oracle); this would yield the unlikely conclusion that $\Sigma_2 P \subseteq \Delta_2 P$.

II.E.2 Complexity Accounting for k

The problem solved by the strategy learning algorithm is in NP (a formula containing several copies of f, with different opponents inserted, must be satisfied). Satisfying the (p, q)-randomization criterion is not much more difficult: the method of *almost uniform generation* [48] can be used to do this in randomized polynomial time with an NP oracle, as follows. Given a tolerance ϵ, an almost uniform generator outputs a solution y to the given decision problem with probability $u'(y)$, where u is the uniform probability and $(1+\epsilon)^{-1}u \leq u'(y) \leq (1+\epsilon)u$. This can be done with an NP oracle in randomized time polynomial in the input size and $\log(\frac{1}{\epsilon})$ [48]. We may reasonably meet the (p, q)-randomization criterion by, for example, setting $\epsilon = \frac{1}{2}$ to obtain $q = 1$ and $p = \frac{2}{3}$. This implies that games can be solved with an NP oracle in expected time polynomial in $\lg(|\mathcal{H}|)$, $\lg(|\mathcal{X}|)$, and k using the covering competitive algorithm. This result may be seen as giving a subset of problems in $\Sigma_2 P$ (those reducible to games with polynomially large k) which may be solved in randomized polynomial time with an NP oracle.

In addition, game learning with polynomially large specification number, is in $\Sigma_2 P \cap \Pi_2 P$. The problem is in $\Sigma_2 P$ since any instance of it is an instance of game learning with arbitrary specification number, which was shown to be in $\Sigma_2 P$. To see that it is in $\Pi_2 P$, reduce it to the following variant of $QSAT$ that is in $\Pi_2 P$: given a boolean formula f, $\forall y \exists z\ f(y, z)$.

Assume we are given a polynomial bound on specification number k. Note that, for purposes of the decision problem, the teaching set of size at most k must defeat *every* first-player strategy if no perfect first-player strategy exists, since at least one member of the teaching set defeats every imperfect first-player strategy. Given a boolean formula giving game outcome between a first-player and second-player strategy, construct the boolean formula f over variables representing a first-player strategy h and second-player strategies $x_1 \ldots x_k$. The formula f is TRUE iff none of $x_1 \ldots x_k$ defeats h. Now,

$$\forall x_1 x_2 \ldots x_k \exists h \; f(x_1, \ldots, x_k, h)$$

iff there exists no teaching set that never loses; iff there is some perfect first-player strategy that defeats the teaching set of size at most k that is guaranteed to exist.

This shows that the game learning problem, with polynomially large specification number, is in $\Pi_2 P$. Since it is also in $\Sigma_2 P$, it is in $\Sigma_2 P \cap \Pi_2 P$.

II.F Approximate Game Learning

In this section, we show how to learn mixed strategies that are guaranteed to win with high probability. The methods extend exact learning in two ways. First, they do not depend on specification number, so even when perfect strategies exist they can succeed where exact learning fails due to large specification number. Second, if perfect strategies do not exist, the methods can still find approximations to the optimal mixed strategy.

Once we deviate from exact learning, results may no longer apply to adversarial problems in general. Mixed strategies usually make sense for games, but may not make sense in other domains. For example, the goal of sorting network design (see Section IV.B) is a single small circuit that sorts correctly; it will usually make little sense to mix over a large number of approximately correct circuits.

The methods here are related to algorithms of interest in concept learning due to their connection to boosting [36]. The algorithms used in concept learning update a distribution over strategies by increasing the weight on strategies that compete successfully; this is similar to the competitive methods described here. But in concept learning applications, the "strategy spaces" are typically of small size, so that enumeration of all strategies is possible. The algorithms used in these applications rely on this, and are inefficient for the large games that we consider.

The methods described below can efficiently learn mixed strategies large games. These methods are primarily of theoretical interest, because they depend on an unlikely ability to sample complex mixed strategies from nearly uniform distributions. Because of this lack of realistic applicability, we do not attempt to describe the methods in terms of strategy learning algorithms. Instead, we use an NP oracle as in Section II.E to directly obtain complexity results.

II.F.1 Games and Mixed Strategies

A mixed strategy is a distribution over pure strategies that specifies the probability with which each one should be selected for play. For any zero-sum two-player game, the optimal mixed strategy for a player is the one that achieves the best possible payoff against the worst-case opposing strategy. The minimax theorem [96] shows that the worst payoff for the optimal first-player mixed strategy is the same as the worst payoff for the optimal second-player mixed strategy. This payoff gives the value V for the game.

Assume that we are given a game g, that takes as input a first-player strategy h represented with n_1 bits, and a second-player strategy x represented with n_2 bits. Assume g returns 1 or 0 indicating whether the first player or second player was the winner, respectively, in time polynomial in n_1 and n_2. The goal is to find a "small" mixed strategy (in the same sense used by Lipton and Young [58]), that randomizes uniformly over a polynomially large multiset of pure strategies and obtains a value for the game that is close to optimal.

We first consider approximate learning in the case where there is a perfect first-player strategy ($V = 1$), then extend it to games where $0 < V < 1$.

II.F.2 When a Perfect Strategy Exists

First, assume that a perfect strategy exists. For convenience, assume that it is a first-player strategy h_p: $\forall x\ g(h_p, x) = 1$. The goal of approximate learning is to find a small mixed strategy for the first player that wins most of the time. We use a sort of "majority" algorithm [19] that samples first-player strategies which defeat a set of second-player opponents. Then, it finds a second-player strategy that defeats a significant fraction of these. If no such second-player strategy exists, we are done. Otherwise, it probably eliminates a significant fraction of remaining first-player strategies. We give algorithm details, then the proof of performance.

Learning Algorithm

The algorithm is presented in Figure II.2. It relies on almost uniform generation, as did the results in Section II.E. Almost uniform generation samples from a set A_n, where n is the input size and membership in A_n can be decided in time polynomial in n. Let u be the uniform probability of sampling a particular element of A_n ($u = \frac{1}{|A_n|}$). Almost uniform generation chooses members y of A with probability $u(y)$, where $\frac{1}{1+\epsilon}u(y) \leq u \leq (1+\epsilon)u(y)$. This can be done in time polynomial in n and $\log(\frac{1}{\epsilon})$, with an NP oracle [48].

Steps 2 and 3 of the learning algorithm may be done with the aid of an NP oracle. Values for the constants are given below.

Performance

Note that if the above algorithm terminates, the mixed strategy obtained by sampling strategies uniformly from the (multi)set F wins at least $1 - \delta$ of the time against any possible opponent. We will bound the required time to obtain the following:

Theorem 8 *If a game with polynomial-time computable outcomes has a perfect first-player strategy, a mixed strategy that loses with probability at most δ against any*

1. Initialize S, the set of second-player opponents seen so far, to the empty set.

2. Use almost uniform generation with tolerance ϵ to sample a set F of m first-player strategies that each defeat all members of S.

3. Find a strategy s that defeats at least a fraction δ of F. Add s to S, or terminate if no such s exists.

4. Goto 2 if not finished.

Figure II.2: Approximate Strategy Learner

opponent may be found with an NP oracle in expected time polynomial in $\frac{1}{\delta}$ *and the strategy representation sizes.*

Proof: We need to show that, if some s is found in step 3, a large fraction of the remaining first-player strategies are probably eliminated.

Let H be the set of first-player strategies, in our encoding, that defeat all members of S. Each time step 3 above is executed, H shrinks. Before step 2 is executed, consider a second-player strategy x that defeats less than δ' of H; δ' is specified below. Since each member of H that x defeats has a probability of at most $\frac{1+\epsilon}{|H|}$ of being chosen on any given trial, the probability that a particular member of the sample F is defeated by x is at most $\delta'(1+\epsilon)$.

The choice of each member of F may now be treated as a Bernoulli trial, with probability of "success" (choosing a first-player strategy that x defeats) of at most $\delta'(1+\epsilon)$. Chernoff bounds may be used to upper bound the probability of an excessive number of successes [51]. Using a Chernoff bound, the probability that we obtain more than a fraction $\delta'(1+\epsilon)+\alpha$ of successes is at most $e^{-2m(\alpha^2)}$.

If we set $\delta = \delta'(1+\epsilon)+\alpha$, this Chernoff bound is an upper bound on the probability that a particular x defeating less than δ' of H isn't eliminated from consideration by the choice of F. The total number of x's that defeat less than δ' of

H is at most the total number of second-player strategies, which is 2^{n_2}. So, the total probability that *any* x defeating less than δ' of H is available after F is chosen is at most $2^{n_2}e^{-2m(\alpha^2)}$.

We want:

$$2^{n_2}e^{-2m(\alpha^2)} < p_{fail}$$

$$2m(\alpha^2) > -\log(p_{fail}) + \log(2^{n_2})$$

$$m > \frac{1}{2\alpha^2}\log(\frac{2^{n_2}}{p_{fail}})$$

With m chosen to be this large, we have a probability of at least $1 - p_{fail}$ of obtaining an F, such that any x that defeats at least $\delta'(1-\epsilon)+\alpha$ of it will defeat at least δ' of the remainder of H. The expected elimination is at least $\delta'(1-p_{fail})$ of H. Since the size of H cannot drop below 1, and $\lg(|H|) \leq n_1$, the expected number of steps to termination is $O(\frac{n_1}{\delta'(1-p_{fail})})$.

The final mixed strategy has a worst-case probability of winning of at least $1-(\delta'(1+\epsilon)+\alpha)$. If we desire $1-\delta$ to be equal to this, we may choose $\alpha = \frac{\delta}{2}$, $\epsilon = \frac{1}{2}$, and $\delta' = \frac{\delta}{3}$. Choose $p_{fail} = \frac{1}{2}$. Now, an adequate m, by the bound above, will be $O(\frac{n_2}{\delta^2})$. The total number of steps is $O(\frac{n_1}{\delta})$. Since steps 2 and 3 may be done in polynomial time with an NP oracle, the total expected time (with the NP oracle) is polynomial in n_1, n_2, and $\frac{1}{\delta}$. $\square$

II.F.3 When No Perfect Strategy Exists

Classes of pure strategies, as we have defined them, are exponentially large in parameters n_1 and n_2. An optimal mixed strategy may require an exponentially large representation. Lipton has shown, however, that small mixed strategies that randomize over polynomially many pure strategies may achieve a payoff close to optimal [58].

In particular, let $z = \frac{n_2}{2\beta^2}$, for some chosen β. Then, there exists a set of z strategies for the first player, such that the mixed strategy that chooses uniformly at random from these achieves a payoff of at least $V - \beta$ (for the game with binary payoffs as we have defined here).

Now, for a given β, let H^z be the first-player strategy set consisting of mixed strategies that randomize uniformly over z pure strategies from H, with z given by the above equation. Define a new game, in which a first-player strategy from H^z wins if it achieves a payoff of at least $V - \beta$ against the given second-player strategy. Game outcome can be computed in polynomial time, since the exact payoff can be computed from z game outcomes, with z polynomial in n_2.[9]

Since this new game has binary outcomes, and the first player in this new game has a perfect strategy, the algorithm from the previous section can be used to find an approximation to this "perfect" strategy. The approximation wins the new game at least $1 - \delta$ of the time, and can be found in time polynomial in n_1, n_2, and $\frac{1}{\delta}$ with an NP oracle by the result in the previous section. The resulting strategy randomizes over m mixed strategies (with m given as before), with each mixed strategy itself randomizing over z pure strategies. It obtains a payoff in the original game of at least $(1 - \delta)(V - \beta)$. Since z is polynomial in $\frac{1}{\beta}$, the overall algorithm runs in expected time polynomial in $\frac{1}{\delta}$, $\frac{1}{\beta}$, and the strategy representation sizes, with an NP oracle.

Theorem 9 *If a game with polynomial-time computable outcomes has a value V, a mixed strategy that obtains a payoff of at least $(1 - \delta)(V - \beta)$ against any opponent may be found with an NP oracle in expected time polynomial in $\frac{1}{\beta}$, $\frac{1}{\delta}$, and the strategy representation sizes.*

This improves a computational complexity result in [58], where it is shown that approximate game learning is in $\Sigma_2 P$. From the above, we may conclude that approximate game learning can be done in randomized polynomial time with an NP oracle.

[9] If V is not known in advance, it can be approximated by binary search: if a V that is too large is guessed, the learning procedure may fail by eliminating all strategies in H^z.

II.G Conclusion

We have shown the existence of a simple competitive algorithm that is able to successfully bootstrap its way to perfect strategies for a game under fairly general conditions. This gives some validity to the intuition behind the coevolutionary approach to adversarial problems.

This model may have applications in computational complexity. The covering competitive algorithm, and the approximate game learning algorithms in Section II.F, can be used to show that certain adversarial problems may be solved in randomized polynomial time with an NP oracle. This may reduce the known computational complexity of some problems that lie naively in $\Sigma_2 P$.

Future work may be able to bring the theory closer to practical coevolutionary and game learning algorithms. This may require a different, more detailed learning model. The strict assumptions made in the current model about the existence of strategy learning algorithms are difficult to satisfy in practice. Example 5 is a start towards providing a practical strategy learning algorithm, but there are severe limitations on possible extensions to this result. Approximate game learning should be a promising direction for obtaining more realistic results, but Section II.F is only a small step in this direction.

In the next chapter, we make use of several concepts from this theoretical approach in designing practical heuristics for adversarial problems. Teaching sets and specification number may be relevant even when we are not using theoretically complete algorithms. The covering competitive algorithm guarantees progress by ensuring that new strategies defeat all previous strategies. Simpler algorithms that lack this guarantee may fail to make progress. These basic observations may also be relevant in practice.

Chapter III

Heuristic Coevolutionary Methods

III.A Introduction

Chapter I discussed competitive coevolution as a heuristic method for adversarial problems. Coevolution refers to the simultaneous evolution of two or more species with coupled fitness. Within this broad context, a host species can be associated with other species in a number of ways, some which are mutually beneficial. In the context of adversarial problems, however, we focus on interactions between two species that are assumed to be strictly competitive. The fitness of an individual in one population is based on direct competition with some individual(s) from another population.

Individuals from each of the coevolving populations must take turns testing and being tested. In this chapter we use the term *host* to refer to the individual whose fitness is under consideration, and *parasites* to refer to the individuals that are testing the host (this terminology differs somewhat from Hillis' [44] and from biological terminology). In all experiments done here, two evolutionarily distinct populations (i.e. no exchange of genetic material between populations) are used, with competitions taking place between populations. One population contains candidate solutions to the adversarial problem, and the other contains test cases. Each population gets its turn as both hosts (when its fitness is being measured) and parasites (when it is being used to measure fitness).

Since the parasites are also evolving with a fitness based on a competition's outcome, the success of a host implies failure for its parasites. When the parasites evolve to overcome this failure, they create new challenges for the hosts; the continuation of this may lead to an evolutionary arms race. New genotypes arise to defeat old ones. New parasite types drive further innovation, creating ever-greater levels of complexity and performance by forcing hosts to respond to a wider range of more challenging parasite test cases.

The problem of testing candidate solutions can be considered separately from that of developing good solutions. A perfect solution to the testing problem is a minimal and complete set of extremely difficult test cases, and is typically very difficult to identify for realistic problems. Note, however, that such a test set may well make a poor parasitic population, even if we knew what it was. Hosts forced immediately to compete against such rigorous testing are all likely to fare poorly. Evolution requires information from fitness *variation* to guide search. Coevolution may provide this, too. In any generation, the parasites used will have varying degrees of evolutionary success, but none are likely to be far beyond the difficulty required to defeat the current hosts. As the generations progress, we can expect a drive for more and more difficult test cases to arise. The result should be a *pedagogical* series of parasites that gradually increase in difficulty as host solutions gradually increase in competence.

In this chapter, we test coevolutionary heuristics in the context of game-learning. 2-D Tic-Tac-Toe, Nim, and 3-D Tic-Tac-Toe are used as test problems. Several new methods for competitive coevolution are explored. "Competitive fitness sharing" changes the way fitness is measured. "Shared sampling" provides a method for selecting a strong, diverse set of parasites. The "hall of fame" encourages arms races by saving good individuals from prior generations. The "phantom parasite" is used to maintain balance between competitors in an arms race. Brood selection, a method from genetic programming [93], avoids excessive testing of poor offspring. We provide several different motivations for these methods, and mathematical insights into their use. Experimental comparisons are done, and a detailed analysis of these experiments is done in terms of testing issues, diversity, extinction, arms race progress measurements, and drift.

III.B Motivation

III.B.1 Goals of Competitive Coevolution

As described above, an implementation of competitive coevolution should provide for steady progress in the arms race. The genetic algorithm used must not only be strong in searching the particular domain, but also maintain adequate diverse sets of parasites for testing the opposing population. Some of our early experiments in competitive coevolution displayed repeated convergence on specialists that were poor test cases overall, with little arms race progress seen.

The use of diverse, strong opponents has been discussed in the context of game-playing [32, 91]. Here, we want diverse parasites at every stage, appropriate to the current level of the hosts. The parasites need to be diverse in such a way that they provide appropriate testing for *all* hosts in the current population. The parasites must be neither so difficult that the hosts rarely win, nor so easy that the hosts almost always win.

To these ends, we will describe several methods that encourage progress in the arms race and select strong individuals and appropriate test sets. First, we give two revealing viewpoints of competitive coevolution. Then, we will describe the methods, and analyze them mathematically and experimentally.

III.B.2 The Infinite Population View

Consider an evolutionary model similar to one used in [57]. Genotypes occupy a fraction of a population, so that these fractions sum to 1. Since these fractions can be any real number between 0 and 1, the population size is essentially infinite. Only a finite number of *distinct* genotypes are present in the population. New genotypes may enter via crossover and mutation of the old genotypes. Reproduction still takes place in discrete generations, but all genotypes are guaranteed some positive reproduction so that no genotype can ever become completely extinct.

This idealized model has several important features that differentiate it from a finite-population GA. First, no genotype will go extinct due to sampling errors. For example, innovations that exist in low numbers initially have a significant probability of being lost in a finite-population GA, but will not be lost in this model.

Second, consider a coevolutionary version of this model in which two distinct populations compete. Each genotype from one population competes with each genotype of the other, and receives fitness based on the fraction of the opposing population that it can defeat. Again, no genotypes ever completely disappear, so one that was not a good test earlier may rise again later if it is successful against a new opponent. A new genotype cannot be very successful (in terms of reproducing to the point of occupying a large fraction of the population) unless it is able to defeat most of the opponents ever seen in the opposing population.

Some of the methods described below can be thought of as attempting to capture these features in a finite-population setting.

III.B.3 Concepts from Theory

Chapter II discussed a theory of learning in competitive environments. In a simplified model, this theory shows that coevolution-like processes are capable of efficiently solving adversarial problems. This result depends on strong assumptions about the search algorithm used. It would be very difficult to prove that the GA satisfies these assumptions, so the theory cannot be directly applied to competitive coevolution. Nevertheless, it does shed light on desirable properties of competitive procedures.

It is important, for the sake of rapid convergence to perfect solutions, that new individuals (or sets of individuals) be found that are capable of defeating all of the prior individuals. Without such a guarantee, the system can get stuck on weak strategies that defeat each other in a cycle. If we have such a guarantee, we will be sure to progress towards optimality. Theoretical randomization criteria can be described under which such a method will discover perfect solutions in polynomial time (see Chapter II). For our purposes here, we should seek methods that progress by producing new strategies that defeat older ones (an arms race).

Define an "optimal solution" as one that defeats all possible opponents. By definition, only one side may have an optimal solution. The other side, without an optimal strategy, can only hope to defeat non-optimal opponents. A *teaching set* is a set of individuals that work together by having, for each possible non-optimal opponent, at least one individual in the set capable of defeating that opponent [42]. If no reasonably-sized teaching set exists, then optimal individuals cannot be practically distinguished, and coevolution cannot be expected to find optimal individuals. In general, small teaching sets seem to exist for many problems [42, 10].

For competitive coevolution, this suggests:

1. We should encourage some sort of niching, so that such *sets* of individuals may arise and be stable in a population.

2. The required size of samples, against which we test each individual, will be related to the size of teaching sets for the problem.

3. The method for choosing such samples should come as close to choosing a teaching set as is possible in the current evolutionary context.

III.C Methods

We now describe three methods for competitive coevolution based on the general issues discussed in the introduction and Chapter I, the infinite-population model that we attempt to approximate, and the theoretical arguments mentioned above. We experimentally test these methods below, separately and in combination with each other.

III.C.1 Competitive Fitness Sharing

The usual way to assign fitness based on success in a set of competitions is to sum the score (often a binary value indicating success/failure) across all competitions [8, 44, 12]. This measure will be called "simple fitness" here. In "fitness sharing" a sharing function is defined to take into account similarity (according to some metric) among individuals [41]. An individual's simple fitness is divided by the sum of its similarities with each other individual in the population, rewarding unusual individuals. Related to this is the "emergent fitness sharing" technique found to promote generalization in an immune system model [90]. and the "universal suffrage operator" that encourages diverse coverage of examples in a GA-based concept induction system [39].

Our method is similar to these techniques. In the "competitive fitness sharing" (henceforth, "fitness sharing" or "shared fitness") method we propose here, each parasite is treated as an independent resource to be shared by those hosts in the population that can defeat it. This is like having a separate sharing function for each parasite. The fitness assigned to a host defeating parasites with the set of indices X is $\sum_{j \in X} \frac{1}{N_j}$, where N_j is the total number of hosts in the population defeating parasite j. Fitnesses are most comparable when all hosts compete against the same set of parasites, and this is the case for all experiments here.

The effect of this method is to reward hosts that defeat parasites few others can, even though the rewarded host might not defeat *as many* parasites as others can. This is most relevant when there do not yet exist hosts that can defeat all of the parasites. In such a case, any host that defeats parasites that others cannot may well contain important genetic material.

As an example of the advantage that fitness sharing may provide, consider a situation in which a host type G defeats I parasites which no other host type can defeat. Assume that other host types defeat a greater number of parasites, on average. Thus, G is important in that it contains information about defeating parasites that no others can, but it defeats a below-average number of parasites overall. Assume G only had a small number of representatives in the population. This might be because it is a new innovation, or an old niche that recently became important. Or, it may simply be that random fluctuations carried its population to low numbers. Under simple fitness, G would have below-average fitness, and would stand a good chance of becoming extinct this generation (with exact probability depending on the type of selection). Under competitive fitness sharing, though, individuals of type G will receive high fitness, because they are rare: a total of I fitness will be divided among a small number of individuals. Other, more prevalent types will have lower per-individual fitness, because their fitness is being divided among a larger number of host individuals. Therefore, the small number of individuals of type G stand a good chance of being selected, preventing the immediate extinction of G. A more detailed analysis along these lines is provided in Section III.D.2. Note also that important genotypes such as G are never lost in the infinite population model; competitive fitness sharing helps ensure that they are not lost in the finite-population approximation.

Competitive fitness sharing is more able to allow the survival of important but poorly represented types. So, a greater number of diverse niches should be supportable under competitive fitness sharing. When many such niches exist they are necessarily small and subject to extremely low numbers due to sampling errors. Under simple fitness, there is a greater danger of extinction under these conditions than with competitive fitness sharing.

Recently, several authors have used a tournament format based on [90] with competitive coevolution. This format yields fitness with properties similar to those given by competitive fitness sharing. This has been successfully applied to the Prisoners' Dilemma [28] and the game of Dots-and-Boxes [98].

III.C.2 Shared Sampling

It is desirable to reduce the computational effort expended in competition by testing each individual in the host population against only a limited sample of parasites from the other population. One way to get a representative sample of parasites is to pick a random subset of them. This is one technique explored below.

However, more information is available about individuals from the previous generation, and if parasites are selected from this set, we can obtain a more challenging sample. For example, parasites with the best fitness (either simple or shared) in the last generation might be preferentially selected for play in this generation. If there are several niches in the parasite population, however, this would tend to pick similar individuals from the niche that happened to have highest fitness, rather than selecting a diverse set of parasites from all niches.

It would be preferable to choose parasites from the previous generation that challenge all segments of the host population. Given current information, we want to come as close as possible to choosing a teaching set. A natural way to do this is to use competitive fitness sharing for sampling, as we did for selection. New sample members are chosen to have maximal competitive shared fitness within the current sample. This is equivalent to choosing the sample from the population via truncation selection using competitive fitness sharing with *continuously updated sharing* [69]. In this way, each successive member of the sample is chosen to be one that competed well against individuals that the other members of the current sample did not compete well against. This technique will be called "shared sampling," and is specified in detail by the algorithm in Figure III.1. Note that the "opponents" there are the individuals that were sampled in the previous generation to rate the fitness of the parasite population currently being sampled.

This sampling technique helps to address the concern mentioned above (Section III.B.3) about providing teaching sets. In order to distinguish opponents on the basis of competition with a sample, the sample should contain tests that defeat any opponent. Otherwise, tests against this sample can make some opponent look unbeat-

```
Initialize current sample to be the empty set.

For each opponent i from the previous generation:
    beat[i]=0 /* beat[i] will count how many in current sample defeat i */

While the current sample is not yet full:
{
    /* Compute shared fitness each parasite would have if added to sample */
    For each parasite j not yet in sample:
    {
        samp_fit[j]=0 /* samp_fit[j] will hold the shared fitness of j */
        For each opponent k beat by j last generation:
        {
            /* If j were in sample, 1+beat[k] sample members would defeat k */
            /* So, 1/(1+beat[k]) is the shared fitness for j from opponent k */
            samp_fit[j] += 1/(1+beat[k])
        }
    }
    /* Add the best parasite to the sample */
    Let m be such that samp_fit[m] is maximal.
    Add parasite m to current sample.
    /* Update beat[] based on the addition of m to the sample */
    For each opponent i beat by m last generation:
        Increment beat[i]
}
```

Figure III.1: Shared Sampling

able. We expect the coevolutionary process to approach this ideal, having each sample include some losses against each opponent, as it proceeds. Shared sampling selects a set of strong individuals from the previous generation that would have provided a relatively complete set of tests.

It is interesting to note that shared sampling can also provide some of the effects of competitive fitness sharing described above, even when competitive fitness sharing is not being used. In addition to choosing parasites that challenge all hosts, shared sampling will tend to choose more representatives of parasite types that beat a large number of hosts. So, parasite types that are defeated by a large segment of the host population will generally be represented less in the sample than those defeated by few members of the host population. Therefore, even under simple fitness, hosts receive greater fitness for defeating parasites that few other hosts can defeat. This effect - that simple fitness with shared sampling can provide performance comparable to that of competitive fitness sharing - will be seen in some of the results below.

III.C.3 Hall of Fame

In the infinite population model, new hosts can only be very successful if they are able to defeat most or all of the old parasites, since these old parasites are still present. If the genetic operators can continue to find new genotypes in both populations that defeat all previous opposition, progress towards optimal individuals is guaranteed. This is clearly an important feature.

A finite population, in contrast, has a very short memory. A genotype must be successful almost every generation to stay in the population. Once gone, it can be difficult to rediscover. In using the GA for static optimization problems, this is not very significant because old genotypes were strictly worse according to the fitness measure. In competitive coevolution, however, such old genotypes might have become successful again. And, losing them from the population allows the opposing population to be successful with new types that would lose against old, extinct genotypes. The result is that progress towards the optimum is no longer guaranteed.

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. Hosts are tested against both current parasites, and a sample of the hall of fame. Successful new innovations cannot overspecialize; they are required to be robustly successful against old parasites.

We have experimented with performance-based methods of sampling the hall of fame. For example, we tried updating "fitness" of hall of fame members by playing them against the parasites for the current generation, and selecting a sample from the hall of fame based on this fitness. We have found the computational effort required to maintain performance information on members of the hall of fame is much greater than the benefit obtained over random sampling. Therefore, we use random sampling of the hall of fame in all experiments described here.

III.C.4 Phantom Parasite

Advantage in the competition, in terms of ability to defeat current opposition, may shift from one population to the other repeatedly during coevolution. This is usually not a problem; selection can continue to guide a population as long as there are hosts in it capable of defeating each opposing parasite. When, however, the populations are so far out of balance that there is some undefeatable parasite, coevolution may come to a halt. This undefeatable parasite will spread through its population, until the losing host population is unable to defeat any parasite. In this case, selection has no fitness information that can guide hosts towards defeating the opposition.

The losing host population needs easier, defeatable parasites that provide informative fitness to guide evolution. Evolution can produce improved hosts that defeat most of these easier parasites. If the easier parasites are related to the currently undefeatable parasite, then these improved strategies may also be able to defeat this difficult parasite. The easier parasites provide pedagogical stepping stones to the more difficult parasite.

Unfortunately, evolution in the opposing population may tend to discard such pedagogical opponents. Since they are defeatable, they receive lower fitness than the unbeatable parasite. Evolution under these conditions is discussed in more detail in Section III.G.6.

We can provide an explicit method for retaining pedagogical parasites when an undefeatable parasite exists, to help restore coevolutionary balance. A "phantom parasite" is added to each sample of current parasites. This phantom parasite has no explicit genotype, and no actual competitions are run between it and the hosts. Instead, it is used in conjunction with competitive fitness sharing to provide a niche for pedagogical individuals. Each generation, after computing competition outcomes against current parasites, the outcome against the phantom parasite is explicitly defined as follows:

1. Hosts that lose to some current parasite defeat the phantom parasite.

2. Hosts that defeat all current parasites lose to the phantom parasite.

Note that hall of fame parasites are not used in this context; defeat of an individual by a member of the hall of fame does not imply that the current opposition can overcome this same individual.

The phantom parasite only results in fitness differences when hosts are present that are perfect against all current parasites. In such a situation, only defeatable individuals that may be of pedagogical value receive fitness from this phantom parasite. Under competitive fitness sharing, such defeatable individuals are unlikely to become extinct since they are the only individuals capable of exploiting this resource. Under shared sampling, at least a few defeatable parasites are likely to be sampled since the phantom parasite cannot be defeated without them. Thus, when a difficult individual exists that cannot be defeated by current opposition, easier individuals of possible pedagogical value are likely to be retained and sampled. A simple analysis of the effect of the phantom parasite on fitness is presented in Section III.D.3.

Competition in the simple games used here tends to be fairly balanced. When competitive fitness sharing is used, both sides are usually able to retain adequate diversity to defeat any current opposition. Because of this, the phantom parasite rarely affects the outcome of coevolution on these problems. The use of the phantom parasite is more important in naturally asymmetric adversarial problems, where balance problems can be persistent. Section IV.A discusses such an asymmetric problem, and the effect of the phantom parasite is seen more readily there.

III.C.5 Brood Selection

Many biological species employ a reproductive strategy in which many (hundreds or more) offspring are produced from a single mating pair of parents. In a stable population, almost all of these offspring will die long before reproducing. So, there is competitive pressure for survival among the members of such a brood. Since many die very young, survival may be determined by a limited test with a significant element of chance.

Tackett [93] discusses this concept of *brood selection* in the context of genetic programming.[1] During reproduction, each pair of mating parents produces a relatively large number of offspring. A limited fitness test is applied to these offspring to determine which one makes it into the next population.

The limited fitness test used for testing brood members is usually much faster than the complete fitness test used for comparing population members. Brood selection therefore allows a greater number of individuals to be generated and tested each generation, with only a small increase in computation time. The method is likely to be useful when:

1. A fast, limited fitness test is available. This is often true when each individual is tested on a sample of test cases; a smaller sample may be used to obtain a faster, noiser test.

2. Difficult search requires many individuals to be tested each generation. If, for example, genetic operators are often destructive, or if fitness improvements are rare, brood selection may be able to efficiently test the large numbers of individuals that must be generated each generation.

[1] Genetic programming is the application of genetic algorithms to the evolution of programs[53].

In competitive coevolution, the first condition is always satisfied: we are always testing hosts against a sample of parasites. It is easy to use a smaller sample of parasites for brood testing. The natural fit of brood selection helps motivate its consideration here.

The second condition may hold for many adversarial problems, for example those in which representation and operators are complex. If many of the generated individuals are substantially worse, it doesn't make sense to do complete fitness tests on them; brood selection culls these poor performers before time is wasted on a complete test. In such problems, brood selection offers an efficient alternative to an increased population size.

Brood selection can have side effects on coevolutionary dynamics. Since brood selections prevents many poor performers from reaching the population, pedagogical individuals may be removed from an opposing population by the use of brood selection. This may have a greater negative impact on coevolution than the benefit obtained from brood selection, especially if the search problem is not difficult in a way that makes brood selection advantageous. So, we do not expect brood selection to be appropriate for every adversarial problem.

Our implementation of brood selection generates a fixed number of offspring (5 typically works well and is used for all brood selection experiments here) from each pair of parents. These are tested against parasites subsampled from the current ones without replacement. This subsampling is done separately for each brood, to prevent all chosen offspring from overspecializing against the same parasites. The number of parasites subsampled is chosen (somewhat arbitrarily) so that the number of competitions for all brood testing in a single generation is identical to the number of competitions required for full fitness tests of the population. The best member of the brood (according to simple fitness against the subsampled parasites) is admitted into the population.

III.D Analysis

III.D.1 Equivalence of Equilibria under Simple/Shared Fitness

Simple fitness directly selects for strong competitors. Shared fitness places emphasis on diversity, but it is difficult to understand its complete effect on coevolutionary dynamics. We would hope that it does not remove a long-term drive towards strong competitors.

We show here that the equilibria of coevolution are identical under simple and shared fitness. Equilibria are thus one element of long-term coevolutionary dynamics that is preserved when we move from simple fitness to shared fitness. This result suggests that competitive fitness sharing does not sacrifice long-term gains to preserve diversity in the short term.

We must make several simplifying assumptions to enable this analysis. To avoid roundoff problems, we assume an infinite population, with each of a finite number of genotypes occupying a fraction of the population. To avoid fitness noise, we assume full round-robin competition. Under realistic GA dynamics, noisy fitness and selection along with the creation of new genotypes by crossover and mutation would prevent populations from ever permanently reaching equilibrium. In the simplified GA considered here, equilibria are states in which all individuals have equal fitness; if the real GA does tend toward any "nearly equilibrium" points, they must resemble such equal-fitness states.

Definitions and Equilibrium Conditions

Each genotype in each population occupies a certain fraction of that population. These fractions sum to 1 in each population. The vector of these fractions for the first population will be denoted X, with components x_i, and Y for the second population, with components y_j. A first-population genotype will be denoted i and a second-population genotype j.

The matrix which defines the binary competition outcomes is denoted B, with component b_{ij} equal to 1 if genotype i defeats j, 0 otherwise.

Simple fitness for a host is given by the fraction of the parasites defeated by that host. Since in equilibrium, all fitnesses will be equal, the conditions for simple equilibrium can be expressed as:

$$BY = C_1$$

$$X^T(1 - B) = C_2^T$$

All vectors are column vectors. The T denotes matrix transpose, and 1 will always refer to the all-1 matrix, column vector, or scalar, as appropriate. C_1 and C_2 are constant vectors with all components c_1 and c_2 respectively.

Shared fitness for a host is the sum, over all parasites defeated, of the fraction occupied by that parasite in its population over the the fraction of the host's population that defeats that parasite. Linear algebra notation can be inconvenient for expressing this. To help, adjusted vectors $\hat{X}$ and $\hat{Y}$ will be defined. Component i of $\hat{X}$ is given by $\frac{x_i}{((1-B)Y)_i}$, and component j of $\hat{Y}$ is given by $\frac{y_j}{(X^TB)_j}$, where the subscripts indicate a particular component of the vector. These adjusted vectors give the fitness contribution from each parasite genotype. So, the shared equilibrium conditions can be expressed as:

$$B\hat{Y} = 1$$

$$\hat{X}^T(1 - B) = 1^T$$

These conditions express the fact that all fitnesses must be equal to 1 at shared equilibrium. No genotype can be undefeated (since it could not then be in equilibrium with other genotypes), so each parasite population contributes a total of 1 fitness to the opposing host population. The total of the fractions of the population that each host occupies is 1, so the average fitness must be 1. Since all fitnesses are equal in equilibrium, all fitnesses are 1. Note that, to show that the shared equilibrium conditions are satisfied, it is sufficient to show that all fitnesses are equal; equality with 1 follows by the condition on average fitness that is always satisfied (even away from equilibrium).

Note that since the elements of X sum to 1, $(1-B)X = 1 - BX$, and $X^T(1-B) = 1^T - X^T B$, and similarly for Y.

Equivalence of Equilibria

Lemma 10 *Any simple equilibrium is also a shared equilibrium.*

Proof: First, assume that a simple equilibrium exists, given by $X = V_1$ and $Y = V_2$ (so that $BV_2 = C_1$ and $V_1^T(1-B) = C_2^T$). Then $(1-B)V_2 = 1 - BV_2 = 1 - C_1$, another constant vector with all components equal to $1 - c_1$. And, $V_1^T B = 1^T - 1^T - V_1^T B = 1^T - V_1^T(1-B) = 1^T - C_2^T$, a constant vector with all components equal to $1 - c_2$. So, taking $X = V_1$ and $Y = V_2$, $\hat{X} = \frac{V_1}{1-c_1}$, and $\hat{Y} = \frac{V_2}{1-c_2}$. Thus,

$$B\hat{Y} = (BV_2)(\frac{1}{1-c_2}) = C_1 \frac{1}{1-c_2}$$

a constant vector, and

$$\hat{X}^T(1-B) = V_1^T(1-B)(\frac{1}{1-c_1}) = C_2^T \frac{1}{1-c_1}$$

another constant vector. So, the conditions for a shared equilibrium are also satisfied. $\square$

Lemma 11 *A shared equilibrium is a simple equilibrium.*

Proof: Assume not: we have an assignment to X and Y that is a shared equilibrium but not a simple equilibrium. Consider this shared equilibrium. Let w refer to any genotype receiving lowest simple fitness in the first population, with w' defined similarly for the second population. Let b and b' be genotypes in the first and second populations receiving highest simple fitness.

Denote simple fitness of a genotype g by $si(g)$, and shared fitness by $sh(g)$. We have

$$sh(g) = \sum_j \frac{b_{gj} y_j}{\sum_k b_{kj} x_k}$$

for g in the first population, and

$$sh(h) = \sum_i \frac{(1 - b_{ih})x_i}{\sum_k (1 - b_{ik})y_k}$$

for h in the second population. These can be expressed as:

$$sh(g) = \sum_j \frac{b_{gj}y_j}{1 - si(j)}$$

$$sh(h) = \sum_i \frac{(1 - b_{ih})x_i}{1 - si(i)}$$

Now, assume (towards a contradiction) that w defeats only genotypes with simple fitness smaller than $1 - si(w)$. Then

$$sh(w) < \sum_j \frac{b_{wj}y_j}{1 - (1 - si(w))}$$

$$sh(w) < \frac{1}{si(w)} \sum_j b_{wj}y_j$$

$$sh(w) < \frac{1}{si(w)} si(w)$$

$$sh(w) < 1$$

which is a contradiction, since we assumed that we have a shared equilibrium, in which it must be the case that $sh(x) = 1$ for all x. So, at least one genotype defeated by w must have simple fitness of at least $1 - si(w)$. Since at least one genotype in the population opposing w has fitness at least this large, $si(b') \geq 1 - si(w)$. Symmetrically, at least one genotype defeated by w' must have simple fitness at least $1 - si(w')$, so that $si(b) \geq 1 - si(w')$.

Proceeding similarly, assume that b defeats only genotypes with simple fitness larger than $1 - si(b)$. Then

$$sh(b) > \sum_j \frac{b_{bj}y_j}{1 - (1 - si(b))}$$

$$sh(b) > \frac{1}{si(b)} si(b)$$

which is again a contradiction. So, at least one genotype defeated by b must have simple fitness of at most $1-si(b)$. This implies that $si(w') \leq 1-si(b)$. Symmetrically, $si(w) \leq 1 - si(b')$.

We have $si(b) \geq 1 - si(w')$, which implies $si(w') \geq 1 - si(b)$. Together with $si(w') \leq 1-si(b)$, this means that $si(w') = 1-si(b)$. Symmetrically, $si(w) = 1-si(b')$.

Now, w may only defeat genotypes with simple fitness equal to or less than $1 - si(w)$. If w defeats only genotypes with simple fitness equal to $1 - si(w)$, then:

$$sh(w) = \sum_j \frac{b_{wj} y_j}{si(w)}$$

$$sh(w) = \frac{1}{si(w)} si(w) = 1$$

If w defeats any genotypes with simple fitness less than $1 - si(w)$, then at least one term in the above sum will decrease, and none will increase. This would give $sh(w) < 1$, a contradiction. So, w only defeats genotypes with simple fitness equal to $1 - si(w)$. Similarly, w' defeats only genotypes with simple fitness $1 - si(w')$.

Proceeding similarly, we know that b may only defeat genotypes with simple fitness equal to or greater than $1 - si(b)$. If b defeats only genotypes with simple fitness equal to $1 - si(b)$, then:

$$sh(b) = \sum_j \frac{b_{bj} y_j}{si(b)}$$

$$sh(b) = \frac{1}{si(b)} si(b) = 1$$

If b defeats any genotypes with simple fitness greater than $1 - si(b)$, then at least one term in the above sum will increase, and none will decrease. This would give $sh(b) > 1$, a contradiction. So, b only defeats genotypes with simple fitness equal to $1 - si(b)$. Similarly, b' defeats only genotypes with simple fitness $1 - si(b')$.

Since we are assuming that we do not have a simple equilibrium, assume without loss of generality that the first population does not have all equal simple fitness. Call the portion of the first population with simple fitness less than $si(b)$ the *remainder*. No genotype in the remainder can lose to any genotype with simple fitness $si(w')$ (since genotypes with simple fitness $si(w')$ can only defeat genotypes with fitness $1 - si(w') = si(b)$). So, each genotype in the remainder defeats every opponent with simple fitness $si(w')$. Since genotypes in the first population with simple fitness $si(b)$ can only defeat opponents with simple fitness $si(w')$, each genotype in the remainder defeats all of the opponents that any genotype with fitness $si(b)$ defeats. So, genotypes in the remainder receive simple fitness of at least $si(b)$, which contradicts the definition of the remainder.

Thus, there cannot exist a shared equilibrium that is not a simple equilibrium. $\square$

From these two lemmas, we get:

Theorem 12 *Given particular sets of first and second population genotypes, the sets of simple equilibria and shared equilibria are the same.*

III.D.2 Extinction Probability

Assume that out of all parasites tested against in a particular generation, there were O parasites, each of which had at least one host capable of defeating it. The goal in this section is to find the probability that, for at least one of these O "defeatable parasites," selection does not actually choose any host capable of defeating it.

Probabilities will be derived here assuming that fitness proportional selection is used. The result should be comparable under tournament selection, but the analysis is more difficult. A similar, far more extensive analysis has been done for fixed fitness functions with conventional fitness sharing [61].

First, assume that competitive fitness sharing is being used. The total fitness received by the host population is O, since each defeatable parasite contributes a total of 1 fitness to the host population, divided evenly among those hosts able to defeat it. So, a host with fitness f has probability $\frac{f}{O}$ of being selected during the fitness proportional selection of a single individual. If some set of hosts has total fitness F, the probability of selecting a member of this set is $\frac{F}{O}$.

Since each defeatable parasite contributes a total of 1 to those hosts able to defeat it, the hosts capable of defeating it must have a total fitness of at least 1. So, for each defeatable parasite, the probability of selection choosing a host capable of defeating that parasite is at least $\frac{1}{O}$. If N hosts are selected for a new population, the probability of not selecting any opposition for a particular parasite is at most $(1-\frac{1}{O})^N$. So, the probability of not selecting any opposition for at least one parasite is at most $O(1-\frac{1}{O})^N$. As an example, for $N = 1000$ and $O = 100$, the probability of such an extinction is less than 0.005.

Under simple fitness, the probability of not selecting any opposition for at least one defeatable parasite can be much higher. To take a worst case, consider a situation in which every defeatable parasite except for one, o_0, is defeated by every member of the host population except for one, p_0. p_0 defeats o_0; this is what makes o_0 defeatable. Now, if the host population has N members, $N-1$ of them receive fitness $O-1$, and 1 of them receives fitness 1. Each selection, host p_0 has a probability of $\frac{1}{(N-1)(O-1)+1}$ of being chosen. In selecting N hosts, p_0 has a probability of $(1-\frac{1}{(N-1)(O-1)+1})^N$ of not being chosen at all. Since N and O will generally be large, this probability can be very high. In our example with $N = 1000$ and $O = 100$, this probability is about 0.99.

This shows that loss of all opposition for a defeatable parasite is unlikely under competitive fitness sharing, but that no such guarantee is possible with simple fitness. Extinctions are examined empirically in Section III.G.3.

III.D.3 Fitness with Phantom Parasite

In this section, we derive a simplified mathematical description of the effect of the phantom parasite. Consider a population of size N in which there exist "perfect" hosts that defeat all current parasites (such that the phantom parasite is relevant). To make the analysis simple and results comprehensible, assume that the remainder of the population consists of L imperfect hosts that all lose to the same k parasites ($k > 0$). We wish to know, given k, how small L must become before the imperfect hosts gain a fitness advantage (due to the phantom parasite) over the perfect ones. Ignore hall of fame parasites for purposes of this analysis; they could shift the balance somewhat, but results would be qualitatively similar.

Let P denote the total number of current parasites, except for the single phantom parasite. Since there are $N - L$ perfect hosts, each receives fitness $\frac{1}{N-L}$ for each of the k parasites that are not defeated by imperfect hosts, and $\frac{1}{N}$ for each of the $P - k$ parasites that are defeated by imperfect hosts. Total fitness for a perfect host is thus $\frac{k}{N-L} + \frac{P-k}{N}$.

Imperfect hosts each receive fitness $\frac{1}{L}$ from the phantom parasite, and fitness $\frac{1}{N}$ from each of the $P - k$ parasites they defeat. This gives a total fitness of $\frac{1}{L} + \frac{P-k}{N}$.

To find the point where imperfect hosts gain a fitness advantage over perfect hosts, solve:

$$\frac{1}{L} + \frac{P-k}{N} > \frac{k}{N-L} + \frac{P-k}{N}$$

$$\frac{1}{L} > \frac{k}{N-L}$$

$$N - L > kL$$

$$L < \frac{N}{k+1}$$

This sets the approximate point at which perfect hosts and imperfect hosts are balanced in a population when the phantom parasite is used. For example, if imperfect hosts lose to a single parasite ($k = 1$), then they gain a fitness advantage when they occupy less than half of the population. Even when imperfect hosts lose to *all* parasites ($k = P$), they are still likely to be maintained in a large population since they gain a fitness advantage when their numbers fall below $\frac{N}{P+1}$.

Without the phantom parasite, imperfect hosts would always be at a fitness disadvantage when perfect hosts exist, and would be likely to become extinct. If this happens, the population could offer no informative fitness information to guide evolution in the opposing population. With the phantom parasite, imperfect individuals are retained. These may provide pedagogical information to the opposing population, to guide its evolution towards eventual defeat of the "perfect" host.

III.E Experimental Setup

Experiments were done to explore the effectiveness of the new methods described above. These experiments were done on three games. The first of these is the familiar game of 2-dimensional Tic-Tac-Toe (2DTTT), with about 600 relevant nodes in its game graph after removal of symmetries. The second is a small version of Nim, also with about 600 nodes in its game graph. The small size of these two games allows for simple representations and understandable results. Additional experiments used a much larger game, 3-dimensional Tic-Tac-Toe (3DTTT) on a 3x3x3 board, with a relatively complex strategy representation.

III.E.1 Representation of Game Strategies and Genetic Operators

2DTTT

In 2DTTT, players alternate placing their markers on a 3x3 grid. The first player to obtain 3 in a row horizontally, vertically, or diagonally wins. Games frequently end in a draw (which is the result of optimal play on both sides).

The representation used for 2DTTT has each gene corresponding to a single position in the game. Symmetric positions are removed, and the genes are ordered on the genome in the following, rather arbitrary way. A board configuration may be represented as a 9 digit base-3 number, with each digit being 0 to indicate that the square is empty, 1 to indicate that the opponent holds the square, and 2 to indicate that the moving player holds the square. All positions are generated in numerical order starting from 0 (all squares empty). Each position is assigned to a gene if: it is reachable in actual play, does not contain a win, has at least two squares open (so that a choice needs to be made), and does not have a symmetric form (reflected and/or rotated) already present in a previous gene. This yields a genome with 593 positions.

In 2DTTT, a *control function* is learned. Each gene can take on a number of allele values equal to the number of open squares in the corresponding position. This uniquely specifies a legal move from the position.

2DTTT is a simple game and coevolution has a fairly easy time solving it. We use it here partly as a control, to check that the new methods for competitive coevolution do not substantially hamper performance on an adversarial problem that is easy to begin with.

Nim

In Nim, a position in the game consists of several piles of stones. Players alternate removing an arbitrary number of stones from a single pile. The player to take the last stone wins. The configuration used here starts with 4 piles, containing 3, 4, 5, and 4 stones. This configuration allows the first player to force a win with optimal play [18].

A simple one-gene-per-position representation is used for Nim. The positions are placed on the genome in lexicographic order: (0,0,0,0), (0,0,0,1), ..., (3,4,5,4), where each quadruple gives the number of stones in each of the four piles. Actually, (3,4,5,4) is not needed in this representation; since this is the initial position, it never needs to be evaluated (see below). This leaves a total of 599 positions on the genome.

A binary *evaluation function* is used: each gene can take allele values 0 and 1. To make a move, legal moves are tested using the evaluation function, and the first position evaluated as 1 is selected as the move. Positions are tested in the following order. Piles are selected in order from first through fourth. For each pile containing $k > 0$ stones, moves leaving the pile with 0 up to $k - 1$ stones are tested in order. If none of these positions is evaluated as 1, the first move in this ordering (not generally a good move) is selected by default.

Traditional crossover and mutation are used, with a 0.004 probability of crossover at each gene, and 0.0022 probability of mutation at each gene. These were chosen to give averages a bit over 2 crossovers and 1 mutation per generated individual.

Though this problem is easily solved by game-tree search, this particular form of it is somewhat more difficult because the GA has no knowledge about the game tree and cannot directly search it. Also, the GA needs to find a strong set of opposing strategies to use as test cases (a teaching set) in order to solve the game. Random sampling in the space of test cases is inadequate for this game (see Section III.G.2).

3DTTT

In this game, players alternate placing markers ("X" and "O") on a 3x3x3 grid. The first player to obtain 3-in-a-row on any row, column, or diagonal (face diagonal or cube diagonal) wins. The game is a very easy first-player win if it is not restricted, so we disallow the first player from making the initial move at the center location. The game is then more interesting, and becomes a second-player win.

There are far too many distinct positions in this game, even with symmetries accounted for, to use as straightforward a representation as was used for Nim and 2DTTT. Instead, strategies are represented as an ordered list of condition/action pairs. A condition is a set of simple constraints on the current position. Each constraint specifies one of the 27 locations in the cube, and what must be at that position (friendly marker, enemy marker, or empty). The action then specifies a move.

To make a move with such a strategy during play, the rules in the list are checked in order starting from the first rule in the list. The first matching rule with a currently legal action (unoccupied point in the cube) is allowed to move. Conditions are checked in all possible symmetric configurations (rotations and reflections of the cube; 48 total). Only one symmetry needs to match; if it does, the same symmetry is applied to the action.

The genetic operators are applied directly to this representation. Crossover takes place between rules only. Since rule lists may be variable length, the length of the child is chosen randomly from the interval between the parent list lengths (inclusive). Crossover can produce individuals with length different from that of either parent, but not a length that is longer than both parents or shorter than both. After choosing the child length, uniform crossover is done: child rule i (numbered in sequence down the list starting from the first rule) is set to the first parent's rule i with probability $\frac{1}{2}$, and to the second parent's rule i with probability $\frac{1}{2}$. If only one parent has a rule at position i (because the other parent has length less than i), the child's rule is taken from that parent.

Since crossover does not change rules, a variety of mutation types are used to get a reasonable range of rule variation. A single rule's condition can have a single random constraint added or deleted. A single entire rule can be deleted, or randomly created and inserted, at a randomly chosen place in the list; this decreases or increases the length of an individual by one rule. A single action can be mutated, and a single pair of rules can be swapped.

Offspring were created via crossover with 50% probability, and via copying with 50% probability, before applying any mutation. A mutation operator was performed on each child with 90% probability. When a mutation operator was to be applied, one of the six operators described above was chosen uniformly at random. Since the probability of adding a rule and removing a rule are the same, and the expected length of a child produced from crossover is the average of the parent lengths, the average list length in the population should not change due to application of the genetic operators alone. Selection is required to produce changes in average length in the population.

Rules were initialized with the number of constraints chosen randomly from the range $[0-3]$. The number of rules in each individual in the initial population was chosen randomly from the range $[1-8]$. The maximum number of rules was set at 30, and the maximum number of constraints in a rule was set at 8.

Since 3DTTT is a large game, we expect it to be difficult. The complex representation and operators make it hard to generate improved individuals; brood selection may be valuable in this context.

III.E.2 Genetic Algorithm Details

For both Nim and 3DTTT, the first population contains first-player strategies, and the second contains second-player strategies. For 2DTTT, the "perfect strategy" should never lose, going both first and second. In this case, all host/parasite competitions involve two games, with each player getting a chance to go first and second; under competitive fitness sharing, a parasite is treated as two independent shared resources, when it goes first and when it goes second. In each of these games, the two populations use the same representation, but the requirements of the populations are different since the strategies play different roles in the game.

We have found tournament selection to work well in competitive coevolution; parents are selected in independent 2-tournaments. Elitism is important in these problems. If it is not used, an optimal individual may be found and then lost[2]. Additionally, elitism may work well in conjunction with fitness sharing to protect important innovations. An elite size of 1 was used in 2DTTT in Nim, and 20 in 3DTTT. Each population had 400 members in 2DTTT, 500 members in Nim, and 1000 members in 3DTTT.

Sampling was done without replacement (though multiple copies of an individual may appear in a sample if multiple copies appear in the population or hall of fame being sampled). In initial generations, the hall of fame does not contain enough distinct individuals to obtain a large enough sample, so the hall of fame sample is filled out with random individuals as necessary for these first few generations.

The use of standard fitness sharing with tournament selection can lead to fluctuations in niche sizes. The method of *continuously updated sharing* has been developed to overcome this problem [69], This method involves calculating shared fitness within the next generation's population, currently being constructed, rather than in the original population. This resembles the method of shared sampling used here. Continuously updated sharing was tried here with competitive fitness sharing,

[2]To ensure that optimal individuals can be selected as elitists, simple fitness is used for elitism when the phantom parasite is used. Otherwise, fitness loss from failure to defeat the phantom parasite might prevent optimal individuals from being consistently chosen as elitists.

on Nim. It was not found to be helpful. The fluctuations discussed in [69] may be less of a problem due to the presence of a large number of niches, and the fact that competitive fitness sharing sums fitness over many components. These factors would tend to reduce the problem that leads to fluctuations: one large niche winning all selection tournaments against another large niche.

To assess the impact of each new method with a reasonable number of experiments, we do series of experiments that add one method at a time in the following order:

1. Competitive fitness sharing
2. Shared sampling
3. Hall of fame
4. Phantom parasite
5. Brood selection

This ordering reflects the relative importance of these methods, and a few precedence relationships. First, competitive fitness sharing and shared sampling have the most important effect on the short-term behavior of coevolution, and the hall of fame has the most important long-term effect. Second, it does not make sense to use the phantom parasite without competitive fitness sharing and shared sampling, since its effect of creating a niche for pedagogical parasites will not be realized. Finally, we want to consider the complete range of coevolutionary methods before adding brood selection, which may improve search in the individual populations but has only incidental effects on coevolutionary interactions.

Additional experiments are done to explore the effect of several of the methods in isolation. The exact combinations of methods used, and the sample sizes used for testing, will be given for each of the test problems below.

III.F Comparative Results

III.F.1 2DTTT

For 2DTTT, simple fitness and shared fitness were tested:

1. with random sampling of 40 parasites; and

2. with shared sampling of 40 parasites.

Beyond this group of four experiments, the hall of fame, the phantom parasite, and brood selection were added successively. When the hall of fame was used, 20 parasites were sampled from it along with 20 parasites from the current opposition; this conserves the total number of 40 parasites. The brood size is 5, with each brood member tested against a subsample of 8 parasites.

Performance is measured as the number of games required to obtain a perfect first population strategy (that never loses, going first or second). That is, the genetic algorithm is run until a perfect strategy is found, and a count is made of the total number of games played during fitness and brood evaluations. In the case of 2DTTT, we count game-pairs since each host plays each parasite twice, once going first and once going second.

Table III.1 shows results for the variants studied. The **Hall** column has a "*" when the hall of fame was used, **Ph** has a "*" when the phantom parasite was used, and **Br** has a "*" when brood selection was used. The numbers in parentheses under **Sampling** and **Hall** show how many current and hall of fame parasites were used, respectively. **Reps** gives the number of runs of that type, and **Success** gives the percentage of these runs that found optimal solutions. **Avg** gives the average performance (in millions of games), and **S.D.** gives the standard deviation in performance.

To test significant differences in performance, we use a Kruskal-Wallis rank test at the 5% nondirectional level. This test allows us to detect a significant difference among a set of samples, then locate the pairwise significant differences with a set of

No.	Fitness	Sampling	Hall	Ph	Br	Reps.	Success	Avg	S.D.
(i)	Simple	Random (40)				10	100%	8.34	1.81
(ii)	Simple	Shared (40)				10	100%	4.57	1.37
(iii)	Shared	Random (40)				10	100%	4.75	0.93
(iv)	Shared	Shared (40)				10	100%	4.60	1.18
(v)	Shared	Shared (20)	* (20)			10	100%	6.55	4.11
(vi)	Shared	Shared (20)	* (20)	*		10	100%	4.83	1.06
(vii)	Shared	Shared (20)	* (20)	*	*	10	100%	5.02	1.12

Table III.1: Number of Game-Pairs to Solve 2DTTT (in Millions)

	(i)	(ii)	(iii)	(iv)	(v)	(vi)	(vii)
(i)	X	*	*	*	*	*	*
(ii)		X	-	-	-	-	-
(iii)			X	-	-	-	-
(iv)				X	-	-	-
(v)					X	-	-
(vi)						X	-
(vii)							X

Table III.2: Significant Differences ($p < 0.05$) in Number of Games to Solve 2DTTT

post hoc tests [35]. This test is used for all significance tests in this chapter. In each case, some significant difference exists among the samples, so we will only report the pairwise significant differences. For 2DTTT, significant differences are shown in Table III.2. A "*" indicates a significant difference.

2DTTT is easy for coevolution, even without the new methods. The new methods improve performance: the runs that used none of the new methods did significantly worse than all of the runs that used new methods. No other differences are significant. This is appropriate, since this problem was largely included as a control to ensure that the new methods do not hamper coevolutionary performance on an easy problem.

The hall of fame seemed to hurt slightly, although the difference is not significant. This may be because coevolution is so fast on this problem; the 20 parasites from the hall of fame may be unnecessary and largely wasted. For such an easy problem, it may be adequate and even slightly superior to use only current parasites.

Brood selection has little effect for 2DTTT. This is to be expected, since the representation is simple and search is straightforward. Most offspring of highly-fit parents should be reasonably fit, so that there is little need for brood selection. Note, however, that coevolutionary performance is robust to the addition of brood selection: since each generation requires twice as many games, coevolution is taking about half as many generations.

III.F.2 Nim

For Nim, simple fitness and shared fitness were tested:

1. without sampling (using entire parasite population);
2. with random sampling of 100 parasites; and
3. with shared sampling of 100 parasites.

The hall of fame was tried in isolation, with simple fitness and random sampling. Finally, the hall of fame, phantom parasite, and brood selection were added successively to sharing and shared sampling. A brood size of 5 was used, with each brood member tested against 10 parasites. Details are given in Table III.3. Performance is measured as the number of games required to obtain a perfect first player strategy.

Most of the runs that used sampling used the same total number of parasites (100). Smaller numbers of parasites sometimes worked, but performance was not as consistent when this was done. An exception to this arose when the hall of fame was used in conjunction with sharing and shared sampling. In this case, a solution may be consistently found with a smaller number of parasites, so a total of only 50 were used.

Table III.3 shows results for the variants studied. When sampling is not used, each game is used twice: once with the population 1 individual playing host and once with the population 2 individual playing host; the results given in this case reflect the fact that each such game is only played and counted once, with the outcome used for both purposes.

No.	Fitness	Sampling	Hall	Ph	Br	Reps	Success	Avg	S.D.
(i)	Simple	None (500)				5	0%	($\gg$ 3000)	-
(ii)	Simple	Random (100)				5	0%	($\gg$ 3000)	-
(iii)	Simple	Shared (100)				10	100%	201	60.6
(iv)	Shared	None (500)				10	100%	806	330
(v)	Shared	Random (100)				10	100%	353	116
(vi)	Shared	Shared (100)				10	100%	145	27.5
(vii)	Simple	Random (50)	* (50)			10	100%	1040	714
(viii)	Shared	Shared (25)	* (25)			10	100%	73.0	15.6
(ix)	Shared	Shared (25)	* (25)	*		10	100%	67.0	14.0
(x)	Shared	Shared (25)	* (25)	*	*	10	100%	65.7	18.1

Table III.3: Number of Games to Solve Nim (in Millions)

	(i)	(ii)	(iii)	(iv)	(v)	(vi)	(vii)	(viii)	(ix)	(x)
(i)	X	-	*	*	*	*	*	*	*	*
(ii)		X	*	*	*	*	*	*	*	*
(iii)			X	*	*	*	*	*	*	*
(iv)				X	*	*	-	*	*	*
(v)					X	*	*	*	*	*
(vi)						X	*	*	*	*
(vii)							X	*	*	*
(viii)								X	-	-
(ix)									X	-
(x)										X

Table III.4: Significant Differences ($p < 0.05$) in Number of Games to Solve Nim

Runs using simple fitness, without sampling and with random sampling, never completed. These were run for 3 billion games (12000 generations and 30000 generations, respectively), and though there was initial progress, these runs showed no hope of eventual success.

Since the Kruskal-Wallis significance test works with ranks, we can sensibly include runs that did not complete; these are all ranked identically as the worst (since no run that completed took longer than these were run). Significant differences are shown in Table III.4.

The game cannot be solved when none of the new methods are used. Using any one of the three most important methods is adequate to eventually solve the game. Using methods in combination allows significantly faster solutions, with the use of all methods being fastest overall. The effect of the phantom parasite is marginal; it rarely makes any difference since extreme balance problems are rare for this game. The effect of brood selection is also marginal, but it does reduce the number of required generations so that performance does not suffer.

III.F.3 3DTTT

For 3DTTT, each of the two populations contained 1000 members[3]. Runs were stopped when a perfect second-player strategy was found. New methods were added successively as described above, for a total of six experiments. A total of 100 parasites were used in each experiment. A brood size of 5 was used, with each brood member tested on 20 parasites.

No.	Fitness	Sampling	Hall	Ph	Br	Reps.	Success	Avg	S.D.
(i)	Simple	Random (100)				5	60%	(> 1210)	(473)
(ii)	Shared	Random (100)				5	100%	266	54.7
(iii)	Shared	Shared (100)				5	100%	719	328
(iv)	Shared	Shared (50)	* (50)			5	100%	285	71.2
(v)	Shared	Shared (50)	* (50)	*		5	100%	469	58.3
(vi)	Shared	Shared (50)	* (50)	*	*	5	100%	96.5	11.5

Table III.5: Number of Games to Solve 3DTTT (in Millions)

[3]Substantially smaller populations did not work well, probably due to the large fraction of offspring ruined by the complex genetic operators.

	(i)	(ii)	(iii)	(iv)	(v)	(vi)
(i)	X	*	*	*	*	*
(ii)		X	*	-	*	*
(iii)			X	*	-	*
(iv)				X	-	*
(v)					X	*
(vi)						X

Table III.6: Significant Differences ($p < 0.05$) in Number of Games to Solve 3DTTT

Table III.5 gives results. Some of the runs in (i) did not complete, though they were run for 2 billion games (10000 generations). The runs that did not complete were not included in the average and standard deviation; this is indicated in the table with parentheses.

Since coevolution without the new methods was unable to solve Nim, it is somewhat surprising that it was sometimes able to solve the apparently more complex 3DTTT problem. This is discussed further in Section III.G.7.

Table III.6 shows significant differences in performance. In general, the new methods improve performance substantially. The phantom parasite performed poorly, but this is largely due to a single outlier that required nearly 5000 generations. The difference from the preceding experiment without the phantom parasite is not significant, and it might disappear with more runs. One possible disadvantage of the phantom parasite is that we need to use simple fitness for elitism. Elitism then loses some of its ability to protect important innovations that defeat opponents few other individuals can. Such innovative jumps are especially likely with 3DTTT's complex representation.

Performance with competitive fitness sharing and random sampling was surprisingly good. This suggests that GA parameters may not have been chosen well. Random sampling introduces more noise into fitness than shared sampling, which effectively reduces selection pressure. Selection pressure may have been set too high for use with shared sampling.

Brood selection does very well. The performance improves by a factor of about 2.5 over experiments (ii) and (iv). If population size were the constraining factor in evolution on this problem, we would expect an increase in population size to cause a proportional decrease in the number of generations required. With brood selection, it appears as if the brood size of 5 (coupled with cost of twice as much testing work) effectively increases the population size by 5 for this problem; the full impact of brood selection is being realized. This makes sense given 3DTTT's complex representation and genetic operators.

As a speed comparison, we attempted to solve 3DTTT with a naive conventional depth-first minimax search. Over 7 billion nodes were searched in 1 day of CPU time on a 132 MHz PPC 604. At this point, the search was still exploring variations that begin with the first possible first-player move and the first possible second-player response to this first-player move. A much larger amount of time would have been required to complete the search.

There are many early wins in 3DTTT. We improved the naive minimax search by utilizing this knowledge: we modified search to check for two-in-a-row configurations that result in immediate wins for the moving player. Doing this allowed a rapid solution to the game (under a minute of CPU time). Even though the improved minimax search can quickly obtain a solution to the game, the final coevolved solution to 3DTTT still has the advantage of being very compact, fast, and comprehensible.

III.G Interpretation

III.G.1 Typical Runs

Figure III.2 shows the average number of wins for both populations in 2DTTT during one run, in which sharing and shared sampling were used. In the early part of the run, many flaws in the first population are discovered by the second population and weeded out. In some runs (such as the one displayed), this initial stage is sufficient to find a perfect strategy. In other runs, after this initial stage, the

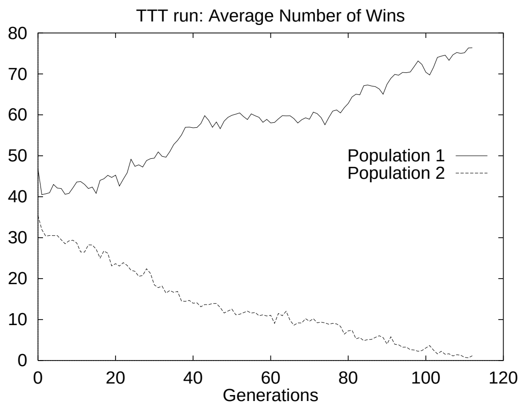


Figure III.2: Average Number of Wins for Both Populations in 2DTTT

best first population individuals typically defeat all opponents ("defeat" here means "do not lose when going either first or second") in most generations. The second population occasionally discovers flaws in the first, which are then quickly corrected by the first population, until a perfect strategy is found.

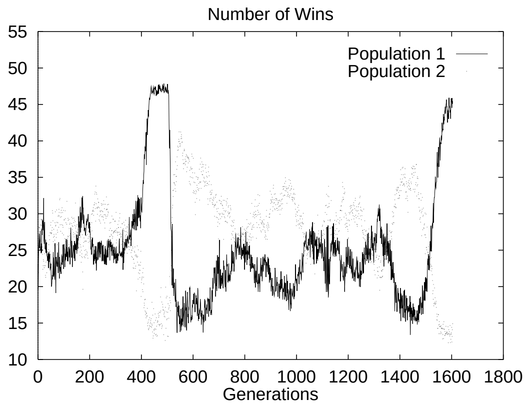


Figure III.3: Number of Wins for Both Populations in Nim

Nim is much more difficult than 2DTTT. This is probably due to several factors. First, there is greater constraint on the moves a perfect player can make. Second, the game tree is somewhat deeper and has substantially more distinct paths through it. Finally, the representation is more epistatic, since unlike 2DTTT, several genes enter into each move decision.

In typical runs on Nim, populations take turns being ahead in the arms race, with fitness measured against current opponents fluctuating for both populations. Figure III.3 shows average number of wins during a typical successful run, using fitness sharing, a shared sample of 25, and a hall of fame sample of 25. The competition seems balanced, with neither population dominating for too long.

In some of the runs with fitness sharing, shared sampling, and the hall of fame but without the phantom parasite, there are periods of about 50 generations where one population has a dominant individual that appears to be perfect until it is finally defeated. Such a period can be seen around generation 450 in Figure III.3. These periods do not occur before the final stages of coevolution in runs that do use the phantom parasite; this small effect is one of the few visible differences when using the phantom parasite for these simple test problems. The phantom parasite is of greater utility in less balanced competitions; this may be seen in Section IV.A.

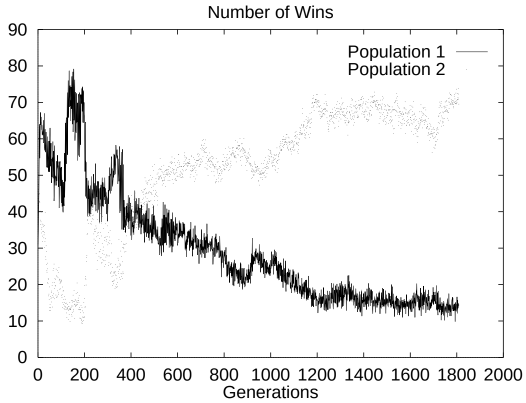


Figure III.4: Number of Wins for Both Populations in 3DTTT

For 3DTTT, there is less fluctuation than in Nim. Figure III.4 shows a typical plot of average fitness for a successful run using fitness sharing, a shared sample of 50, and a hall of fame sample of 50. The first population typically dominates early in the run (there is an advantage in having an extra piece on the board when playing against weak opponents). Later in the run, the second population begins to dominate. At this point, the first population is still managing to cover the second population; very rarely is a particular second-population genotype able to defeat *all* first-population parasites for more than one generation. But, the first population falls behind in ability to defeat a large number of parasites with a single strategy. Larger sets of first-population hosts are required to cover all second-population parasites, and first-population fitness falls.

Fitness trends in faster 3DTTT runs employing brood selection are qualitatively similar, but accelerated.

III.G.2 Testing

An important basic question is whether coevolution was necessary at all. If a large, randomly chosen set of parasites was also sufficient to produce a perfect strategy, then search over examples was unnecessary. Several experiments were done in which a fixed set of test cases was chosen randomly (corresponding to a fixed second population in the experiments above), and strategies were tested against these. No sampling was used; all parasites were used with every individual tested.

For Nim, both with and without fitness sharing, no strategy could be found that defeated all test cases (with a variety of test case set sizes used, from 2000 up to 100000), though runs with fitness sharing came quite close (for example, failing to defeat only 12 cases out of 100000). Population sizes of up to 10000 were used, so it appears that coevolution is really necessary to find perfect strategies for Nim.

Random tests were also used on a smaller version of Nim (3,4,5). Here, strategies were found that could beat all tests, but even when 200000 test cases were used, these strategies were not perfect. This suggests that even if the GA had succeeded in finding strategies that could beat all test cases for large Nim, these strategies would likely be imperfect.

Comparing these results using random testing with those of competitive coevolution brings to mind issues discussed in the introduction. The first is that coevolution is indeed able to find a set of test cases adequate for the evolution of optimal individuals. The results on the small version of Nim indicate that finding such adequate sets can be a difficult problem, requiring much more search than random sampling can be expected to accomplish.

The second point concerns finding *pedagogical* sequences of test cases. For the large version of Nim, random tests apparently provided a rough fitness landscape that was too difficult for the GA to successfully search. There existed some test cases which were a sort of "needle in the haystack"; defeating other test cases did not provide enough information to get close to defeating these difficult cases. This is not as much of a problem with competitive coevolution: it seems that one population never gets so far ahead that the other population cannot catch up (until a perfect individual is found). The difficulty of test cases stays relatively appropriate at each step of evolution.

III.G.3 Extinction

As described in Section III.D.2, fitness sharing and shared sampling should theoretically reduce the extinction of host types that defeat parasites others cannot. Here, we provide empirical support. Each generation, for each population, a count is made of the number of sampled parasites P meeting two conditions:

1. There existed hosts H this generation which defeated P.
2. No such host H was selected to contribute genetic material to the next generation.

This describes the sort of extinction events considered in Section III.D.2. For purposes of comparison, the average number of extinction events per generation is calculated, and the average of this quantity across multiple runs is presented. Extinction events are typically fairly rare during a successful run.

Note that the loss of a single host may be counted as several extinction events (one for each parasite which it beat, but which no other host could beat). For 2DTTT, each parasite can yield two separate extinction events: one when it goes first and one when it goes second.

No.	**Fitness**	**Sampling**	**Hall**	**Ph**	**Br**	**P1**	**P2**
(i)	Simple	Random (40)				0.766%	0.00435%
(ii)	Simple	Shared (40)				0.613%	0.0162%
(iii)	Shared	Random (40)				0.0155%	0.0%
(iv)	Shared	Shared (40)				0.0269%	0.0%
(v)	Shared	Shared (20)	* (20)			0.0177%	0.0%
(vi)	Shared	Shared (20)	* (20)	*		0.0309%	0.0%
(vii)	Shared	Shared (20)	* (20)	*	*	0.0358%	0.0%

Table III.7: Average Extinctions Per Generation for 2DTTT (Expressed as a Percentage of the Total Number of Parasites)

	(i)	(ii)	(iii)	(iv)	(v)	(vi)	(vii)
(i)	X	-,*	*,-	*,-	*,-	*,-	*,-
(ii)		X	*,*	*,*	*,*	*,*	*,*
(iii)			X	-,-	-,-	-,-	-,-
(iv)				X	-,-	-,-	-,-
(v)					X	-,-	-,-
(vi)						X	-,-
(vii)							X

Table III.8: Significant Differences ($p < 0.05$) in Extinctions for 2DTTT (P1,P2)

Table III.7 shows the average extinction events per generation for each of the seven variants, on 2DTTT. P1 refers to extinctions among second-population hosts tested against first-population parasites, and P2 refers to extinctions among first-population hosts tested against second-population parasites. The number of extinctions is presented as a percentage of the total number of parasites used. This figure can range from 0% (no extinction events) to 100% (every parasite had some host which defeated it but was not selected).

Extinctions among second-population hosts occur in all sets of runs. Late in 2DTTT runs, each first-population parasite is defeated by only a small number of second-population hosts. It is easy for selection to generate an extinction event by missing just one of the many hosts that make up such a sparse cover of the first-population parasites. Extinctions among first-population hosts are much more rare, probably because their evolutionary task is easier.

Table III.8 shows significant differences in extinction events. As expected, the use of competitive fitness sharing significantly reduces extinction events. Beyond this, additional new methods do not significantly change extinction.

No.	**Fitness**	**Sampling**	**Hall**	**Ph**	**Br**	**P1**	**P2**
(i)	Simple	None (500)				0.559%	0.384%
(ii)	Simple	Random (100)				0.148%	0.0334%
(iii)	Simple	Shared (100)				0.0476%	0.00781%
(iv)	Shared	None (500)				0.000578%	0.000028%
(v)	Shared	Random (100)				0.00025%	0.00015%
(vi)	Shared	Shared (100)				0.00796%	0.0%
(vii)	Simple	Random (50)	* (50)			0.431%	4.51%
(viii)	Shared	Shared (25)	* (25)			0.00118%	0.0%
(ix)	Shared	Shared (25)	* (25)	*		0.000645%	0.0%
(x)	Shared	Shared (25)	* (25)	*	*	0.00514%	0.0149%

Table III.9: Average Extinctions Per Generation for Nim (Expressed as a Percentage of the Total Number of Parasites)

	(i)	(ii)	(iii)	(iv)	(v)	(vi)	(vii)	(viii)	(ix)	(x)
(i)	X	*,-	*,*	*,*	*,*	*,*	-,*	*,*	*,*	*,*
(ii)		X	-,*	*,*	*,*	*,*	-,*	*,*	*,*	*,*
(iii)			X	*,*	*,*	*,*	*,*	*,*	*,*	*,*
(iv)				X	-,-	-,-	*,*	-,-	-,-	-,*
(v)					X	-,-	*,*	-,-	-,-	-,*
(vi)						X	*,*	-,-	-,-	-,*
(vii)							X	*,*	*,*	*,*
(viii)								X	-,-	-,*
(ix)									X	-,*
(x)										X

Table III.10: Significant Differences ($p < 0.05$) in Extinctions for Nim (P1,P2)

Table III.9 shows average extinction events per generation for Nim, and Table III.10 shows significant differences. For variants which could not find perfect strategies, the first 2500 generations were used. Fitness sharing (and shared sampling, by its similar effects) greatly reduce the number of extinction events. Important genotypes are protected to a greater degree. Other methods do not significantly reduce extinction events. It is interesting to see that the hall of fame does not by itself substantially reduce extinction events. Although the use of the hall of fame allows runs to succeed, it is not by itself sufficient to protect important genotypes. It must provide some other advantage; we return to this question below.

No.	**Fitness**	**Sampling**	**Hall**	**Ph**	**Br**	**P1**	**P2**
(i)	Simple	Random (100)				0.0149%	0.375%
(ii)	Shared	Random (100)				0.0%	0.0%
(iii)	Shared	Shared (100)				0.000192%	0.000024%
(iv)	Shared	Shared (50)	* (50)			0.0%	0.0%
(v)	Shared	Shared (50)	* (50)	*		0.0020%	0.0203%
(vi)	Shared	Shared (50)	* (50)	*	*	0.00161%	0.0149%

Table III.11: Average Extinctions Per Generation for 3DTTT (Expressed as a Percentage of the Total Number of Current Parasites)

	(i)	(ii)	(iii)	(iv)	(v)	(vi)
(i)	X	*,*	*,*	*,*	*,*	*,*
(ii)		X	-,-	-,-	-,*	-,*
(iii)			X	-,-	-,*	-,*
(iv)				X	-,*	-,*
(v)					X	-,*
(vi)						X

Table III.12: Significant Differences ($p < 0.05$) in Extinctions for 3DTTT

Table III.11 shows extinction events for 3DTTT, and Table III.12 shows significant differences. Again, runs with competitive fitness sharing have significantly fewer extinction events, while other methods do not significantly reduce extinction events. The phantom parasite increases extinction events among first-population hosts. This is likely due to an increase in the number of parasites that *can* be defeated by the first population. Without the phantom parasite, the second population becomes dominated by strategies that the first population cannot defeat; no extinction events are counted for such undefeatable parasites.

III.G.4 Useful Diversity

The simple nature of 2DTTT and Nim and our representation of strategies for them allows a close look at the usage of positions in the game. Each time a move is made from a position, during any game in a single generation, a counter for that position is incremented. These counts reveal important positions that often occur as a part of real games. The number of positions with usage above a "significant usage" threshold is measured each generation, for both populations. This threshold is set fairly high (see below), with the goal being to identify positions whose usage has been strongly selected for by the GA, rather than those that happen to be used a few times due simply to a chance mutation. A larger number of significantly used positions should therefore indicate a greater diversity of games played by selected individuals. Such diversity is important to test hosts more fully each generation and provide them with richer fitness information. For purposes of comparison, significant usage is averaged over all generations in a run, and the average of this quantity across multiple runs is presented. Because of the close correspondence between the set of positions and the genes in our representation, this measure is similar to measures of gene usage [15].

Table III.13 shows significant usage for the seven variants, on 2DTTT. The significant usage threshold was set at 320, the number of games that 4 hosts play against all their parasites.

Table III.14 shows significant differences in usage for 2DTTT. The use of fitness sharing (and shared sampling, by its similar effects) yield a significant increase in usage. These methods allow survival of a more diverse set of genotypes, resulting in

No.	Fitness	Sampling	Hall	Ph	Br	P1	P2
(i)	Simple	Random (40)				47.47	36.15
(ii)	Simple	Shared (40)				50.25	39.60
(iii)	Shared	Random (40)				59.24	45.93
(iv)	Shared	Shared (40)				57.07	47.35
(v)	Shared	Shared (20)	* (20)			58.92	45.97
(vi)	Shared	Shared (20)	* (20)	*		61.10	50.13
(vii)	Shared	Shared (20)	* (20)	*	*	61.26	49.16

Table III.13: Significant Usage for 2DTTT (expressed as a percentage of the total number of parasites)

	(i)	(ii)	(iii)	(iv)	(v)	(vi)	(vii)
(i)	X	-,-	*,*	*,*	*,*	*,*	*,*
(ii)		X	*,*	*,*	*,*	*,*	*,*
(iii)			X	*,-	-,-	*,*	*,*
(iv)				X	*,-	*,*	*,-
(v)					X	*,*	*,-
(vi)						X	-,-
(vii)							X

Table III.14: Significant Differences ($p < 0.05$) in Significant Usage for 2DTTT (P1,P2)

greater diversity in usage during games played. By encouraging sampling of a variety of pedagogical parasites, the phantom parasite may also increase significant usage to a small extent.

Table III.15 shows significant usage for the ten variants, on Nim. The significant usage threshold was set at the number of games that 4 hosts play against all their parasites. This is 2000 for the runs without sampling, 200 for the runs using sharing, shared sampling, and the hall of fame (since these used a total of only 50 parasites), and 400 for the remaining runs. For variants which could not find perfect strategies, the first 2500 generations were used.

Table III.16 shows significant differences in usage. The use of fitness sharing

No.	Fitness	Sampling	Hall	Ph	Br	P1	P2
(i)	Simple	None (500)				44.29	46.92
(ii)	Simple	Random (100)				58.33	64.43
(iii)	Simple	Shared (100)				91.75	85.86
(iv)	Shared	None (500)				90.42	94.45
(v)	Shared	Random (100)				91.86	97.18
(vi)	Shared	Shared (100)				97.70	97.15
(vii)	Simple	Random (50)	* (50)			48.22	70.75
(viii)	Shared	Shared (25)	* (25)			102.5	104.6
(ix)	Shared	Shared (25)	* (25)	*		101.3	106.2
(x)	Shared	Shared (25)	* (25)	*	*	82.53	92.58

Table III.15: Significant Usage for Nim

(and shared sampling, by its similar effects) yield a significant increase in usage. These methods allow survival of a more diverse set of genotypes, resulting in greater diversity in usage during games played. Brood sampling seems to slightly decrease significant usage. By eliminating poor offspring, it can make populations more homogeneous in some cases. As with extinction events, the hall of fame by itself does not seem to substantially increase significant usage. Such diversity does not explain the advantage provided by the hall of fame.

During a run, significant usage displays revealing trends. Figure III.5 shows average wins and significant usage for the first population, during a run in which fitness sharing and shared sampling of 100 parasites were used. The second population's average wins is a near-mirror image (though not quite, due to sampling), so it is not shown.

Significant usage displays an initial rise, followed by fluctuations which are due in part to temporary convergence and failure in the population. When the population begins to converge to a temporarily successful type (fitness spikes at 670 and 1260 generations), diversity loss can be seen in the drop in significant usage. During periods of relative failure (for example at 450 generations), few types are strongly selected for, and significant usage also drops.

	(i)	(ii)	(iii)	(iv)	(v)	(vi)	(vii)	(viii)	(ix)	(x)
(i)	X	*,-	*,*	*,*	*,*	*,*	-,*	*,*	*,*	*,*
(ii)		X	*,*	*,*	*,*	*,*	-,-	*,*	*,*	*,*
(iii)			X	-,*	-,*	*,*	*,*	*,*	*,*	*,-
(iv)				X	-,-	*,*	*,*	*,*	*,*	*,-
(v)					X	*,-	*,*	*,*	*,*	*,-
(vi)						X	*,*	*,*	*,*	*,*
(vii)							X	*,*	*,*	*,*
(viii)								X	-,-	*,*
(ix)									X	*,*
(x)										X

Table III.16: Significant Differences ($p < 0.05$) in Significant Usage for Nim (P1,P2)

III.G.5 Arms Race Progress

Domain Dependent Measure

In an ideal arms race, individuals in both populations become strictly stronger, continuously. In general, we will not have an efficiently computable, absolute measure of strength to use in checking this condition. The simplicity of 2DTTT and Nim, however, allows an attempt to find such a measure.

One way to measure progress is what we will call the “losable percentage”: the percentage of reachable positions from which a loss may be forced through some line of play, for the best member of the first population in each generation. Roughly speaking, each such losable node is an opportunity that the second population may exploit by leading the first-population strategy to that node, then forcing a win. As these opportunities are exploited and the first population corrects these defects, the losable percentage should decrease, reflecting the arrival of strategies with fewer flaws. A perfect strategy would have a losable percentage of zero.

Figure III.6 shows losable percentage during a 2DTTT run in which competitive fitness sharing and random sampling were used. In this game, losable percentage seems to provide a fairly clear measure of progress. There is steady progress for most of the run as first-population flaws are weeded out. Near the end, the second popu-

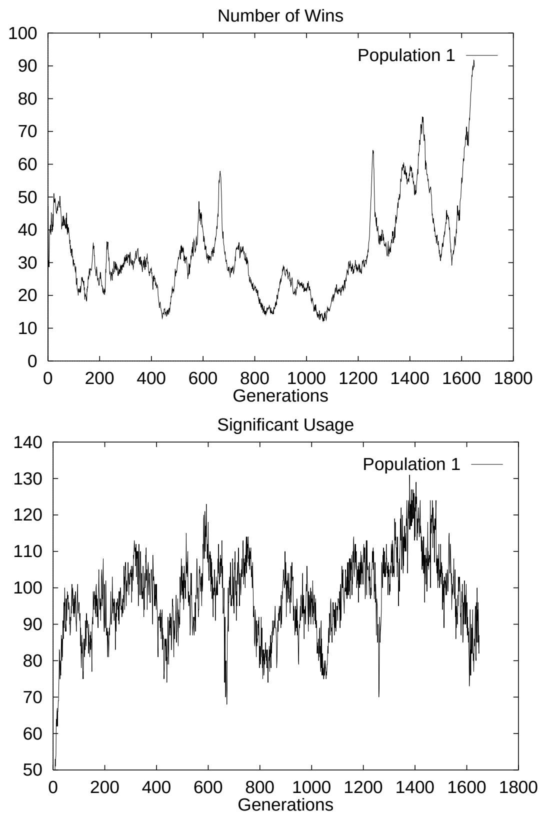


Figure III.5: Number of Wins and Significant Usage for Nim

lation is unable to defeat the best first population strategy for a substantial length of time, until a discovery by the second population finally pushes the evolution of a perfect strategy in the first population.

Figure III.7a shows the losable percentage during a successful Nim run with fitness sharing, shared sampling of 25 parasites, and a hall of fame sample of 25 parasites. As in most runs, there is an initial decrease as gross flaws are weeded out. In this run, there is a noisy decrease afterwards to perfection.

In Nim, progress is not always clear from these graphs. Figures III.7b shows the losable percentage during a successful Nim run with simple fitness, random sampling of 50 parasites, and a hall of fame sample of 50 parasites. Here, the losable percentage fluctuates with little trend. There are periods where the first population is dominating and a single top strategy remains the best for several generations.

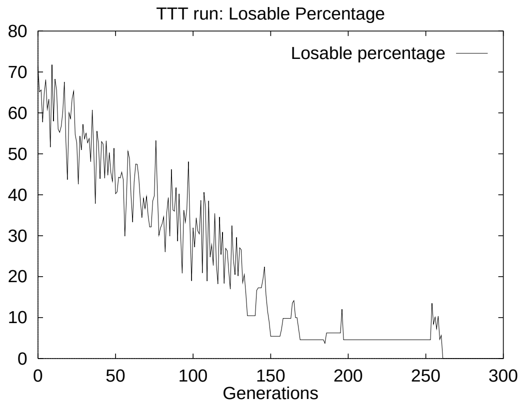
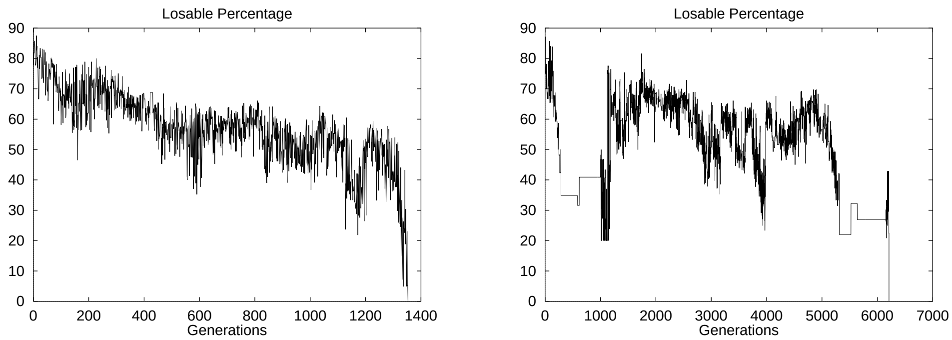


Figure III.6: Losable Percentage for 2DTTT



(a) Shar. Fit./Shar. Samp./Hall **(b)** Simp. Fit./Rand. Samp./Hall

Figure III.7: Losable Percentage for Nim

Eventually, a perfect strategy is found.

Either coevolutionary progress is not present for large portions of the run shown in Figure III.7b, or losable percentage is an incomplete measure of progress in the arms race. In either case, it would be preferable to have a domain-independent view of the arms race. This will be discussed in the following section, and this particular run will be reconsidered.

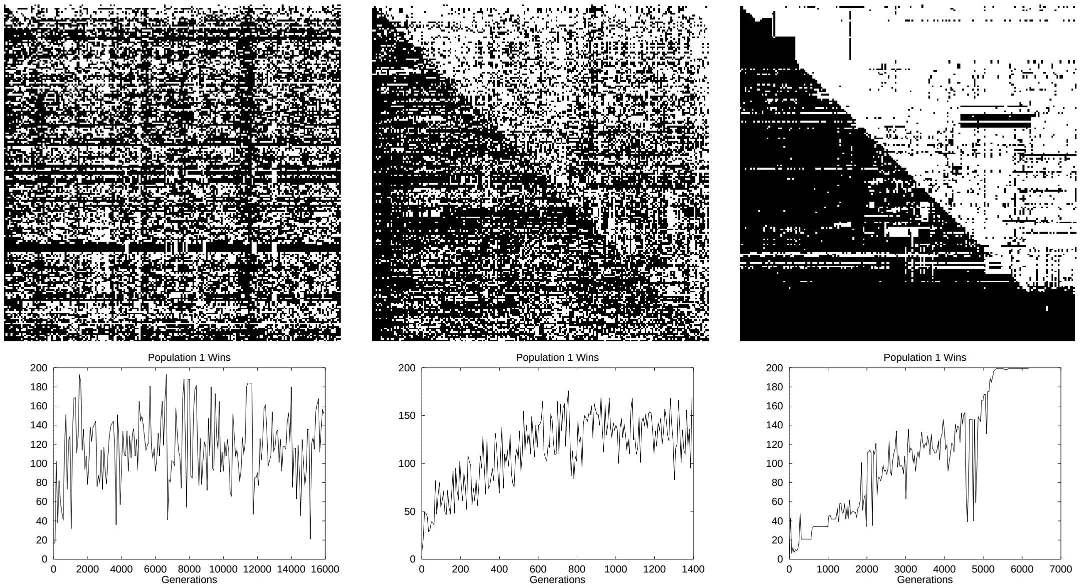
Domain Independent View

A different notion of progress is that newly generated hosts should defeat parasites that prior hosts were able to defeat, as well as new parasites. This sort of progress is essential to the efficiency of theoretical methods discussed in Chapter II. Coevolutionary cycles arise when newly generated hosts are specialized to defeat new parasites, but cannot defeat old parasites that prior hosts were able to defeat.

We may attempt to measure progress according to this idea by examining new individuals and testing them against new and old opponents. The best individual from each generation, according to number of wins, is saved (as in the hall of fame). At the end of the run, a full round-robin tournament is played among these individuals. The outcome of this tournament is displayed on a grid, with each row corresponding to a first-population individual and each column corresponding to a second-population individual. Generations go down and left to right, starting from the first generation in the upper left-hand corner. A black dot indicates that the particular first-population strategy won the game between the row and column individuals. For display purposes, we usually sample generations at a fixed interval to obtain a 200 by 200 bitmap. Similar diagrams have been used to show the outcome of pursuer-evader coevolution [23].

In the ideal arms race, each individual would defeat the opponents from all prior generations. This would correspond to a black triangle in the lower left, and a white triangle in the upper right. Short of perfection, we would like to see long-term progress in the form of a dark left edge and a light upper edge. Since we are only taking a single individual from each generation, the diagram may appear noisy if a larger *set* of individuals was required to cover all opposition in most generations. In this case, the particular best individual chosen will be inadequate to cover all previous generations by itself. This effect sometimes shows up as a solid block of black in the white triangle (or vice versa) when it extends over several generations.

To help visualize progress in the arms race, we add up the number of first-population victories (black squares) in each generation (row). The graph of this quantity should display a linear rise to show coevolutionary progress, as first-population hosts are able to defeat second-population parasites from a larger number of generations. For the 3DTTT runs, we add up second-population victories instead, since the game is a second player win.



(a) Simp/Rand/No Hall 16000 gen **(b)** Shar/Shar/No Hall 1514 gen **(c)** Simp/Rand/Hall 6212 gen

Figure III.8: Arms Race for Nim

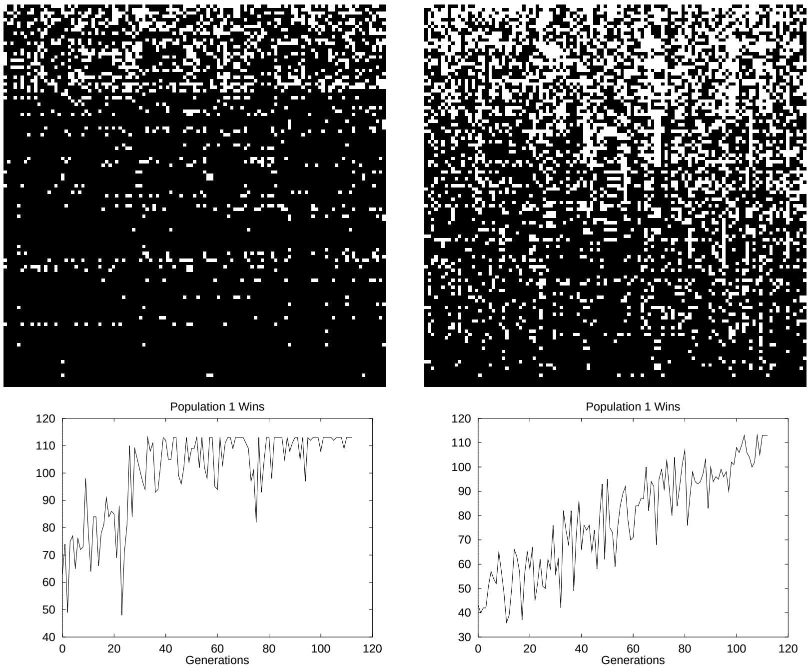
Figure III.8 shows the arms race diagrams for several combinations of the new methods, on Nim. Figure III.8a shows the breakdown of the arms race for a run in which none of the methods were used. This plot shows 16000 generations; the game was never solved during this run. Initial generations looked promising (though this is somewhat difficult to see in this figure), but by 16000 generations there is no long-term progress visible. The prominent relevant feature is the diagonal: newly generated types are specialized to defeat current opposition. The graph of first-population victories shows no long-term trend. Other runs that fail to solve the game are qualitatively similar.

Figure III.8b shows the outcome of a run with fitness sharing and shared sampling, in which the game was solved after 1514 generations. Here, the two triangles may be clearly seen, showing strong progress. The graph of first-population victories shows a noisy rise. This is typical in successful runs that don't use the hall of fame.

Figure III.8c shows a run with the hall of fame, but with simple fitness and random sampling. This run completed after 6212 generations. The two triangles are sharply defined. The hall of fame makes a big difference in the arms race, since it causes direct selection for hosts that defeat the best parasites from prior generations. The improvement in arms race progress appears to be the source of the performance advantage provided by the hall of fame. Note, however, that the arms race is still not ideal as defined above; search cannot be expected to perfectly meet the demands imposed by the hall of fame.

The run in Figure III.8c is the same as the run for which losable percentage is shown in Figure III.7b. For long periods during this run, the arms race diagram shows clear progress that cannot be seen in losable percentage. As each population exposes flaws in opposing strategies, these flaws are weeded out. Due to the hall of fame, populations remember how to exploit these flaws so that they have a difficult time reappearing. This progress is difficult to see with losable percentage. Losable percentage is a particular domain-dependent global measure of solution quality, but it is inadequate for revealing all coevolutionary progress.

Figure III.9 shows the arms race for a 2DTTT run in which competitive fitness sharing and shared sampling were used. The figure shows all 113 generations of the run. Two diagrams are shown; one with the first population playing first, and one with it playing second. The greater difficulty of playing second can be seen in Figure III.9b. While progress in the first population can be seen in the graphs, the related graphs for the second population show little upward trend. The second population serves to expose flaws in the first population, but individuals in the second population do become broadly stronger. During evolution, each member of the second population can defeat only a small fraction of the first population; to see second-population progress, we would need to see the increasing coverage of potential



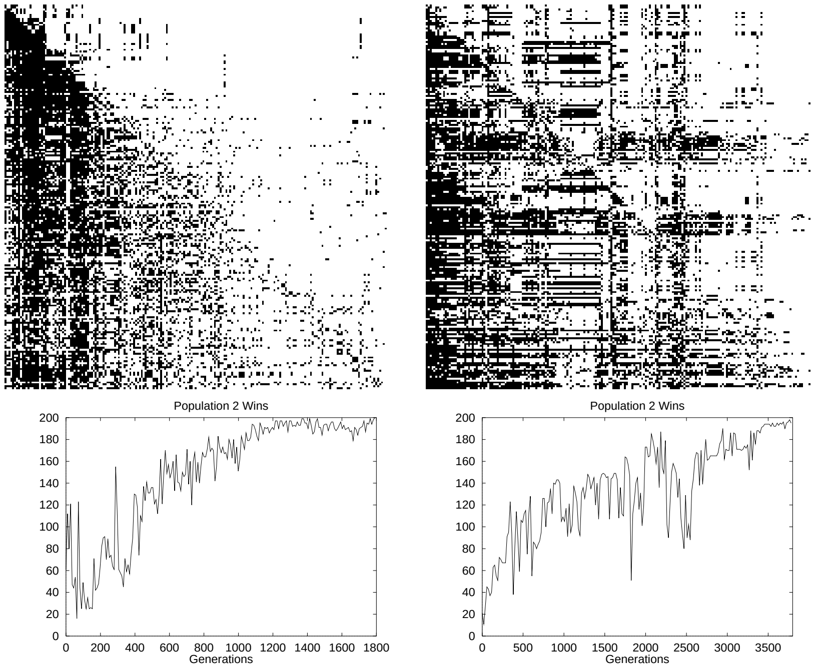
(a) Pop 1 First **(b)** Pop 1 Second

Figure III.9: Arms Race for 2DTTT

first-population flaws by the *entire* second population, taken together.

Figure III.10a shows the arms race in a successful run on 3DTTT using fitness sharing, a shared sample of 50 parasites, and a sample of 50 parasites from the hall of fame. This run solved the game after 1800 generations. During the early part of the run, the arms race is fairly balanced. The first population is eventually able to defeat most early second population opponents, and largely retains this ability. Later in the run, single first-population hosts are not adequate to defeat all prior parasites. First-population strategies do exist that defeat the second-player strategies from later generations, but larger sets of first-population strategies are required to cover all of these opponents. As a result, the representative first-player strategy chosen for the arms race diagram defeats sparser subsets of these opponents. This feature contributes to the clear rise in second-player victories during the run.

Figure III.10b shows the arms race in a successful run on 3DTTT using simple fitness, no hall of fame, and a random sample of 100 parasites. This run



(a) Shar/Shar/Hall
1800 gen

(b) Simp/Rand/No Hall
3875 gen

Figure III.10: Arms Race for 3DTTT

solved the game in 3875 generations. There appear to be longer stretches where later strategies are failing to beat earlier opponents; this is to be expected since the hall of fame is not used. There is still an increasing trend in second-player victories during the run. This feature stands in contrast to Nim, where no long-term progress in the arms race was visible when the new methods were not used. This difference is discussed in Section III.G.7.

III.G.6 Timing and Drift

For pedagogical purposes, the competition between populations should be fairly balanced at all points during coevolution. This way, fitness is most informative and neither side is in danger of losing track of the opposition. Competitive fitness sharing and shared sampling prevent the loss of important genotypes and increase useful diversity. This may help prevent timing problems, since it encourages selection of opposition for all parasites. This way, it is difficult for a parasite to become dominant, with no hosts able to defeat it.

When one side in competitive coevolution gets very far ahead, both sides lose fitness information because the outcome of most competitions is the same. For the losing side, this implies that there will be little selection in the direction it needs to go. For the winning side, this may allow genetic drift[4] if mutation can overcome the now-minimal selection pressure. In cases of extreme unbalance, the phantom parasite allows selection of some of the mutated individuals, as discussed in Section III.C.4.

The effect of these drifting mutants in competitive coevolution is a potentially beneficial one. Drift will provide degraded, pedagogical versions of the particular parasites that the losing hosts are finding so difficult to defeat. Drift will continue to the point where these degraded parasites may be defeated; this begins to give the losing population some direction. Ideally, the drifting parasites could provide a smooth hill in difficulty, up to the current undefeatable parasite type.

It is difficult to view this effect during a run, because it cannot be easily separated from everything else that is happening. In an attempt to see whether this effect could be significant, a separate experiment was done. The first population was seeded entirely with copies of a first player strategy (for the game of Nim) that was very good but not quite perfect. This difficult first player was chosen from a point near the end of a successful competitive coevolution run. The second population was seeded randomly, and competed against the first (fitness testing done with a full round-robin tournament, 500 individuals per population, competitive fitness sharing, no phantom parasite).

As a first test, if the first population is not allowed to drift, there is no fitness variation to guide the second population (they all lose every game), and the search fails (40 million randomly generated 2nd player strategies were all unable to defeat the seed player). So, next, drift is allowed, by using the usual Nim GA parameters.

[4] We use the term "drift" to refer to fluctuations due to random forces (as opposed to selection). The term is often used when discussing random convergence of a gene to a particular allele. With our fairly high mutation rates, such convergence does not seem to be an important issue.

Elitism is such that the seed individual is not displaced until it is defeated.

To track the second population, a comparison was made between the moves that must be made to defeat the seed individual, and the moves that were actually made in games won by the second population against drifting opponents. Figure III.11 was obtained as follows. Every game won by a second player was checked to see how many of the moves made (1st and 2nd player) during that game also arise in one of the games by which the seed player can be defeated. Such moves are rare, because there are only a handful of games by which the seed player can be defeated; these games will be called "winning paths." The graph shows this quantity as a fraction of total moves played; if this fraction becomes 1, the participating 2nd player is able to defeat the seed player (this happens at generation 49 in this run).

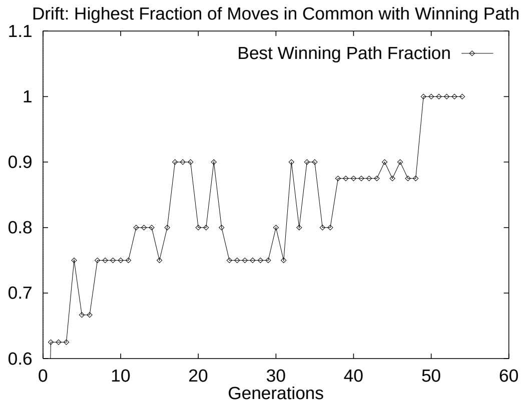


Figure III.11: Drift Provides a Hill to Climb

A closer look at the games played by the second-population individuals shown in the graph shows how this is working. Since the seed player is not perfect, there is a particular position from which the seed player makes a mistake. Very few games' paths lead it to this mistake, however, and perfect play is required to capitalize on it after it is made. Drifting versions of the seed player relax both of these conditions. Over the drifting population, there exist more paths to reach the mistake position. Also, exactly perfect play is not necessarily required to capitalize on this mistake against degraded first-population individuals. This allows the second population to find this mistake and make use of it, then improve the play both before and after reaching the mistake until the seed player can be defeated.

Drift in the first population creates degraded individuals which provide sufficient information to guide second population towards defeating difficult first-population opponents. This is a form of self-correction to timing that may occur during coevolution. It resembles the way in which the Baldwin effect can exploit individuals' plasticity (eg. to learn during their lifetimes) to convert difficult evolutionary problems into tractable ones [16].

Drift is likely to be an important factor in tuning parameters. The phantom parasite explicitly causes selection of drifting, pedagogical individuals in cases of extreme unbalance. Mutation rates must be high enough to permit significant random variation in a successful population, and sample size must be large enough to include degraded individuals. Brood selection may prevent degraded individuals from entering the population each generation, lessening the effect of drift. Other factors may enter into the control of drift as well. For example, in an early set of experiments on 2DTTT, random sampling of opponents was found to require *fewer generations* to solve the game than full round-robin testing [80]. This is surprising: random sampling may save work overall, but it yields noisier fitness information that should lead to a larger number of generations required. It appears now that full round-robin testing was less effective because it did not permit drift to as great an extent.

III.G.7 Search Difficulty and Coevolutionary Difficulty

Of the problems studied, 3DTTT seems most difficult in some ways. The game, representation, and genetic operators are the most complex. For consistent success, a larger population and larger sample were required. It is therefore surprising (see Section III.F) that simple coevolution, without any new methods, was able to solve 3DTTT some of the time, since it was completely unable to solve Nim.

We believe that the complex representation and operators used in 3DTTT make search difficult. Operators will often be very destructive, and improved individuals are fairly rare. But these factors would make evolution difficult even in a non-coevolutionary setting, against a fixed set of opponents. In general, 3DTTT is probably a difficult *search* problem. This conclusion is consistent with the gains from brood selection for this problem: brood selection aids the difficult search in each population without directly improving coevolution.

This stands in contrast to Nim, with its simple, crossover-friendly representation. Operators will rarely be very destructive. Two individuals that play well in different parts of the game graph will often be easy to combine via crossover into a child that plays well in the union of these parts. These suggest that Nim should be an easier *search* problem. This is consistent with the lack of gains from brood selection; in an easy search problem, there is no need to generate a large number of individuals and filter them with a limited fitness test.

So, there must be other sources of difficulty in Nim that cause coevolution without the new methods to fail. For example, the coevolution-specific problems of arms race progress and pedagogy may be particularly severe in Nim. The paths from initial random strategies to perfect strategies might be longer and more difficult to traverse (in a pedagogical sense) than in 3DTTT, making Nim a more difficult *coevolutionary* problem.

One possible way to get a rough separation of search difficulty from coevolutionary difficulty in an adversarial problem is to experiment with changes in population size and the use of brood selection. If large populations and brood selection allow very short successful coevolutionary runs, as is the case in 3DTTT, then lengthy arms races are not required to solve the problem. It may be easy for coevolution even though if it is a difficult search problem. If, on the other hard, a large number of generations are required even with brood selection (this is true to some extent in Nim), then the problem is consistently difficult from a coevolutionary standpoint.

III.H Guide to Using Coevolution

We have identified several factors important in competitive coevolution, and have examined the performance of several new methods in light of these factors. Competitive fitness sharing, shared sampling, the hall of fame, the phantom parasite, and brood selection significantly improve coevolutionary performance. Solutions are particularly efficient when these methods are used in conjunction, with methods having varying importance for different adversarial problems.

Competitive fitness sharing modifies short-term dynamics by selecting hosts that defeat parasites few other hosts defeat. Selection of individuals on this basis creates a diverse population that covers the opposition. Important genotypes that defeat strong parasites are unlikely to be lost. This is important for both testing (when the population serves as parasites) and for future evolution. The use of competitive fitness sharing preserves the coevolutionary equilibria that exist under traditional simple fitness.

Shared sampling selects diverse sets of parasites that test all segments of a host population. In addition, the performance-based nature of shared sampling gives it some of the effects of competitive fitness sharing: important parasites are sampled more frequently, so that hosts able to defeat them are strongly rewarded. This allows successful coevolution with shared sampling, even when fitness sharing is not used.

The hall of fame preserves individuals from previous generations for later use as parasites. This helps enforce progress in an arms race: successful newly evolved hosts must defeat old parasites as well as current ones. This is important to convergence on optimal solutions. The use of the hall of fame is found to serve this purpose well in practice; it plays a role in coevolution that is distinct from the roles played by shared sampling and fitness sharing.

Timing and pedagogy are important in coevolution. If one side gets far ahead of the other, both sides lose informative fitness and evolution slows. The diversity provided by fitness sharing can help prevent such timing problems, because each population maintains a cover of the opposing population. Additionally, drift may play an important role in correcting timing problems. When a population is far ahead in the arms race, it wins almost all competitions and has little fitness information to guide it, so that it begins to drift. Drifting individuals are less difficult to defeat and can provide a hill for opponents to climb towards defeating individuals that are currently out of reach. The phantom parasite permits explicit selection of drifting, pedagogical individuals when balance problems are extreme; this is discussed further in the context of a naturally asymmetric problem in Section IV.A.

Since we have introduced a fairly large number of methods for competitive coevolution, it would be helpful to give some guidance for applying these methods to new applications. Since coevolution depends heavily on underlying GA parameters, we should also give some guidance for setting these in the context of coevolution.

The experiments presented above provide an easy rough answer to the question of which methods to use: results are best (or insignificantly different from the best) when all methods are used together. When exploring a new application, this is a good first step. The most important caveat relates to brood selection: if the adversarial problem has an easy underlying search, then brood selection may not be necessary. Since brood selection may have negative side-effects on coevolution (for example, by removing pedagogical individuals), it should be used carefully.

Population sizes need to be fairly large to maintain the diverse niches needed during coevolution. This is problem-dependent, but it is reasonable to start with a very large population since population sizes larger than the minimum required may yield coevolution speedups anyways.

GA operators should be tuned to place emphasis on exploration, for example by using high mutation rates. This helps encourage diversity, and the higher fraction of degraded individuals produced may actually be helpful in terms of providing pedagogical parasites for the opposition.

Parasite samples need to be fairly large to encourage progress and provide guiding fitness information for both sides. In this chapter, sample sizes are roughly

10% of the population size; this seems to work well. If this level of testing is computationally excessive, brood selection may help by culling poor offspring before they are tested extensively.

The arms race diagrams discussed in Section III.G.5 are an important tool for tracking coevolutionary progress. As long as the arms race appears strong, it is sensible to continue a coevolutionary run. Once the arms race breaks down (yielding, for example, an arms race diagram such as that seen in Figure III.8a), the run will probably not yield a solution to the adversarial problem.

The following chapter discusses several applications. To explore the success of a simple recipe for using competitive coevolution, we use all methods discussed in this chapter in each case (with the exception of brood selection in Section IV.A). If this recipe is generally successful, then competitive coevolution may not be much more difficult to tune and use than the GA by itself.

It should be noted that parallel implementations of competitive coevolution are almost trivial. Each processor evaluates a partition of a host population against the entire parasite sample, and broadcasts the competition results. For applications of any significant complexity, computation time for the evaluation completely dominates communications overhead, so that the full benefits of parallelization are realized. Some of the experiments presented in Chapter IV were performed on a parallel Cray T3D.

Chapter IV

Applications

In this chapter, we use competitive coevolution in several applications. The adversarial problems that arise in these applications are diverse, coming from distinct classes of practical adversarial problems discussed in Section I.D. Unlike the test problems in Chapter III, there are no straightforward, simple solutions for these problems. They are more realistic tests of coevolution. The same basic coevolutionary setup discussed in Section III.H is used for each.

Section IV.A discusses the design of controllers that are robust across a wide range of possible plants. Simulation is used to estimate the real-world performance of a particular controller on a particular plant. The specific application is the design of blood-pressure controllers that can successfully control a variety of patients. This problem is distinguished by real-valued parameters that describe individuals, and complex nonlinear constraints on feasible individuals. In addition, the competition is very asymmetric, with the search for controllers being much easier than the search for plants. Genetic operators and coevolution parameters must be chosen carefully to control the balance of coevolution in the face of this asymmetry.

Section IV.B discusses the design of sorting networks, which are sorting algorithms of a specialized form suitable for hardware implementation. We seek networks that correctly sort all relevant test sequences. Existing theoretical work allows us to bound the specification number for various instances of the adversarial problem. This knowledge allows us to explore the practical importance of specification number, and the tradeoffs between successful search and ease of testing.

Section IV.C explores the coevolution of strategies for the game of Go. Go is a difficult game that remains challenging for computers. We coevolve cellular automata-based strategies of variable complexity. This provides an essentially open-ended competitive environment. Experiments consider the extent of arms races in this domain, the evolution of useful complexity, and the competence of evolved strategies.

Section IV.D describes an application to drug design. Drug resistance is a major obstacle to the effectiveness of many antiviral drugs. We treat drug designs as solutions, and resistance mutations as test cases. We use a computational model of drug interaction to search for drugs that are robust across possible resistance mutations. This section presents preliminary work that uses a simplified model of inhibitor binding to HIV protease. The simplicity of this model permits the use of theoretically complete methods from Chapter II to exactly solve the adversarial problem.

IV.A Coevolution of Controllers and Plants

IV.A.1 The Problem

One class of adversarial problems uses a simulation of real-world interaction to evaluate competition outcome (see Section I.D). An example of this class is the design of controllers that are robust across the range of possible plants they might be required to control. Simulation is used to evaluate the behavior of particular controllers on particular plants. We consider the design of drug-delivery controllers for the control of patient blood pressure during surgery. After presenting prior work on an appropriate model and its use in controller design [86, 87], we use coevolution to study the existence of perfect controllers under the model.

A model of blood pressure response to sodium nitroprusside infusion has been developed based on experimental data [87]. It depends on several parameters describing a specific patient, and on an externally supplied level of drug infusion each timestep. The model has been used in the design of CMAC controllers that map blood pressure to drug infusion level in real-time during the course of simulated surgery [86]. Given a patient and a candidate CMAC controller, the model can evaluate the performance of the controller on the patient. This evaluation forms the basis for an adversarial problem in which we seek a controller that successfully controls all possible patients. Below, we discuss prior work on this model and its use in controller design (see [86] for full details). A coevolutionary experiment that closely adheres to this prior work is then described.

The form of the model was chosen to allow adequate flexibility in simulating the behavior of real patients [87]. Patients are specified by the parameters in an equation that describes the dynamics of a discrete-time simulation:

$$X[k+1] = \alpha X[k] + \beta u[k-d] + D_0 + D_1 k + D_2 k^2 + D_3 k^3$$

where $X[k]$ is the mean arterial pressure at timestep k and $u[k]$ is the drug infusion rate at timestep k. d is the delay between drug infusion and drug effect. Each timestep corresponds to two seconds of real time. The coefficients and d describe the patient. All parameters are continuous, except for the integer d.

Development of controllers for actual use in surgery would require guarantees of robustness across a wide range of surgical scenarios.[1] As a first step, experiments described here focus on one particular scenario [86]. The initial condition is $X[0] =$ 100 mmHg, and the simulation is iterated for 700 timesteps. During a simulation, the patient is said to "die" if mean arterial pressure ever drops below 50 mmHg, or if it rises above 120 mmHg after timestep 200. The goal is to keep patient blood pressure near a target level of 70 mmHg.

Constraints on parameters are important so that candidate controllers are not unfairly penalized for failing on unrealistic patients. Simple range constraints are placed on the parameters α, β, D_0, and d. In addition, the steady-state blood pres-

[1] Anthony Sebald, personal communication

sure, $\frac{D_0}{1-\alpha}$ is constrained to lie between 80 and 120 mmHg. Finally, the uncontrolled ($u = 0$) blood pressure must lie between 80 and 120 mmHg for all 700 timesteps. This final criterion expresses a complex nonlinear constraint that must be enforced via simulation, and is a significant source of difficulty for this problem.

The drug infusion rate u is determined by a CMAC controller that has current blood pressure and rate of blood pressure change as inputs. A CMAC controller [4] divides its input state space with a grid, and has a separate weight for each cell in the grid.[2] A particular input activates N cells that are close to it in state space; N is a parameter that determines the extent of generalization. The weights from these cells are summed to obtain an output that is interpreted as current drug infusion rate. Weights are adjusted so that control is increased or decreased depending on whether blood pressure is above or below the target value of 70 mmHg. This basic weight adjustment algorithm is fixed so that at least the sign of the weight changes is always reasonable. Weight adjustment size is proportional to a learning rate that is increased periodically during the simulation if blood pressure is getting further from the target.

The five controller parameters specify the number of selected cells (N), the assumed delay in the patient, the default weight change size, the initial learning rate, and the number of time steps between learning rate changes.[3] Controller parameters have simple range constraints. One important feature of these constraints is that controllers cannot continually apply zero control to a patient that is not at the target blood pressure; all controllers are constrained to apply some positive control. As a result, controllers cannot keep patients alive simply by doing nothing.

[2] In actual implementations, some cells share weights due to a hashing scheme used for memory efficiency.

[3] Hash table size was also included as a parameter by Sebald and Schlenzig. In our simulations, hash table size is simply set to a large value to avoid collisions that are necessarily random and destructive. Since we know in advance that we wish to use a large hash table size, there is no reason to include this size as a parameter for these experiments.

IV.A.2 Minimax Evolution

The blood pressure controller design problem was originally formulated as a minimax optimization problem[86]. Total squared error from the target blood pressure of 70 mmHg is summed over all timesteps of the simulation to score a particular controller on a particular patient. If the patient “dies,” as described above, a large penalty is assigned to the controller. This reflects the fact that patient death is worse than any error for a patient that lives. The minimax goal is to find the controller for which the worst-case patient (specific to that controller) gives the smallest error.

Sebald and Schlenzig use a minimax form of evolutionary programming to approach this problem [86]. An evolving population searches for controllers. Each controller in this population is evaluated by a separate complete evolutionary search for patients that are difficult for that controller. This inner loop searches for the patient that maximizes the error assigned to the controller. The highest error found during this inner loop is used as the fitness of the controller for the outer loop search. The outer loop should then search for the controller with minimal worst-case error.

The minimax evolution experiment used a population size of 50 for both inner and outer loops, running the inner loop for 20 generations and the outer loop for 50 generations. Each generation, selection keeps the best half of the population as elitists and for use as parents. Parents produce offspring without crossover. Mutation adds a gaussian deviate to each parameter, with fixed variance depending on the range of the parameter.

This experiment produces controllers that appear to be successful: the most difficult patients found by the inner loop only give small error. But a more extensive search for patients, evaluated for error against the best controller found during the run, yields patients that die under the care of this best controller. So, the best controller found is unsuccessful by the minimax criterion. Of course, we don't know whether successful controllers exist in this representation.

Minimax evolution bears some similarity to competitive coevolution. Minimax evolution should be successful in the limit of large populations and long searches.

When the search done by the inner loop is limited, however, the outer loop only needs to produce mediocre solutions that are successful in fooling this limited (and possibly noisy) evaluation. Each inner-loop evaluation starts with a newly initialized population of test cases, so there is no arms race to produce continually better test cases. Furthermore, the lack of sharing of results between inner-loop searches may be wasteful.

IV.A.3 Coevolution

Minimax evolution is intended to produce controllers with small worst-case error. But the controllers found allow some patients to die and therefore incur the maximum penalty. Before attempting to minimize worst-case squared error, it is natural to ask whether there exists any controller that is at least successful in keeping all patients alive. Death is a binary outcome of competition between controller and patient. Since the methods in Chapter III are designed for adversarial problems with binary competition outcomes, they may be applied directly to this problem.

Experimental Setup

Most of the standard coevolution setup described in Section III.H is used. This includes competitive fitness sharing, shared sampling, the hall of fame, and the phantom parasite. The GA used within this framework is problem-specific. The controller and patient search spaces are very different, so separate GA operators and parameters are used for the two populations.

Preliminary runs revealed several difficulties that are relevant to the design of a successful GA for this problem. First, good (i.e., difficult) patients are difficult to discover without carefully designed operators. The best patients tend to lie at the edges and corners of the nonlinear constraint region. Figure IV.1 shows a cut through the constraint region. Feasible patients lie in the shaded area. One difficult patient found by coevolution lies at the lower right corner of this area. Such patients are hard to find, because the elongated area is difficult to navigate with standard genetic

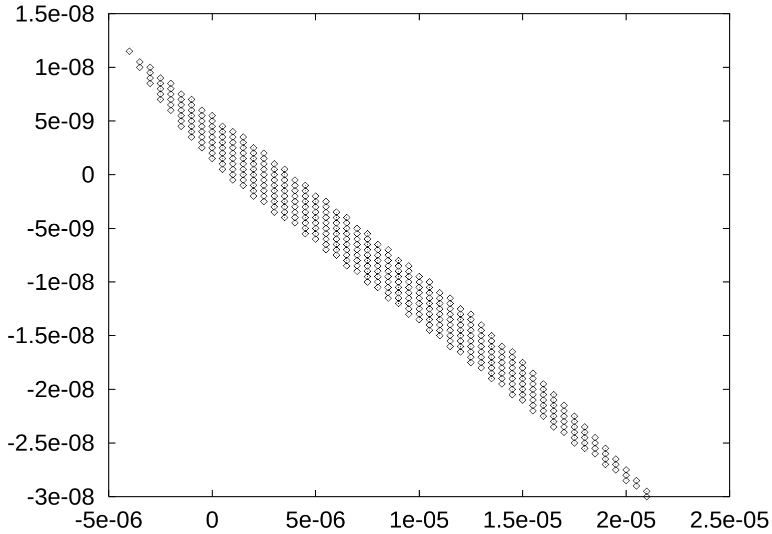


Figure IV.1: Feasible Patients Along D_2 and D_3 with Other Parameters Fixed.

operators. Using simple crossover and mutation often yields offspring that are outside of this region. Simply rejecting such offspring results in premature convergence, and a lack of exploration near the boundaries of the constraint region where the best patients are found.

To allow a continuous exploration of the space of patients, "α-box" crossover is used [92]. Figure IV.2 shows this form of crossover in two dimensions. The two patients A and B being crossed define the corners of box 1. If offspring points are always chosen within this box, the population rapidly converges. So, each side of the box is expanded symmetrically (by a factor of $\alpha = 0.5$, here) to produce box 2, with a dashed outline in Figure IV.2. The offspring is chosen to lie at a random point in this expanded box.

To aid search in finding edges of the constraint region, patients that are generated outside the constraint region via crossover and mutation are "repaired" [43]. A binary search is done along the line between the new point (outside the constraint region) and the old point (inside the constraint region). An arbitrary parent is chosen as the old point when repairing the results of crossover. The binary search is used to find a repaired point inside the constraint boundary, as close as

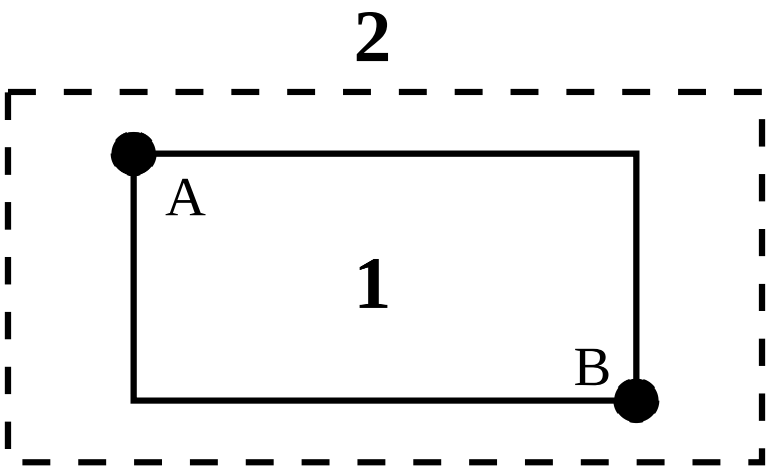


Figure IV.2: α-Box Crossover

possible to the new point. This binary search terminates in a small number of steps (15), but is successful in finding repaired points very close to the constraint boundary. This helps greatly in exploring the extent of the region of feasible patients.

The second relevant difficulty found in preliminary runs is the balance between controllers and patients, which affects the diversity in the populations. Early in the run, randomly generated controllers tend to handle randomly generated patients very well, so that the competition is unbalanced in favor of controllers. Since this happens early in the run, competitive fitness sharing never has a richly structured competition from which to select diverse, strong competitors. The run may never progress far. Even if a more balanced competition is eventually established, the diversity of niches required to progress to a solution may not be present.

Several measures help overcome these problems. First, the patient population is made much larger to give it an advantage. The patient population has 1024 individuals, while the controller population has 128 individuals. The large patient population also aids it in navigating a more difficult constrained search space, and in maintaining patient diversity. Second, brood selection is not used. Brood selection tends to weed out poor controllers early in the run, when they could provide valuable fitness information. Later in the run, it can destroy slightly inferior niches more quickly (by emphasizing contributions from a parent from a superior niche), resulting in diversity loss. Finally, the phantom parasite should help prevent persistent balance problems; this is discussed further below in Section IV.A.4.

Remaining parameters were set based on intuitions about the problem and observations during preliminary runs; few alternatives were explored for these. Uniform crossover is used for controllers. Offspring are created via recombination with 50% probability in the controller population, and 100% probability in the patient population. Gaussian mutations are used for patients and controllers, with standard deviation chosen to be about one-tenth of the total range for each parameter. Mutation affects only a single randomly chosen parameter, and is always applied to each offspring. An elite size of 1 is used for controllers, and 10 for patients. The sample includes 30 parasites from the previous generation, and 20 from the hall of fame; the smaller sample from the hall of fame makes sense because runs are not very long.

Results

Three runs were started with the setup described above. For two of the runs, controllers began to dominate after 100-200 generations. Only the third run, for which this was not true, was continued (for 550 generations); this is the run discussed here.

Figure IV.3 shows the arms race between controllers and patients for this run. Controllers change going down the vertical axis, and patients change going across the horizontal axis. The arms race is strong in very early generations, but then breaks down into a noisy checkerboard pattern. This checkerboard indicates an alternation between multiple niches, none of which dominates all opponents. There is a fairly regular alternation between niches coming out on top (thus being entered into the hall of fame) when no single niche dominates. In the middle of the run, a controller appears that is perfect against all prior generations. This is shown by a "*" in Figure IV.3. When it appears, patients exist that defeat it, so that it does not take over the population; the checkerboard alternation is eventually restored. The patient population gradually loses the diversity it needs to cover the controllers (probably due

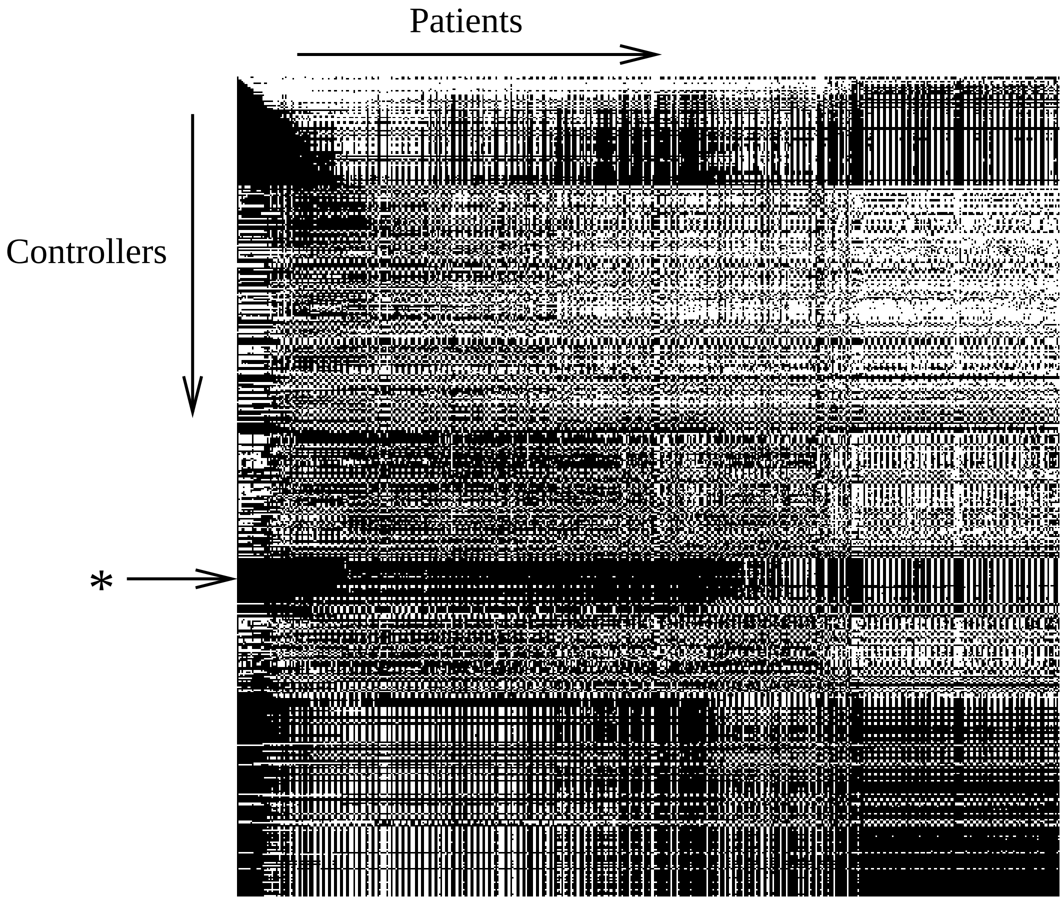


Figure IV.3: Arms Race: Controllers vs. Patients

to random drift), so that by the end of the run controllers are found each generation that are perfect against the current sample. At that point, the hall of fame is critical in revealing imperfections in the controllers.

No perfect controller is found during this run. It may be that no perfect controller exists. To provide more support for this conclusion, we select a subset of patients from the hall of fame, and do extensive search in an attempt to find a controller that can handle all of these. The subset is selected, by a greedy set covering algorithm, to cover all controllers in the hall of fame. To provide robustness, the cover is expanded until each controller in the hall of fame is covered at least five times. This yields a set of 23 patients. Figure IV.4 shows the uncontrolled trajectories for 5 representative patients from this set of 23. These patients show extremes in behavior, with some trajectories visiting both ends of the allowable range of blood pressure. The diversity of these patients makes them difficult to cover with a single controller.

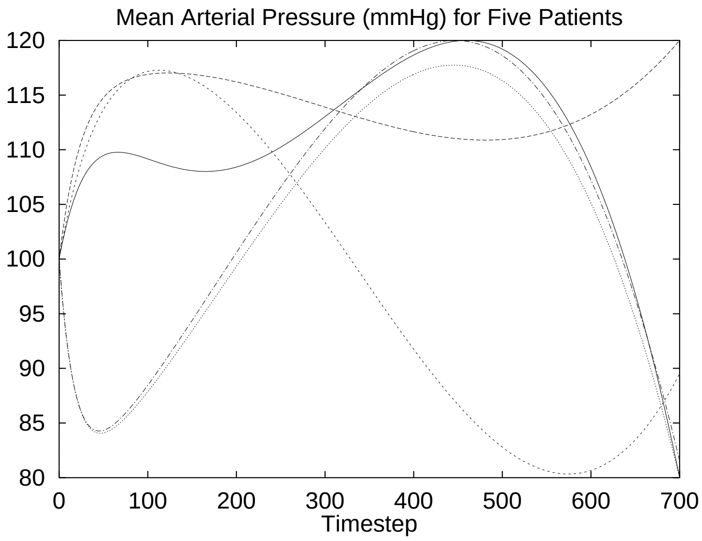


Figure IV.4: Difficult Patients from the Hall of Fame

A non-coevolutionary genetic algorithm is used to search for controllers that can control all 23 patients without killing them. Fitness for controllers is the sum of outcomes over all patients in this test set. Use of binary outcomes for each patient in this non-coevolutionary search results in a landscape with large plateaus where all nearby controllers handle the same set of patients. To provide a smoother landscape, partial credit is used to break ties between controllers that handle the same number of patients correctly. Partial credit for a single patient is linearly related to the maximum amount by which blood pressure exceeds the death threshold; partial credit for a patient is near maximal when the controller comes very close to letting the patient live. Partial credit is summed over all patients that defeat a controller.

Mutation is allowed to affect all parameters, to escape local maxima where single-parameter mutations are inadequate. Two experiments with a population size of 100 were run for 1000 generations, and three experiments with a population size of

1000 were run for 140 generations. All runs ceased producing fitness improvements well before termination.

The best controllers found still fail on 3 of the selected 23 patients. Such controllers were produced by two of the runs with population size 1000; the best controllers produced by the remaining runs failed on 5 of the 23 patients. This lack of successful controllers is consistent with results from similar analysis of earlier coevolutionary runs using different coevolutionary setups. These extensive search results provide strong evidence that there does not exist a perfect controller under this model.

It should be noted that careful attention to coevolutionary balance is required to obtain complete sets of difficult patients, such as the set described above. In preliminary runs that used different parameter settings and suffered from balance problems, a controller would arise that could not be defeated by any patients produced by subsequent coevolution. In the manner of [86] as described above, a separate test was done to evolve patients that maximize error against one of these successful controllers. This test produced no patients that died under this controller's care; from this test, the controller appeared to be perfect. But patient death, rather than error, is the performance measure we care about here. So, the test was modified to evaluate patients based on the minimal distance between their controlled blood pressure and the death threshold. This modified fitness measure should produce more difficult patients than error, since error is not monotonically related to distance from death. Indeed, with this fitness measure, patients could be found that defeated the controller. Such patients were not found during the stalled coevolutionary run that produced the controller. Careful control of balance is required to complete the search for difficult patients.

IV.A.4 Effect of the Phantom Parasite

This application provides an opportunity to observe the effect of the phantom parasite on coevolutionary balance. The phantom parasite has little impact on coevolution for the nearly symmetric test problems used in Chapter III. The natural asymmetry in this controller design problem increases the importance of the phantom parasite.

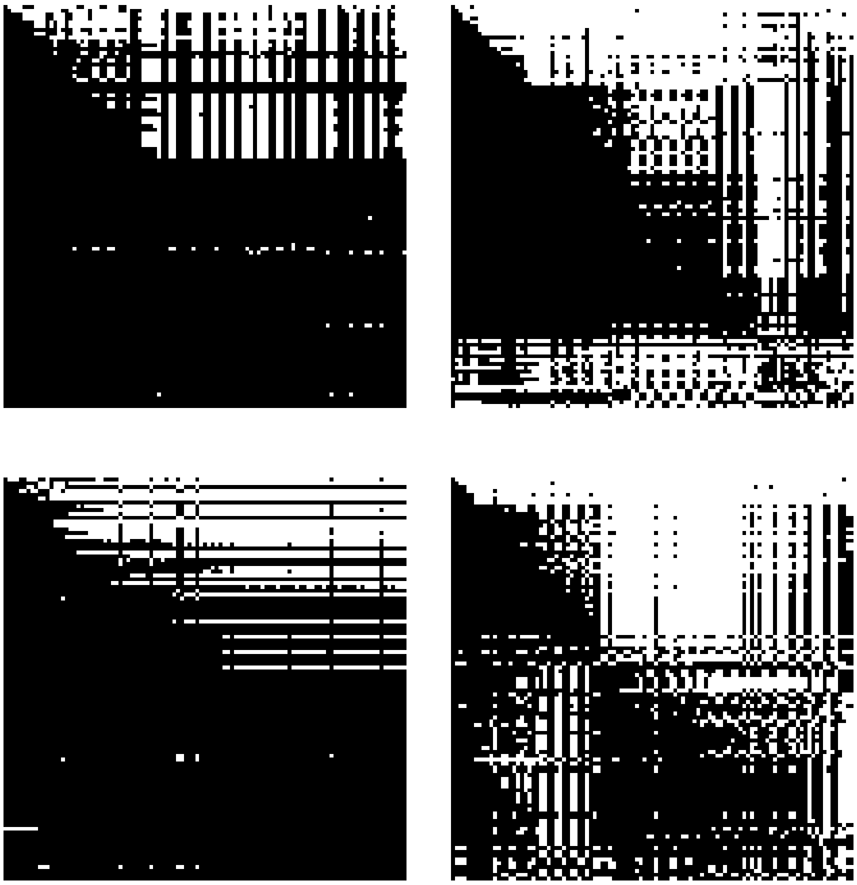
Two short runs (105 generations) were done with the phantom parasite, and two without, using parameters from Section IV.A.3. The only parameter change was to use equal-sized populations of 512 each, to make balance problems more pronounced than they would be with the unequal-sized populations used above.

In all four runs, controllers were dominant in most generations. This dominance was far more severe in the runs that did not use the phantom parasite. In both of these, a single undefeated controller emerged as the elitist by generation 50, and was not defeated by any patient for the remainder of the run. Without the phantom parasite, such undefeatable controllers eventually occupy most of the controller samples. Pedagogical controllers, that could provide informative fitness and guide patients towards defeating the elitist, are neither selected nor sampled.

With the phantom parasite, there is much more turnover in the controller population. No single controller persists undefeated for very long. This difference may be seen in the arms race diagrams in Figure IV.5. With the phantom parasite, controllers are frequently defeated by patients in the hall of fame. Without the phantom parasite, the best patients rarely defeat controllers from the last half of the run. [4]

Balance problems are not as extreme in runs that use a smaller controller population. Still, one run (out of three) using the unequal population sizes from Section IV.A.3 but no phantom parasite produced a result like that seen above; an elitist controller emerged early in the run and was never defeated. This did not happen in any of three runs with the phantom parasite and unequal population sizes.

[4]The single undefeated elitist that arises in runs without the phantom parasite is *not* the same as the individual chosen for the hall of fame each generation. Different individuals are chosen for the hall of fame each generation. This is why there are a few defeats for controllers from later generations in the hall of fame, even though the elitist was never defeated.



(a) Without Phantom Parasite (b) With Phantom Parasite

Figure IV.5: Arms Race With and Without Phantom Parasite

IV.A.5 Discussion

Several characteristics distinguish this adversarial problem and make it difficult. The problem is has continuous parameters, and complex nonlinear constraints on feasible test cases. Genetic operators are designed to explore the boundaries and extent of this constraint region. There is a natural asymmetry between solutions and test cases, which contributes to severe balance problems. The phantom parasite, and the use of unequal population sizes, helps overcome these problems. Finally, the problem is especially difficult because it has no single perfect solution; coevolution instead must find a complete set of test of cases that can prove this.

Coevolution is successful, within the limitations of the given model. The product of coevolution is a set of patients for which there is no adequate controller. Although the result obtained is not the one desired for the success of the application, it is a useful answer given the current model. The result is positive in that it shows the existence of small teaching sets for this problem. Beyond this, the experiments indicate that the controller specification is inadequate and needs to be improved, or that the patient constraints are too liberal. The difficult patients found should be useful for determining which of these is the case, and may aid further development of the system.

When given a new model of interaction that forms the basis for an adversarial problem, we may not know *a priori* the details of the structure of the competition. We would like to know the sizes of the sets of solutions and test cases that are required to cover all competitors. This would reveal whether perfect solutions and small teaching sets exist, and may help in improving the model. The results in this section indicate that coevolution is not limited to finding perfect solutions when they exist. Coevolution can differentiate between adversarial problems that have perfect solutions and those that do not, and can find covering sets of solutions and test cases. This makes it a useful tool during the development of models of interaction.

IV.B Sorting Networks

IV.B.1 Introduction

For most adversarial problems, little information is available in advance about the difficulty of testing, or about the effect of various choices of representation and bias for candidate solutions on this difficulty of testing. In this section we discuss sorting network design, an application for which the difficulty of testing can be theoretically analyzed to some extent before experiments take place. Alternative choices of bias on the space of candidate solutions in this problem allow us to explore the behavior of coevolution under varying conditions of theoretical testing difficulty.

Sorting networks are sorting algorithms of a particular simple form, suitable for hardware implementation [52, 50]. Sorting networks consist of a sequence of compare-exchange gates that operate on a fixed-length sequence of numbers. Each compare-exchange examines two particular elements (for example, the third and fifth elements in the sequence) and swaps them if they are out of order. Since it easy to run a candidate sorting network on a test sequence and see whether the output is correctly sorted, the design of sorting networks that correctly sort all sequences is a natural adversarial problem.

Existing theoretical work on sorting networks yields insights into the difficulty of testing during coevolution of sorting networks and test sequences. We discuss the implications of relevant theoretical work below, and review coevolutionary aspects of prior experimental work on sorting network design. We present a small set of experiments that explore the behavior of coevolution under different conditions of theoretical testing difficulty. We then discuss the behavior of coevolution in domains where testing is difficult.

IV.B.2 Implications of Prior Work

Theory: Specification Number

Recall that a teaching set is a set T of test cases, such that for each possible imperfect candidate solution C, at least one member of T is handled incorrectly by C. Any candidate solution that correctly handles every member of a teaching set must be perfect. Specification number is defined to be the size of the minimal teaching set. In theoretical performance bounds on simplified coevolutionary algorithms, specification number provides a measure of testing difficulty; see Chapter II for more details.

Prior theoretical work on sorting networks allows us to determine specification number for the sorting network design problem. A completely naive test of candidate networks might use all $n!$ permutations of n distinct inputs. The *zero-one* principle [52] shows that this is unnecessary: any sorting network that sorts all 2^n sequences of zeroes and ones will sort any sequence correctly. Since sorting networks cannot disrupt any of the $n+1$ sorted sequences of zeroes and ones on n inputs, the zero-one principle places an upper bound of $2^n - n - 1$ on specification number for sorting networks on n inputs.

We can use a result of Parberry to obtain a matching lower bound on specification number, for reasonably inclusive classes of candidate sorting networks [72]. Parberry has given a general construction for *single-exception* networks that correctly sort every unsorted input except for one particular sequence. Given an arbitrary n and an unsorted sequence s of zeroes and ones of length n, the construction yields a single-exception network that sorts every sequence but s. The single-exception network is comparable in size to the smallest completely correct network on n inputs. Reasonably inclusive classes of candidate sorting networks would include such single-exception networks. As long as the class allows arbitrary sequences of compare-exchanges, of length equal to or slightly larger than the optimally small network length, the class includes the single-exception nets. To show that a net from such a class is not one of the single-exception networks, it would have to be tested on all $2^n - n - 1$ unsorted sequences; every one of these sequences is required to distinguish perfect from imperfect networks. Combined with the upper bound given above, the specification number of sorting network design, for inclusive classes of sorting nets, is exactly $2^n - n - 1$.

Since the positive results from Chapter II depend on specification number, they become practically useless for a problem such as this in which specification number grows exponentially quickly. For practical search, the large specification number may be less relevant. Parberry's construction gives no clues as to the relative prevalence of single-exception networks, compared to more easily testable networks, or compared to perfect networks. Single-exception nets might only be obtainable from Parberry's precise construction; they may be so peculiar that they are unlikely to ever arise in practice. If this is the case, the testing problem for practical coevolutionary purposes could be much easier than the large specification number suggests. If, on the other hand, such single-exception networks do arise in practice, then the testing problem is probably quite difficult even in practice. Sorting network design is valuable in allowing us to observe coevolution under potentially difficult conditions of high specification number.

For practical search, it is possible to apply a variety of biases that can reduce search space size and search difficulty, along with specification number and testing difficulty. Hillis' work (see below) on search for 16-input networks constrains all nets to start with the "butterfly," a particular sequence of 32 compare-exchanges. Since many of the best human-designed 16-input networks begin with the butterfly, this is a reasonable way to reduce the search space. Baum [14] and Juillé [49] point out that the butterfly also greatly reduces the required testing problem. Only 151 distinct unsorted sequences remain after passing all 2^{16} zero-one sequences through the initial 16-input butterfly, so specification number for sorting nets biased by the butterfly is 151. This is far smaller than the specification number of 65519 for unbiased 16-input networks.

Although it would be possible to manually restrict testing of butterfly-biased networks to a complete set of 151 sequences, coevolutionary experiments such as those performed by Hillis [44] are still valuable in this domain. Below, experiments with such biased networks serve as a basis of comparison against coevolutionary experiments on unbiased sorting networks, where specification number is much larger. Also, the ease of manual testing is very specific to butterfly-biased networks. A variety of useful biases are possible (for example, developmental biases [50]), not all of which possess small test sets that are easy to analytically derive. So, the general study of coevolution on biased classes of networks is worthwhile.

Hillis

Several authors have explored the use of genetic algorithms and related evolutionary methods for sorting network design. The original work along these lines is due to Hillis [44], and it is the most relevant work here.

Hillis used an evolutionary algorithm to design 16-input sorting networks

[44]. This size has been a focus of efforts to produce nets with a minimal number of compare-exchange gates. Hillis constrained networks to begin with the butterfly, and evolved the remainder of the network. Experiments used a diploid representation that allowed for a variable number of compare-exchange gates. Evolution is spatially distributed on a 2-D grid to exploit the parallel Connection Machine used in the experiment.

Networks were selected for ability to sort correctly, as measured by performance on a collection of test sequences. Sequences were restricted to consist of zeroes and ones; such sequences are sufficient by the zero-one principle. Initial experiments used randomly chosen tests from this space. This succeeded in producing correct sorting networks, the shortest of which were 65 compare-exchanges long. This is several longer than Green's network, the shortest known on 16 inputs with 60 compare-exchanges [52].

In an effort to obtain better results, Hillis tried to do more effective testing by using coevolution. Test cases consisted of 10-20 test sequences, and competition was restricted to be between test cases and sorting networks at the same location on the grid. Coevolution succeeded in producing correct sorting networks, the smallest of which were smaller than those produced with random testing, with only 61 compare-exchanges. Note that coevolution doesn't directly increase selection pressure for *shorter* networks. Hillis wrote that coevolution drove a continuous exploration that was not seen in the experiments with random testing. Presumably this, coupled with a larger number of correct sorting networks found overall, accounts for the improved size of correct sorting networks found with coevolution.

There are several important differences between competitive coevolution as described in Chapter III, and the form of coeovlution used by Hillis. The placement of 10-20 test sequences on each test case genome should encourage greater diversity of test sequences in general. But, since this number of test sequences on the genome is too small to form a complete teaching set, coevolution must still rely on diversity in the test case population for complete testing. Search would become very impractical if complete teaching sets (of size 151 in this case) could be built on a single test case genome, and smaller numbers of multiple test sequences per test case may still complicate search. Since we need to rely on a diverse population of tests anyways, it seems simpler for most problems to focus on getting coevolution to work properly using this mechanism for providing complete tests, rather than employing an ad hoc mechanism for creating test diversity.

One exception to this reliance on diversity within a test case population may arise when testing is very difficult and specification number is very high. If there is little hope of the test case sample containing a near-complete teaching set, then it may be more important to evolve difficult sets of tests that are appropriate to current opposition; this is discussed in more detail in Section IV.B.5. Hillis' technique of placing multiple test sequences on a single test genome may allow this evolution of difficult sets of test cases, such that it is especially difficult to sort all of the tests in a set correctly with a single network.

The spatial aspect of Hillis' coevolution is also interesting. While competitive fitness sharing helps preserve diversity once a richly structured competition exists between the populations, it cannot build diversity when one niche is dominant. Such a situation can easily arise near the beginning of the run, when most random competitors are very poor and search simply selects the arbitrary individuals that happen to be more successful. The situation may also arise near the end of the run when near-perfect solutions begin to dominate. The spatial distribution may be of value in maintaining diverse niches in such situations. This is discussed further in Chapter V.

Other Experimental Work

Juillé used a problem-specific distributed search algorithm to find sorting networks *without* using the initial bias of 32 compare-exchanges used by Hillis [49]. The algorithm used exhaustive testing in a specialized way. Compare-exchange gates were added sequentially to an initially empty network. As partial networks were built, the set of remaining unsorted test cases was tracked; once the set becomes empty the sorting network is correct. Search proceeded by reproducing successful partial networks into neighboring locations on a grid. The method produced correct 16-input networks with the minimum known length of 60 compare-exchanges, and improved the best-known result for 13-input sorting networks. While quite successful, it is difficult to compare results from this largely problem-specific technique with those obtained by more general search algorithms and coevolutionary methods.

Kammeyer and Belew used a grammar-based developmental representation to evolve specifications that could generate sorting networks of arbitrarily large size [50]. During evolution, small networks were generated from candidate grammars and tested exhaustively. The best specifications resulting from evolution were checked manually to prove that they could generate correct networks for any number of inputs. This work succeeded in producing novel families of correct networks.

The developmental aspect of this work helps make the testing problem tractable. Search over arbitrary large networks would be difficult, due to the extensive testing required. By only generating large networks that are developmentally related to small networks, the testing problem can be essentially reduced to that of testing small networks. This may be a useful general approach to reducing specification number and the practical difficulty of testing.

IV.B.3 Experiments

We perform experiments on the coevolution of 16-input networks and sequences. We compare results for "unbiased" experiments that search for complete sorting networks, to those from "biased" experiments that constrain all sorting networks to start with the butterfly. By obtaining results on unbiased networks as well as networks biased by the initial butterfly, we can compare the behavior of coevolution under different conditions of testing difficulty.

Coevolution takes place between 16-input sorting networks in the first population, and length-16 test sequences of zeroes and ones in the second population. The coevolutionary setup discussed in Section III.H is used for these experiments.

This setup uses competitive fitness sharing, shared sampling, the hall of fame, the phantom parasite, and brood selection.

Our primary goal is the search for correct networks, but we also follow Hillis in seeking sorting networks that are short (have a small number of compare-exchanges). We add selection pressure for length by breaking fitness ties (for purposes of tournament selection, elitism, and brood selection) in the sorting network population by choosing the network with shorter length. This simple method may not provide a high-performance search for optimally short networks (see Juillé's method for a performance system [49]), but it should prevent networks from becoming excessively long.

In the effort to produce shorter sorting networks, evolution may meaningfully proceed for many generations after the first perfect network is found. With the phantom parasite, the evolution of the sorting network population may become oscillatory after this perfect network is found. Perfect networks win all selection tournaments until they take over too large a fraction of the population. At that point, the gain for imperfect networks from the phantom parasite becomes large enough so that they win all competitions and take over the population. Once imperfect networks are common, perfect networks can take over again, and so on. To help prevent such oscillations from becoming problematic, we use *continuously updated sharing* in the sorting network population to give perfect and imperfect networks fairly stable niche sizes within each generation. This method measures shared fitness within the new generation being constructed [69], and was described in section III.E.2.

In our experiments, a sorting network is represented directly by an ordered list of arbitrary compare-exchanges on the 16 elements. In biased experiments, this list is appended to the butterfly before testing. List length may be modified by the genetic operators, but has an upper limit of 50 compare-exchanges in the biased experiments (in addition to the butterfly's 32 compare-exchanges) and 100 compare-exchanges in unbiased experiments.

Sorting network mutation occurs with 1% probability per compare-exchange, and replaces the compare-exchange with a randomly chosen compare-exchange. Sorting network crossover replaces a contiguous sequence of compare-exchanges from one parent with a (possibly different length) contiguous sequence of compare-exchanges from the second parent. This operation may be highly destructive, so we bias it towards replacing and inserting short sequences, as follows. Length of the sequence starts at zero, and is incremented by one with 75% probability. This is done repeatedly, so that the probability of the sequence having length at least 1 is 75%, at least length 2 is 0.75^2, at least length 3 is 0.75^3, and so on. This is done for both the sequence replaced, and the sequence replacing it. The lengths are usually fairly short, but can occasionally become long. The lengths may be 0, meaning that simple insertions and deletions are possible consequences of crossover.

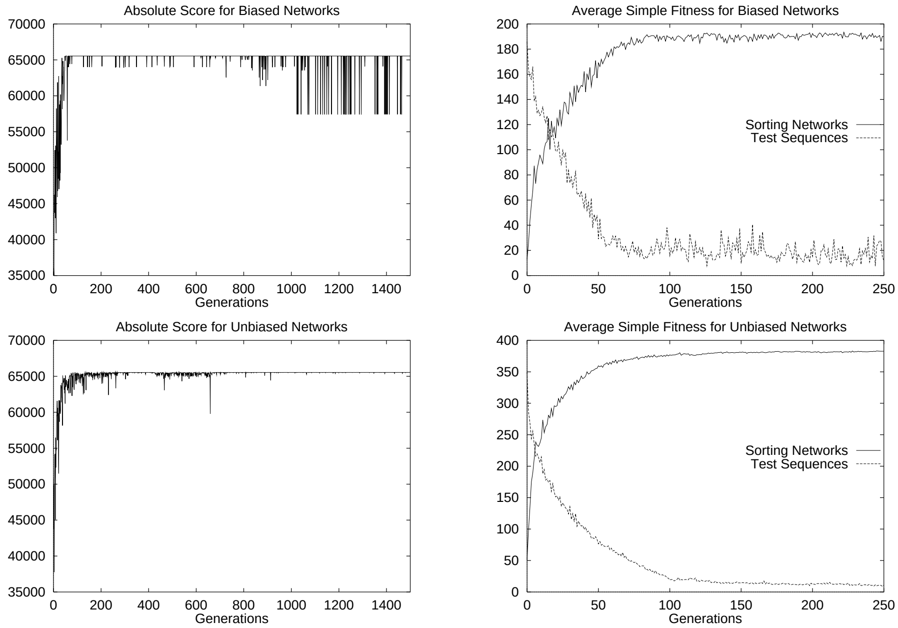
Test sequences are bit vectors. Standard two-point crossover is used for test sequences. Mutations are bit-flips that occur with 2% probability per bit.

In experiments described below, each population has 2000 individuals. Experiments with larger population sizes yielded results qualitatively similar to those described below. Brood selection creates a brood of 5 individuals in the biased experiment, and 10 in the unbiased experiment, testing the brood against a subsample of 40 parasites. 200 parasites are used for the biased experiment, and 400 for the unbiased experiment; in both, hall of fame parasites make up 25% of the sample. Elitism is used with an elite size of 10, and tournament selection is used with a tournament size of 2.

IV.B.4 Results

Biased Experiments

Runs biased by the butterfly are successful in producing perfect sorting networks, requiring under 100 generations to do this. The top row of Figure IV.6a shows, for one biased run, the results of testing the best individual (according to simple fitness) each generation on all 2^{16} test cases. There is a sharp rise in this absolute score



Biased

Unbiased

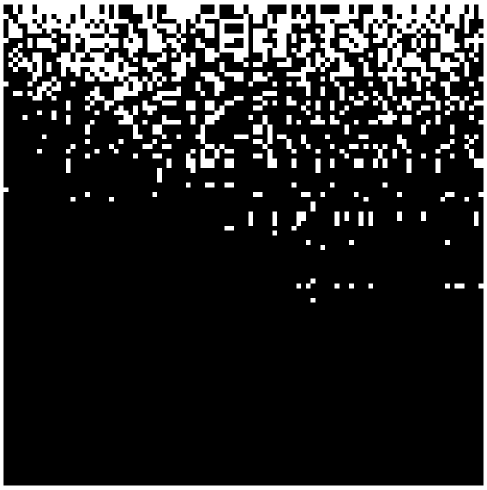
(a) Absolute Score **(b) Simple Fitness**

Figure IV.6: Sorting Network Evolution

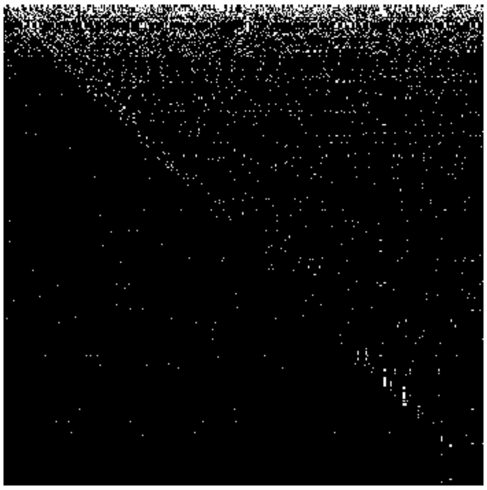
in early generations, with the first perfect individual found at generation 48.

This first perfect individual contains 82 compare-exchanges, the maximum possible. This initial size is unsurprising since more compare-exchanges is always better from the perspective of correctness; larger networks are better on average. Since size is used to break fitness ties, it is gradually reduced during later generations. During this period, smaller imperfect networks arise that appear perfect against current parasite sequences for one generation, and are therefore selected as the best and displayed as downward spikes in absolute score in Figure IV.6a. In shorter networks found in later generations, these imperfections are more destructive, and downward spikes become larger. These large spikes continue to the end of the run (3000 generations). A perfect network with 61 compare-exchanges is found in generation 1064, and no further progress is made in size.

The 61 compare-exchange network matches Hillis' best result. A comparison with Hillis on the basis of search time is difficult, since Hillis does not exactly state how long coevolutionary runs took. Non-coevolutionary were said to run for up to



(a) **Biased**
100 generations



(b) **Unbiased**
350 generations

Figure IV.7: Sorting Network Arms Races

5000 generations with a population of 65536 [44]. In coevolutionary runs, with 10-20 parasites per test, this would be about 3-6 billion total competitions. Hillis states that coevolutionary runs were faster than noncoevolutionary runs, but it is unclear whether this was due to reduced testing per sorting network, or a smaller number of generations. Our coevolutionary run discussed here required 1.7 billion competitions to produce a a perfect network with 61 compare-exchanges.

The top row of Figure IV.6b shows the first 250 generations of average fitness in the evolution of biased networks. In the early part of the run, sorting network fitness increases quickly. Test sequence fitness drops quickly as individual tests are able to defeat fewer and fewer networks. During this time, the number of test sequences required to cover all current sorting network flaws increases; the testing problem becomes more difficult as networks evolve to have few flaws. Except for fluctuations like those seen here, average fitness is flat for the remainder of the run. Maximum simple fitness is 200, and the average sorting network fitness stays fairly close to this after perfect networks are found.

Figure IV.7a shows the arms race for the first 100 generations of coevolution in biased networks. Sorting networks change going down the vertical axis, and test sequences change going across the horizontal axis. Black dots show sorting network victories. Sorting networks quickly gain in ability to cover sequences, but sequences do not gain in ability to cover sorting networks after the first few generations. Sequences continue to expose flaws in sorting networks until the networks are perfect, but the number of sequences required to cover flaws in all networks rises as networks approach perfection. The competition is heavily unbalanced in favor of sorting networks, and the arms race resembles that seen for the unbalanced 2DTTT test problem in Section III.G.5.

Unbiased Experiments

Unbiased runs require a larger sample to produce perfect sorting networks; runs using a sample size of 200 did not find perfect networks, even after 1000 generations. The bottom row of Figure IV.6a shows, for one successful unbiased run using a sample size of 400, the results of testing the best individual each generation on all 2^{16} test cases. The initial rapid rise in score resembles that seen in the biased case. The unbiased run, however, spends more time close to perfection before finding the first perfect network in generation 309. This initial perfect network has 96 compare-exchanges. This gradually decreases, until a perfect network with 78 compare-exchanges is found in generation 1166. Network sizes never get as small as they do in the biased case; this may be due to the greater rarity of perfect networks among short unbiased networks than among short biased networks. The larger networks emerging in the unbiased run seem to contain more compare-exchanges that only help sort a few sequences: removal of such compare-exchanges yields tiny downward spikes in Figure IV.6a, compared to the large downward spikes that result from compare-exchange removals in the (presumably) near-minimal networks arising late in the biased run.

Shortly before perfect networks are found in the unbiased run, and for the duration of the run after perfect networks are first found, single-exception (and

$2, 3, 4, \ldots$-exception) networks repeatedly arise as the best elitist in the population. They are typically undefeated by any parasite test sequence in the generation in which they arise, but are defeated in the next generation by a newly selected parasite. This is discussed further in Section IV.B.5.

The bottom row of Figure IV.6b shows average fitness in the evolution of unbiased networks. The trend is similar to that seen for biased networks. Fluctuations are smaller in the unbiased case, presumably due to the finer granularity of performance changes without the butterfly. Beyond the 250 generations shown in this figure, average fitness eventually flattens out after about 1000 generations, at a level of about 387 for sorting networks and 5 for test sequences. These fitnesses are more extreme (accounting for the larger number of parasites used in the unbiased run) than in the biased run. Each test sequence defeats fewer sorting networks than in the biased case; the number of test sequences required to cover the sorting network flaws is larger in the unbiased case.

Figure IV.7b shows the arms race for the first 350 generations. This resembles the arms race seen for biased networks; sorting networks quickly gain in ability to cover test sequences, but individual test sequences do not gain in coverage after the first few generations. Since the first perfect network is not found until generation 309, evolution spends a great deal of time exploring networks that are imperfect but have few flaws. During this period, the diagonal is visible in the arms race diagram as test sequences track current sorting network flaws. This ability is retained even after a perfect network is first found, as imperfect networks arise during the search for shorter networks.

IV.B.5 Discussion

In both of these experiments, coevolution is successful in finding perfect sorting networks, then finding progressively shorter ones. This application is unusual in its use of coevolution; long after the adversarial problem is already solved (perfect networks are found), coevolution is able to retain adequate testing ability to expose flaws in novel shorter networks. As a result, perfection is largely maintained in the best networks as they become shorter.

The single-exception nets that give rise to large specification number in the unbiased case are common enough to arise during practical heuristic search; the experiment shows that they are not merely peculiar theoretical constructions. This indicates that complete testing is difficult in practice for the unbiased network design problem. For this problem, specification number seems to be a realistic measure of testing difficulty: there does not appear to be a reduced set of test sequences which would form an adequate teaching set that could be fixed in advance.

Several other factors indicate that testing is practically more difficult on the unbiased problem than on the biased problem. A larger sample of parasites is required for successful runs, indicating that more extensive testing is required each generation. Evolution in the unbiased case spends more time on the plateau near perfection before finally producing the first perfect sorting network; this may be partly due to the difficulty of distinguishing perfect networks from near-perfect ones. Finally, parisitic fitness is lower in the unbiased case after perfect networks are found, indicating a greater difficulty of covering the remaining imperfect networks in the sorting network population.

Despite the greater difficulty of testing unbiased networks, the large difference in specification number, and the larger search space, coevolution is not vastly more difficult in the unbiased experiments. The time required to find the first perfect network is only a few times larger in the unbiased case than in the biased one. Testing is more difficult, but even single-exception networks have flaws quickly exposed, typically after just one generation. Shorter networks are much more difficult to find in the unbiased case, but this may be due to a much lower frequency of short perfect networks in the unbiased case than in the biased case.

This small difference in performance on these two experiments indicates that coevolution is able to successfully test candidate solutions even when testing is difficult and the test case sample is far too small to contain a complete teaching set. This is partly due to the tendency of a coevolving population to closely track its opposition. Such tracking is possible because the opposing population consists of similar individuals that cannot make sudden jumps across the search space each generation. This tracking can be observed in the ability of the test sequence population to expose flaws in single-exception networks after only one or two generations. If these single exceptions were generated uniformly from the $2^{16} - 17$ possible exceptions, or if the test sequence population did not track its current opposition, this would not be possible given that the test sequence population is much smaller than $2^{16} - 17$.

To see how this tracking operates, consider a parasite sample of sorting networks. A large number of test sequences (the entire population) are evaluated against these sorting networks. If each sorting network has flaws exposed by some test sequence, this is enough to guide selection so that the new population of test sequences has a greater fraction of individuals that successfully expose flaws in the sorting network sample. As long as this sample is reasonably large, the genetic similarity of networks in a single population will help ensure that these selected sequences are generally successful in exposing flaws in the entire sorting network population.

This behavior helps explain how a coevolving population can track its opposition, even when a complete cover of all possible imperfect opposition is prohibitively large. The current opposition consists of a population of related individuals, with flaws that can be exposed by a small subset of this complete cover. By the behavior described above, a population may track its opposition as its small set of flaws slowly changes, even though the test population never gains an ability to completely test all potential imperfect opposition.

This tracking yields a reduced type of arms race, that does not produce individual test sequences with increased ability to cover all prior opposing sorting networks. There cannot even be, in the unbiased case, the evolution of a complete teaching set within the population, as was discussed in Section IV.A. Instead, the sequence population tracks the opposing sorting networks, retaining an ability to expose flaws in current opposition while never gaining in ability to expose flaws in arbitrary opposition.

It is interesting to note that this reduced arms race has a theoretical precedent. Section II.D discussed the *covering competitive algorithm*, a simplified coevolutionary algorithm in which new solutions and test cases are always chosen to cover *all* prior opposition. Our heuristic methods are related to this in their use of the hall of fame to promote the defeat of all prior opposition by new individuals in both populations. But the alternative competitive algorithm discussed in Section II.D.5 only needs to force *one* side of the competition to defeat all prior opposition. The other side (the second-player side in the proof of Theorem 6) only needs to defeat a sample of *recent* opposition for the proof to work. The form of coevolution seen in sorting networks is similar to this; test sequences only track recent opposing sorting networks, but sorting networks become progressively stronger against all prior opposing test sequences.

IV.B.6 Summary

Coevolution succeeds in producing perfect sorting networks, in both unbiased sorting network design and with the butterfly bias that reduces search space size and testing difficulty. While our implementation is not carefully tuned to find the shortest possible sorting networks, coevolution does successfully maintain adequate testing capability while producing shorter networks after perfect networks are initially found.

Specification number is very high for the unbiased problem, and the single-exception nets that give rise to this high specification number do arise in practice. Nonetheless, testing during coevolution is not too much more difficult than in the biased case, where specification number is much lower. Even when a large set of test cases is required to cover all possible flaws in imperfect solutions, candidate solutions in an evolved population are genetically related so that a small subset of tests are adequate for exposing flaws at any point during coevolution. By tracking this subset of flaws as coevolution proceeds, the test case population can maintain adequate testing ability against current candidate solutions. Since candidate solutions must succeed against progressively larger sets of prior test cases, they are eventually forced into perfection.

IV.C Coevolving Strategies for the Game of Go

IV.C.1 Introduction

In many problem domains, including most of the adversarial problems discussed in this thesis, all potentially useful solutions have short representations, and there ultimately exists a final solution from this space. In such close-ended domains, computational approaches only need to produce a finite sequence of improvements, until this ultimate solution is found. More complex problem domains in artificial intelligence, such as board games [95], robotics [88], mathematics [55], and artificial life simulations [3], often do not possess such ultimate, perfect solutions of reasonable complexity. Instead, there exist open-ended sequences of useful solutions of increasing performance and complexity. Such problem domains offer an opportunity to demonstrate open-ended intelligent systems capable of climbing these sequences to solutions of arbitrary performance and complexity.

Coevolutionary methods are a natural approach to open-ended adversarial problems. In an arms race, new individuals are produced to defeat prior individuals; there is no reason that this process has to end with some final solution. By allowing solutions and test cases of increasing complexity, it may be possible to extend the arms race indefinitely.

Board games provide a natural competitive environment for studying open-ended problems. Although finite board games such as chess and Go must have some perfect winning strategy, this winning strategy is likely to be so complex that, for practical purposes, we may treat the games as open-ended domains. The simple rules that define board games permit efficient computational approaches and comprehensible results, while still allowing deep strategic possibilities.

In this section, we discuss the coevolution of strategies for the game of Go. We begin with a description of the game of Go and features that make it an attractive test problem. We discuss prior work on computer Go and the limited capabilities of existing programs. Drawing on properties of the game and lessons from preliminary experiments, we develop a representation for strategies that allows relevant computation to be done efficiently. We describe the results of coevolution using this representation in terms of the nature of evolved strategies, performance of these strategies, and measurements of useful complexity during coevolution. We suggest possible directions for future research, including the possibility of using coevolution in the creation of performance-oriented Go-playing programs.

IV.C.2 Rules of Go

The game of Go originated in China thousands of years ago, and has a long history of popularity in China and Japan. This history includes includes a tradition of highly-skilled professional play that extends to the present day. A variety of rulesets are in use, but variations between them are minor. Here, we give a brief description of rules used in our experiments.

Go is played on a square grid of points. A 19x19 grid is usually used, but beginners often play on a 9x9 board; this size is still very challenging and we use it in experiments described here. Players alternate placing stones on the grid in an attempt to capture territory. The first player plays black stones, and the second player plays white stones. Each move, a player places a stone of his color at an intersection on the grid.

A *string* is a set of stones of the same color, connected by horizontal or vertical adjacency on the grid. A *liberty* of a string is an empty point adjacent to the string. If a stone placed by one player results in an opposing string with zero liberties, this opposing string is captured and is removed from the board. A player cannot make suicide plays that leave any of his own strings with zero liberties.

The *Ko* rule forbids repetition of prior positions from the game. This rule varies in minor ways among popular rulesets; here, we simply forbid pairs of single-stone captures that result in a repeated position two moves after its original occurrence.

On any move, a player may pass instead of placing a stone. The game ends when both players pass in succession. Scoring at the end of a game also varies in minor ways among different rulesets; we choose a simple scoring method that is appropriate for computer play. At the end of the game, any empty intersection I is defined as *territory* for a player if there is no path of adjacent (horizontal or vertical) empty intersections from I to a stone of the opponent's color. The score for a player at the end of the game is the number of stones of that player's color on the board, plus the number of points of territory for that player.

IV.C.3 Go as an AI Problem

Although a number of Go programs are in development by various researchers, recent published work on computer Go is fairly rare [56, 85, 46, 20, 77]. More information about existing Go programs and issues in computer Go is available from the Internet [11, 1, 2]; refer to these for further information about topics discussed below.

Beyond providing an open-ended adversarial domain, Go has several additional features that make it attractive as a test problem in artificial intelligence [20]. It has proven quite difficult for computers; current programs play at a low intermediate level that does not come close to the mastery with which computers play other games such as chess [94]. The typical approach to computer game play involves massive search using a fairly simple evaluation function to compare moves. This approach has not worked well at all in Go. Since almost every open intersection is a legal move each turn in Go, the number of choices to explore for each move is larger than in most board games for which search has been successful (for example, an average of roughly 200 for Go as opposed to 35 for chess[20]). Furthermore, a single move can have subtle effects on the position that are difficult to compare with the effects of other move choices. Finally, there are no known methods to rapidly obtain accurate evaluations of positions in Go, making it impossible to do efficient extensive search. Even on a 9x9 board, where search should be more tractable than on the full 19x19 board, programs still only play Go at an intermediate level.

The most successful existing programs for playing Go combine several techniques in a rule-based framework similar to expert systems. Pattern databases are used to suggest reasonable moves. Search is primarily used within small areas of the board to evaluate tactical features of the position (for example, to decide whether or not strings can be captured). Extensive knowledge is required to guide this search and keep the time required for it reasonable; databases of common patterns and outcomes are used for this purpose. Heuristic methods are used to evaluate the longer-term ability of strings to connect to one another and form permanent live groups. An influence function radiates from each stone on the board to obtain a rough heuristic estimate of which player controls more territory.

Part of the difficulty of computer Go is that these different techniques are hard to integrate, even though results from the techniques should be interdependent. Integration problems may become worse as programs become stronger and more complex.

Since brute force approaches, based on massive search with a simple evaluation, fail in Go, the game seems more attractive as a test domain for research in contemporary artificial intelligence than other games such as chess. The requirement in current programs of integration of disparate techniques suggests that Go may demand a broad array of cognitive capabilities. Since direct programming has not produced strong Go programs, the domain is attractive as an application for machine learning and other novel approaches. Such novel approaches have hope of improving on the state of the art sooner than in other games such as chess where conventional methods already work so well.

Reinforcement learning has been used to train neural networks to play Go [85, 31]. While this method has been very successful in other games such as backgammon [95], it has had only limited success in Go. Part of the reason for this may be that learning-based Go programs have been simple experimental systems. While appropriate for research, these systems lack the extensive work that has provided knowledge-intensive focused search and other capabilities of conventional Go programs. Hybrid programs that combine conventional features with adaptive methods may be a promising direction for producing stronger Go programs; this is discussed in Section IV.C.7.

IV.C.4 Representation for Coevolutionary Go

Issues in Representing Go Strategies

Even before considering details of the game that affect representation choice, our Go strategy representation for coevolution must balance several factors. First, strategies must be computationally efficient. Even on a small 9x9 board, a game of Go can easily last almost 100 moves. Since millions of games must be played during coevolution, each move must be made in a small fraction of a second. At present, this places severe limitations on the types of computation that can be used, and the techniques that can be imported from existing computer Go programs.

The second factor is the ease with which evolution can produce useful Go-playing ability. At one extreme, we could give evolved strategies a highly-developed set of features, and simply ask evolution to tune the weights assigned to these various features in producing a final evaluation of a move; this tuning should be evolutionarily easy. But by giving strategies so much information for free, and limiting the computation done by the evolved portion of the strategy, we severely restrict the possibility of long-term arms races. Since we intend to study long-term arms races in an open-ended setting, this choice does not seem appropriate.

At the other extreme, we could give strategies Turing-complete computational capabilities but no useful bias towards computation relevant to Go. With massive computational resources, such an approach might be feasible. In the absence of such resources, evolution is likely to spend most of its time discovering very basic elements of Go-relevant computation, rather than producing viable strategies. This could make arms races slow and results uninteresting.

Between these extremes, we seek a representation with simple computational capabilities that are relevant to Go; important features in Go should be computable by concise combinations of low-level primitives in the representation. Where the representation cannot be easily extended to compute particular important basic concepts, we may give these concepts to evolved strategies as precomputed features. When we do this, we try to provide only simple, exact features, leaving heuristic judgements to be made by coevolution.

Existing learning-based approaches to Go use neural network evaluation functions [85, 31]. To play using an evaluation function, all possible moves must be evaluated each turn. This is computationally expensive, so we instead evolve control functions that directly choose a move. Beyond this difference, our representation shares many capabilities with these neural network representations.

Overview of Representation

Spatial locality is an important feature of Go; most of the important effects of a stone lie in the neighborhood of grid points around the stone. Many of the reasons for choosing a particular move lie in a local neighborhood around the move. Existing Go programs make heavy use of localized pattern matching to take advantage of this locality. This method is the basis for our representation; evolved strategies consist of a set of localized pattern matching rules. Each such rule examines conditions in a local neighborhood of a point, making changes to information associated with that point based on these local conditions.

Each point on the grid has several variables associated with it. Some of these variables contain precomputed information about the position. At the very least this precomputed information includes the status (empty, black, or white) of the point. Several variables do not have any precomputed value; these may be used by evolved

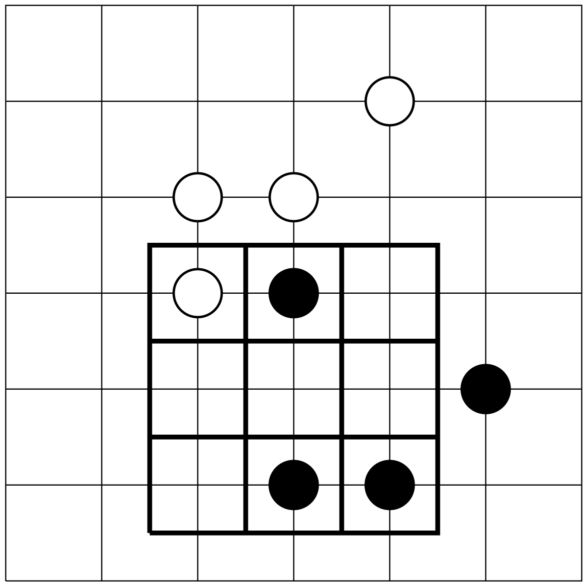


Figure IV.8: Go Board with Neighborhood

rules as temporary storage during the computation.

Rules may modify the value of these variables at a point based on information in a fixed-size local neighborhood around the point. An example of such a neighborhood on a Go board is shown in Figure IV.8.

To make a move, computation begins with all precomputed information placed in the appropriate variables at each point. Other variables are initialized to zero. During computation, each rule is applied simultaneously to each point on the board. That is, the variables at each point are simultaneously modified based on local information, according to the rule. After computation is complete, a move is made based on the value of a distinguished output variable: the legal move with highest associated output variable is chosen for play. If no legal move has an output value above zero, the strategy passes.

This type of computation can be described as a cellular automata. Cellular automata are based on local pattern-matching primitives, but can compute larger-scale features through repeated action of these primitives sequentially in time. Hand-designed cellular automata rules have been used in a program written by Allan Scarff, reportedly with some success [30].

We performed preliminary experiments that used a very simple representation of the form described above, with little added computational capability or Go knowledge [80]. We do not describe these experiments in detail here, since they used a different form of coevolution from the other applications described in this chapter. Instead, we describe representational requirements that are revealed by limitations observed in these experiments. We then give details of our final representation, drawing on knowledge of these requirements.

Lessons from Preliminary Experiments

The preliminary representation uses binary variables at each point. The only precomputed features are two variables specifying the presence of a black or white stone at each point; any other features of the position must be computed by evolved strategies. A third binary variable is used for output to specify the move as described above. Five more variables may be used in arbitrary ways by evolved strategies.

Rules are represented as condition-action pairs, in the style of classifier systems[45]. Conditions consist of 0, 1, or "don't care" for each bit in the points of a 3x3 neighborhood around the central point. Such a neighborhood is shown in Figure IV.8. Actions consist of 0, 1, or "don't modify" for each bit in the central point. A strategy is represented as an ordered list of these rules. Each timestep of the computation, for each point, the action corresponding to the first matching condition (in any symmetric reflection/rotation) in the list is taken. This is repeated until the variables converge to a fixed point, or a maximum number of timesteps are executed.

Genetic operators can mutate one of the condition or action bits in a rule, add a new random rule at the end of the list, swap two rules, or delete a randomly chosen rule. Crossover points are always chosen to be between rules in the list. These operators seem to provide search with an adequate amount of variation.

Coevolutionary experiments were run on 7x7 and 9x9 boards, using an earlier form of coevolution that did not include most of the techniques discussed in Chapter III. After coevolution, resulting strategies were tested against Wally, a weak public-domain Go-playing program that has been used by others to test learned Go strategies [85]. The best evolved strategies from the 7x7 experiment were able to consistently defeat Wally, and the best evolved strategies from the 9x9 experiment were usually able to defeat Wally. While this test does indicate that an interesting level of performance was achieved, strategies were still very weak on a human scale.

Several aspects of this simple representation seemed to hamper the evolution of more competent strategies. First, the rule representation is bulky. Useful rules are those which specify constraints for very few variables in the neighborhood (so that the condition is mostly "don't cares"). If many constraints are specified, the rule never fires. Rather than use a bulky representation that explicitly specifies 0, 1, or "don't care" individually for each variable at each point in the neighborhood, evolution should be more straightforward if rules simply list the constraints that aren't "don't care."

Second, the global decision about which move to make is difficult with the simple binary variable used for move choice. To make the decision between candidate moves of different value, evolved rules must propagate these values across the board and compare them before specifying one of the moves with the binary outputs. In practice, evolved rules are not able to do this complex operation to any great extent, so that global comparisons among different moves are not made well. Since it is easy to compare a wider range of possible output variable values to choose a move after computation is completed, we should lessen the computational burden by allowing such a wider range of output values.

Third, local rules do not have any direct information about absolute location on the board. While Go enjoys a degree of spatial-invariance, many moves (especially in the opening) strongly depend on absolute position on the board. In these preliminary experiments, coevolution was able to produce small sets of rules that could work together over several timesteps to provide information about global position during the opening. While the emergence of such computational capabilities is encouraging, the evolutionary search and strategic computation expended in this effort seem wasteful. To go beyond the evolution of such simple features, towards more competent strategies, we can provide precomputed variables that specify absolute position at each location on the grid.

Fourth, one basic form of nonlocal influence propagates through strings. A string may extend across the board. A move that takes a liberty away from a string affects the entire string, which may include points distant from the move. Since strings are basic units in Go, we can allow computation to treat them as unified entities for some purposes. This should help strategies go beyond moves which have purely local motivations.

Fifth, the representation used in these preliminary experiments makes it very difficult to perform basic counting operations that are important in Go. For example, the number of liberties adjacent to a string is a basic feature in Go. It seems very difficult to compute this feature in the preliminary representation; even if a signal is propagated along a string from each liberty of the string, counting needs to be done by combining these signals. For arbitrarily shaped strings, this combining operation can be quite complex with many possible cases to handle. Since it is awkward for such a basic feature to be so difficult to compute, we can expand the capabilities of the representation with primitives that permit simple counting.

Finally, certain common patterns in Go are difficult to recognize without any ability to search. For example, a *ladder* is a forced sequence of moves that can extend through empty space across the board, with the outcome of the ladder depending on stones distant from its initiation. Since rules always act in parallel over the entire board, computation cannot easily make the sequential, isolated changes required for search. Strategies should have precomputed features giving the results of a simple search that can at least handle basic features such as ladders.

Representation Details

The representation design used in the experiment below draws on the issues discussed above. It is still based on local pattern-matching, but it includes several simple additional features beyond those of the preliminary representation described above. These should make relevant computation easier to evolve and perform. Note that, while there are many details to the representation described below, it is a fairly natural extension of the simple representation used above to include precomputed search features, strings as unified entities, nonbinary values, and counting ability.

A strategy is represented by an ordered sequence of rules, as above. Repeated action of a rule over multiple timesteps was rarely used in a useful way in the preliminary experiments; such repeated action was mostly wasteful. Instead of using rules repeatedly here, we simply go through the list once in this representation. Each rule is applied in turn over the whole board, before going to the next rule; this permits sequential computation. After going through the list, the legal move with highest-value output variable is chosen for play. If multiple moves have the same value, the first one in a fixed left-to-right, top-to-bottom order is chosen.

We described above the need for counting ability, and a wider range of permissible values for move evaluations. To support these in a natural way, variables are no longer binary, and are instead allowed to take on a wide range of values.

Precomputed variables take on values from a fixed finite range. Rules are not allowed to modify these precomputed variables. Separate, modifiable variables are initialized to zero before going through the list of rules, and can take on any integer value during the computation.

Table IV.1 shows the variables associated with each point, and their range of values. Several binary variables give occupancy status. If the point is off-board, all variables except the "off" flag are fixed at zero. Additional precomputed features give absolute horizontal and vertical position of the point. These position coordinates respect the symmetries of the board; for example, the top line of the grid is numbered identically to the bottom line. Coordinates increase from zero at the edge to four at

Feature	Values
Empty point	0-1
Black stone	0-1
White stone	0-1
Off board	0-1
Horizontal line number	0-4
Vertical line number	0-4
Can't move here due to Ko	0-1
This legal move would be capturable	0-1
This legal move leads to capture of an enemy string	0-1
Modifiable variable 0 (evaluation)	any
Modifiable variable 1	any
Modifiable variable 2	any
Modifiable variable 3	any

Table IV.1: Point Variables

the center. Further variables give tactical information (described below), and Ko status of empty points. This Ko information is technically necessary to completely specify a position.

Feature	Values
Capturable enemy string (may have to attack immediately)	0-1
Capturable friendly string (immediate protecting move may save)	0-1
Capturable enemy string (even if enemy is allowed a move to save it)	0-1
Capturable friendly string (even with an immediate protecting move)	0-1
Modifiable variable 0	any
Modifiable variable 1	any
Modifiable variable 2	any

Table IV.2: String Variables

To allow computations that treat strings as a unit, we associate variables with each string on the board. The string variables have identical value for every member of a string, and may be accessed though points in the string during pattern matching just like point variables.

String variables are described in Table IV.2. Several binary tactical variables are precomputed for each string. Three modifiable variables are available during computation; these might be used by evolved strategies to compute the number of liberties of strings, for example.

Each rule consists of a list of constraints and an action. Each constraint specifies a single variable at a particular point in the neighborhood, and an inequality check for that variable against a specified threshold in the range of that variable (0-4 for modifiable variables). The inequality can be $\leq$ or $>$. The neighborhood consists of the 13 points within Manhattan distance two of the central point. An example constraint might specify that string variable 4, at the point which is two grid intersections to the left of the central point, must have a value greater than 3. For a rule to be used at a particular central point, all constraints must be satisfied in one of the 8 possible symmetric reflections/rotations of the neighborhood about that central point.

To add some counting ability, one of the constraints is specified as "countable." Among the matching symmetries from the 8 possible reflections/rotations of the neighborhood of a central point, a count is made of the number of distinct positions in which this countable constraint can lie. This count can range from 0 (if the condition doesn't match in any symmetry) to 4 for the thirteen-point neighborhood used. The action part of the rule specifies a variable in the central point, and a weight (ranging from -4 to +4); the count is multiplied by the weight and added to the value of the specified variable. A flag specifies whether the variable should be initialized to zero before doing the addition. These counting, thresholding, and weighting operations give strategies some of the computational capabilities of neural networks [85, 31].

For actions which modify string variables, this becomes slightly more complicated. For a particular string, all matches of the condition are checked, where the central point in the condition can be any of the member points in the string. Among these matches, a count is made of the number of distinct positions on the board in which the countable constraint may lie; this can be large since the string can have many members which act as central points. This count is then multiplied by the specified weight and added to the value of the specified string variable. Changes to string variable immediately propagate to all member points in the string.

Note that this provides an easy mechanism for counting relevant quantities like the number of points in a string, or the number of liberties of a string. To count liberties, a rule needs a condition with a single constraint that looks for an empty point adjacent to the central point. When the action of such a rule modifies a string variable with a weight of 1, each string receives a count of its liberties in this variable.

Tactical Search

The pattern matching and counting abilities described above are used in a cellular automata-like framework that should be adequate for computing many features relevant to Go. Each rule in a list is applied once in parallel over the whole board; this simple, uniform control structure allows relevant computations to be represented concisely. But without a more complex central control that can explore sequences of moves on isolated parts of the board, the representation lacks tactical search capabilities that are important to Go.

To help overcome this deficit, an explicit tactical search component is added to the representation. The combination of tactical search with parallel pattern matching gives a much more complete representation. Such an augmented representation may permit much of the computation that is needed to play Go well. For this experiment, we add only a limited, fixed tactical search capability; Section IV.C.7 discusses possible extensions.

Limited tactical search is used to precompute several point and string variables, as listed in Tables IV.1 and IV.2. The search has a single, simple form that provides exact results. Search attempts to capture a string with at most two liberties, by trying a move at each liberty. Such moves either capture the string, terminating search, or leave the string with a single liberty, forcing the opponent's response. Search terminates successfully when the string is captured, or gives up when a surrounding string is captured or the string gains more than two liberties. This search can be done very quickly, but it correctly handles ladders and common tactical situations near the edge of the board. When this search determines that a string is capturable, this is a non-heuristic result. Thus, precomputed features based on this search provide exact information rather than heuristic suggestions.

Search is done for each string already on the board, with either player to move first. These searches yield the four precomputed string variable listed in Table IV.2. When an enemy string is capturable, the capturing moves are specified with a precomputed point variable. Finally, each legal move for the current player is tried; moves that are capturable are specified with another precomputed point variable.

IV.C.5 Experimental Setup

Genetic Operators

Individuals are initialized with a number of rules chosen uniformly from 1 to 64. Genetic operators can modify list length, but individuals can have at most 64 rules; list length rapidly increases during evolution, so a limit is needed to keep memory requirements reasonable. To keep initial rules simple and make them likely to be used during computation, each rule is initialized with a single constraint. Genetic operators can modify the number of constraints in a rule, but a single rule can have at most 8 constraints.

Genetic operators should be able to add and delete entire rules, as well as add, delete, and modify portions of single rules. We use two crossover operators and a mutation operator to provide these forms of variation.

The mutation operator randomly chooses a rule from an individual, and changes a randomly chosen component of the rule to a randomly chosen new value. The rule component is chosen uniformly from the action weight, action variable, action initialization flag, countable constraint index, constraint position, constraint variable, constraint threshold, and constraint inequality type. When the chosen component lies within a constraint, a single constraint is chosen randomly.

The first form of crossover replaces a contiguous sequence of rules from one parent with a contiguous sequence of rules from the second parent. Sequence lengths are chosen independently, and are biased towards short lengths since this is likely to be a disruptive operator. The length bias is implemented as in Section IV.B.3. Length of a sequence starts at zero, and is incremented by one with 75% probability. This is done repeatedly, so that the probability of the sequence having length at least 1 is 75%, at least length 2 is 0.75^2, at least length 3 is 0.75^3, and so on. Sequence lengths are usually fairly short, but can occasionally become long. The lengths may be zero, meaning that simple insertions and deletions are possible consequences of this form of crossover.

Since rules may be large, and mutation only makes small changes that do not vary the number of constraints in a rule, we use an additional crossover operation to provide variation within rules. This second form of crossover "blends" a contiguous sequence of rules from one parent into a contiguous sequence of rules of identical length in the second parent. Since this is likely to be destructive, sequence length is biased towards short lengths as described above, starting from a minimum length of one rule and incrementing length with 50% probability. The sequences are blended rule-by-rule, with the first rule in the first parent's sequence blended into the first rule of the second parent's sequence, the second rule from the first parent's sequence blended into the second rule of the second parent's sequence, and so on. Rule blend chooses a random subset of constraints from the second parent's rule (note that constraints are unordered), and replaces it with a random subset of constraints from the first parent's rule.

Coevolutionary Setup

We coevolve strategies for 9x9 Go using the representation described above, with first-player strategies in the first population and second-player strategies in the second population. The coevolutionary setup used in this experiment is taken from

Section III.H; competitive fitness sharing, shared sampling, the hall of fame, the phantom parasite, and brood selection are all used.

Parameters were chosen on the basis of limited preliminary experiments. Parasite samples contain 25 hall of fame parasites, and 25 current parasites. Each population contains 512 members, with 10 elitists. Tournament selection is used, with each parent chosen in an independent tournament of size two. Offspring are created via the "replacement" form of crossover with 50% probability, and via the "blend" form of crossover with 50% probability. Mutation is applied to each offspring. Brood selection is important for this problem, since the complex representation and genetic operators are likely to result in many offspring that are either lethal or functionally identical to a parent. A brood size of 5 is used, with 10 parasites subsampled to test each brood member.

IV.C.6 Implementation and Results

Despite the limited computational capabilities of the strategy representation, competition outcomes are still expensive to obtain. Tactical search and rule matching were heavily optimized to reduce this computational burden. Since effects of moves are usually localized in Go, it makes sense to compute incremental updates when making moves and backtracking during tactical search. This yields significant savings over recomputing all strings, liberties, and other features after each small change to the board. Further savings are achieved by caching search results across moves; these do not have to be recomputed when new moves cannot affect their result.

Rule matching exploits bit parallelism. All point variables are encoded in a single word, and all string variables are encoded in a separate word. This permits all constraints at a single grid intersection to be checked simultaneously. Rule results are also cached: a map of matching conditions is saved after the execution of each rule. On the next move by the same ruleset, a rule's condition only needs to be rechecked at locations that are different than they were before execution of the rule on the previous move. Since most moves in Go have a small effect on the position, the points that need to be rechecked are usually a small subset of the entire board.

A parallel implementation was used, as described in Section III.H. This implementation gives the same results as a scalar implementation. Games are distributed among the processors at the start of each generation, and all results are collected before executing the GA, choosing new parasite samples, and beginning the next generation. Four hundred generations of coevolution were performed using 64 processors of a Cray T3D. This required nearly two processor-years of time to play over 40 million games of Go.

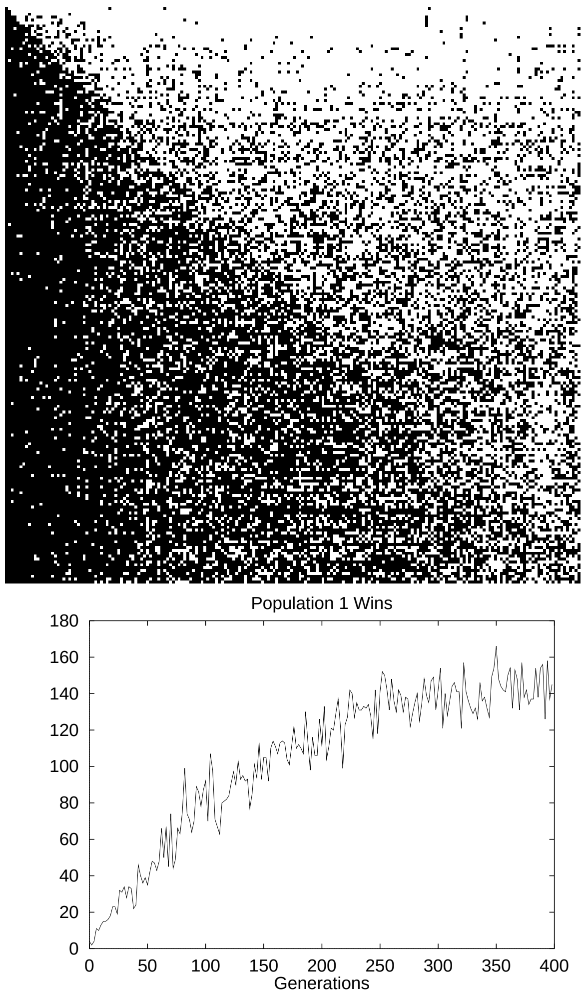


Figure IV.9: Arms Race in Go

Figure IV.9 shows the arms race for this run, giving tournament results between hall of fame first players (varying down the vertical axis) and second players (varying across the horizontal axis). Every other generation is displayed in the tournament, and a graph of total wins for first-population strategies is shown.

Arms race progress is fairly strong in the early part of the run, with new strategies arising that defeat a large fraction of prior strategies. Towards the end of the run, arms race progress begins to break down; single individuals no longer approach a capacity for defeating all prior opposition. The increase in first-player wins in the arms race slows significantly near the end. The run was halted when no further substantial gains in strategy performance were apparent.

Characterization of Evolved Strategies

In the first few generations, the best strategies are those that write any positive values into the output variable (as opposed to those that pass continuously from the first turn). During this brief period, moves tend to be identically evaluated, so that the fixed ordering over equal plays dominates; strategies place one stone after another in left-to-right, top-to-bottom fashion starting from the upper left corner. This blind strategy is unrelated to realistic Go play. The arms race begins with no ability to play the game.

In early generations after this point, strategies typically make little use of the computational capabilities available. Instead, evolved rules write only to the output variable, making moves based on simple local features. During this period, strategies learn to make use of precomputed features.

In later generations, strategies appear to perform more complex computations. Modifiable point and string variables begin to be used for intermediate results, and final output values at some points begin to depend on nonlocal features from outside the thirteen-cell neighborhoods of the points. Beyond these simple observations, however, it is difficult to understand the rather tangled computations being performed by evolved strategies. The easiest way to understand strategy behavior and evaluate performance is to observe games played by strategies.

A computer opponent is used to provide a fixed reference point for testing strategies. Evolved first-player strategies from the hall of fame were tested against the Swiss Explorer Go Program (version 3.3, 1995) [66, 67]. This is a relatively strong Go program that often achieves respectable performance in international computer Go competitions. While tests against it indicate that evolved strategies are not as robustly strong as Explorer, evolved strategies often do well in the opening and middle game, and are occasionally able to defeat Explorer.

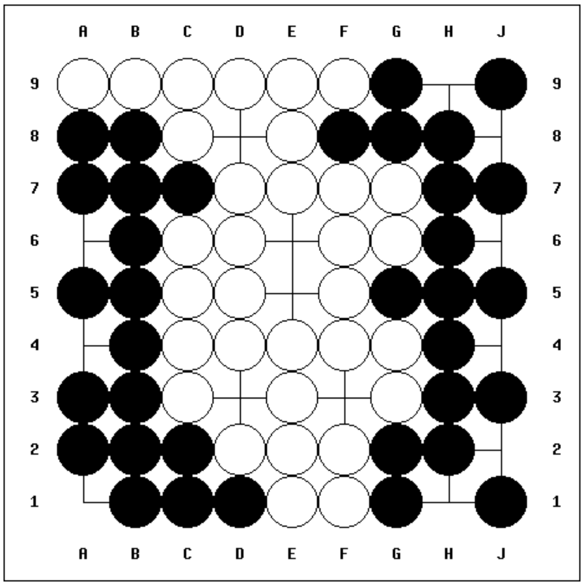


Figure IV.10: Generation 385 defeats Explorer

Figure IV.10 shows the final position of a game in which the hall of fame individual from generation 385 (playing black) defeated Explorer. Unlike many of the games played between evolved strategies and Explorer, neither side suffered major losses due to tactical blunders during this game. Thus, this game gives a fairly reliable display of the result of strategic competition between the two programs.

Of particular note in this game is the unusual strategy employed by black. Black attempts to secure edge territory by moving along the second line from the edge of the board. This is usually considered to be a poor strategy. Moves on the second line have little influence over the center of the board, making it difficult to keep all strings connected together. Also, moves on the second line do not take as much territory as moves on the third line. But this heuristic knowledge is not built into coevolved strategies, so that they are free to explore novel alternatives like this second line approach. These strategies demonstrate that the approach is not a terrible one; unless the second player responds aggressively on the second line to limit the size of black territorial gains, black's hold on the edge territory is very secure and results such as the one displayed here can occur. Such games can easily end in victory for black, or in close victories for white.

This unusual second-line strategy may arise from coevolution as a response to limitations in computational capabilities. Rather than adopting a more conventional strategic approach of creating larger areas of vague influence in the opening, moves on the second line secure territory very early. In addition, by playing stones close together on the second line, defense becomes easy; moves that close small gaps are used to respond to peeps (moves adjacent to gaps in territorial walls), and invasions on the second line are often tactically capturable.

While the above game demonstrates that evolved strategies can defeat Explorer, a more common result is a middle-game lead for the evolved strategy that is squandered in tactical blunders close to the end of the game. Figure IV.11 shows an extreme example of this, from a position near the end of a game between the hall of fame individual from generation 350 (playing black) and Explorer. Explorer has just captured with the move at E1; Black cannot immediately recapture at D1 due to Ko. To secure the safety of the four-stone black string adjacent to E1, black needs to defend (for example at H2). After making this move, white would fill at D1 and the game would be essentially over except for cleaning up dead stones; in this case, black would win. But the evolved strategy fails to make this defending move at H2, instead choosing a needless play at G9. White then captures the four-stone string and wins the game.

Given the precomputed tactical search features available to evolved strate-

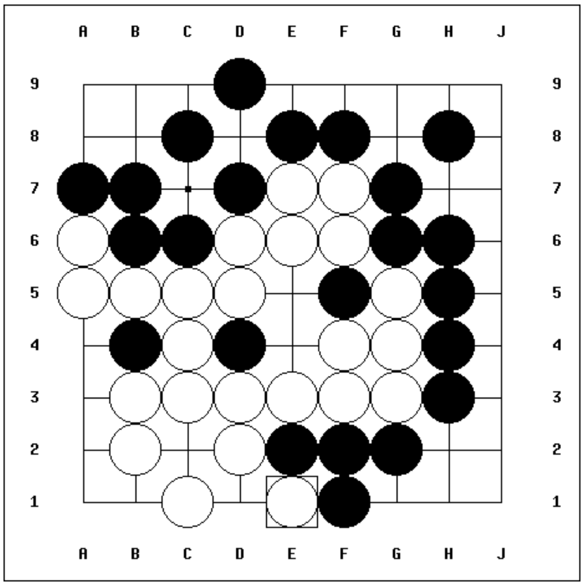


Figure IV.11: Generation 350 vs. Explorer

gies, it is easy to see the reason for this mistake. To evaluate the safety of the black string at F1, tactical search would try attacking white moves at H2 and G1. For black's response, tactical search would find that the attacking stone at E1 could be captured; once an attacking stone is captured, search gives up. In this particular case, the capture of the stone at E1 does not help save the four-stone string, but the limited tactical search cannot reveal this. Because of this limitation, the evolved strategy does not know of the importance of defending the string at F1. While black plays a respectable strategy that remains solidly ahead until the very end, this final tactical blunder on the last contentful move of the game results in loss.

The first-player population near the end of the coevolutionary run contains a more diverse set of individuals than just the second-line strategies shown above. During coevolution, the second population responds to second-line strategies by making its own moves on the second line. Such moves halt territorial advances by first players, allowing the defeat of first-player second-line strategies. The first population retains a niche that uses more conventional strategies to defeat these second-player second-line

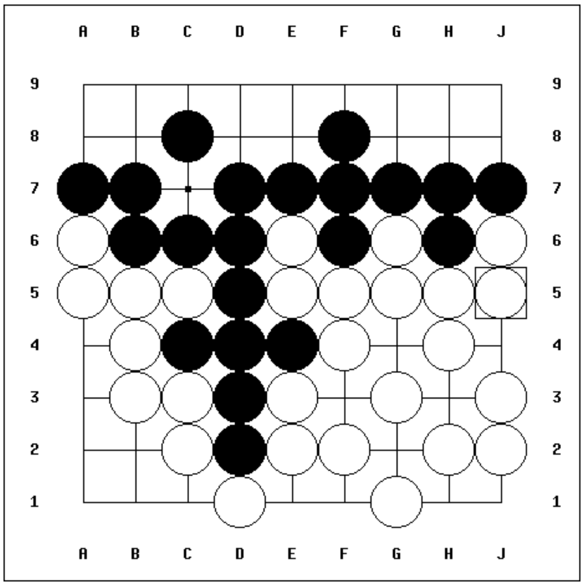


Figure IV.12: Generation 378 vs. Explorer

responses.

The outcome of one of these alternative first-player strategies against Explorer is shown in Figure IV.12, after the final meaningful move of the game. The evolved strategy is taken from the hall of fame in generation 378, and plays black in this game. The evolved strategy makes its first move at D6; this focus on the center is generally considered superior to early moves on the second line. The game shown in Figure IV.12 proceeds without major tactical losses by either side, until the very end of the game when black fails to save a small string on the right side of the board. Much as in the game shown in Figure IV.11, this results in a narrow loss for black in a game that would have been a black victory if a final defensive move had been made (although it seems that the mistake here is simpler and should have been more easily avoidable than the mistake in Figure IV.11).

Measures of Useful Complexity

In an open-ended arms race, individuals must become more complex to achieve higher levels of performance as the limitations of simple individuals are reached. To study this emergence of complexity in the coevolutionary Go experiment, we need a concrete measure of strategy complexity.

Simple measures of complexity may only capture superficial features of strategies, rather than truly *useful* complexity [54]. For example, one measure of complexity that seems appropriate in this representation is the number of rules in a strategy. But rules do not need to be used during computation, so a large number of rules may be present in a strategy without many being used. Unused "junk" rules may affect the evolution of strategies [99], but their presence does not reveal increased computational sophistication.

To provide more accurate measures of computational sophistication, we first decide which rules are likely to impact the actual moves made by strategies during play. While we do not have a perfect method for deciding which rules are important in this way, we have developed a heuristic that seems to work well in practice.

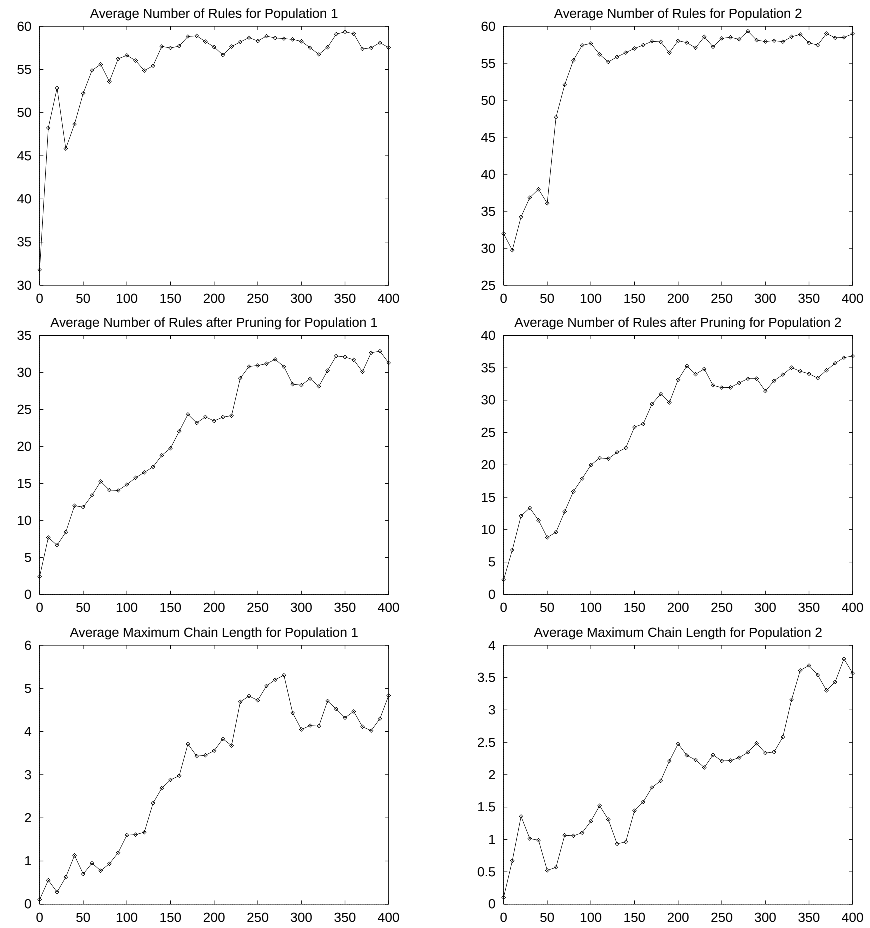
To reduce a rule list to those rules that are important to computation, rules are pruned from a strategy via static analysis if they cannot impact the final contents of the output variable anywhere on the board. In particular, rules with zero output weight, rules that rely on a positive modifiable variable that does not receive positive input from any previous rule in the list, and rules that output to a variable that is never used (at least before it is reinitialized by a later rule) are all removed. These removals are repeated until no further rules can be pruned.

While this will not remove all useless rules, it does seem quite successful in practice. For example, most rule lists in the initial, randomly generated population have most rules removed by this procedure, leaving just a handful of essential rules. Also, other methods considered for deciding which rules are useful seem no less approximate. For example, we might choose rules with conditions that successfully match during actual play as "useful rules," but such matching rules may never affect the output of the computation. As another example, we might try removing each rule from the list to see if behavior is different without it. But the presence of redundant duplicate (or near-duplicate) rules could make *all* of the duplicates appear to have no impact on output by this measure. Our method based on static analysis has the advantage of being computationally efficient, so that measures based on it can be averaged over an entire population.

The size of the pruned rule list should provide a more accurate measure of the useful complexity of a strategy. One potential problem arises because the "replacement" form of crossover used here makes it easy to duplicate rules. Duplication can be used to make changes to a strategy, and it can give strategies some redundancy to survive destructive genetic operators. By themselves, however, such duplicate rules do not represent substantial increases in computational sophistication. We desire a measure that is not affected by redundant duplicate rules.

When intermediate results are written by one rule and used by a later rule, we have a computational chain with more structure than two rules acting independently. Longer chains should represent more sophisticated computations that take more information into account before finally writing a result to the output variable. Such chains are the only method available for computing a point's output variable using information that is not available in the local neighborhood of that point. We define a chain as a sequence of rules (not necessarily contiguous) in a rule list, such that each rule depends on a modifiable variable written by the previous rule. The length of a chain is the number of rules in it, not counting the last rule; if no rules depend on prior results, chain length is zero. By considering chain length within the pruned rule list, we mostly ignore chains that compute intermediate results unused for eventual output. We use the maximum chain length in a pruned rule list as another measure of its computational sophistication.

Figure IV.13 shows these three measures of complexity for the coevolutionary Go run. For each population and each measure, the average across the entire population is plotted every ten generations.



(a) Population 1 **(b)** Population 2

Figure IV.13: Measures of Complexity in Go

The number of rules in individuals rapidly rises to a point near its maximum value after only about 100 generations in the second population, and after a bit longer than this in the first population. In the first 100 generations, most rules in these long lists are junk rules that are unused by computation. These junk rules may serve an evolutionary role in protecting other, important rules from destruction by genetic operators [99], but they do not reveal computational sophistication.

The number of pruned rules in each population displays a gradual rise over the first 200 generations. This is probably indicative of a genuine increase in computational capabilities, although some of this increase may be due to the presence of more near-duplicate rules. Little rise in complexity is seen during the last 100 generations, when coevolution is largely stalled.

The average chain length also increases during the run. In the first few generations, rules used during the computation directly affect the output variable, so that substantial chains are largely nonexistent. Later in the run, chains of significant length become common. Part of the rise in chain length is probably due to rules that read the output variable and also write it, but there also exist long chains that use other modifiable string and point variables. The rise in chain length seems to signify an increase in structured computation performed by strategies.

IV.C.7 Discussion

Coevolution in Open-Ended Domains

Results from the coevolution of Go strategies are promising in a general context of open-ended performance gains. While falling short of the Holy Grail of ever-increasing performance and useful complexity, the experiment displays several important features of coevolution in an open-ended domain.

In an open-ended arms race, complexity of individuals necessarily increases as the performance of individuals with limited complexity becomes inadequate. Figure IV.13 shows that several measures of useful complexity undergo a protracted rise in both populations during the coevolutionary Go experiment. Since the complexity we measure arises from computational structures that are unlikely to be built by chance, selective pressures are building this useful complexity, as would be expected in an open-ended arms race.

Coevolution eventually stalls in this experiment, but this seems to be due largely to the intrinsic limitations of the representation, and the difficulty of evolving significantly stronger robust strategies in it. Against novel opponents of reasonable ability (Explorer in the examples given above), evolved individuals display organized strategic competence in the opening and middle game. Losses may often be traced directly to the intrinsic limitations of the tactical capabilities given to strategies. This suggests that the arms race is successful in pushing strategies to limits of their computational capabilities.

To help test this hypothesis, further evolution was done on the final coevolved population of first-player strategies. In the latter part of the coevolutionary run, individual strategies lose the capability to cover most opposition from the hall of fame; we wish to test whether this lack of individual robustness is due to coevolutionary dynamics or representational limitations. The first population at generation 400 was used to start a single-population GA against the entire second-player hall of fame. Each generation, 100 second-player strategies were sampled from the fixed 400 in the hall of fame; these 100 were used as "parasites" to test the evolving population of first-player strategies. Other parameters from the coevolutionary run remained the same. Over 65 generations, very little rise was seen in fitness. Individual strategies never approached the ability to cover the entire second-player hall of fame. It appears that significant gains in robustness at the end of this coevolutionary run are not possible, due to limitations of the representation and the difficulty of evolving strategies using it. This experiment helps support the conclusion that the arms race pushed strategies to the reasonable limits of their representational and evolutionary capabilities. Strategies with expanded representational capabilities might be more likely to produce an open-ended arms race.

Coevolving a Performance Program?

The programs produced by these coevolutionary experiments are not high-performance Go playing programs. The competing factors discussed in Section IV.C.4 place limits on what we can hope for from coevolution given present-day computing resources. Here we discuss extensions to coevolutionary Go that will become possible once more computational power is available, and contributions to performance programs that may be possible in the short term.

The most obvious idea is to base coevolved strategies on an existing performance program. Such experiments are difficult to perform as a part of a research effort, because source code for performance programs is not usually publically available.[5] Although the computational expense of strong strategies currently prohibits the use of coevolution for the production of substantial novelty in this context, coevolution could provide a strong method for tuning elements such as weights in evaluation functions and search pruning heuristics. Although this would be unlikely to yield protracted arms races, it could give useful results with short coevolutionary runs.

Eventual increases in computing power may make it possible to incorporate features from top-performing Go programs into experiments, like those described here, capable of producing significant additional novelty. There currently exists a large gap between the great computational power commonly available, and the small amount of computation required by the top-performing Go programs. In contrast to the success of brute-force approaches to chess, extension of Go programs with additional blind search is rarely useful. As available computational power increases, the gap between the time required for performance, and the time that could be spent computing, is growing. If this continues, it should eventually be possible to base coevolutionary experiments on the strongest conventional methods available.

During coevolution, complex and subtle changes to strategies can be effectively evaluated. This may allow the coevolution of strategies that are more complex than human designers are currently comfortable with. For example, we may place the focus of tactical search under the control of evolved strategies. Such strategies could smoothly integrate tactical search with heuristic evaluations in ways that have not yet been possible in conventional Go programs. This integration seems vital to human play, and it may also be important for performance in computer Go.

Beyond tactical search control, additional features of the representation should be opened up to evolutionary control. For example, while it is useful to

[5]This is due to the software market for strong Go programs in Asia, and to the existence of a million dollar prize for a professional-strength program.

treat strings as units for some purposes, other structures should be handled as a unit as well in some cases. Groups of multiple strings, areas of territory and influence, and regions that impact search outcome all have features that would best be computed for the entire structure at once. Rather than attempt to define these concepts (which do not have accepted exact definitions, for the most part), the very definition of the structures treated as a unit by strategies could be placed under evolutionary control.

Since there exists a gap between the time required by current performance programs, and the computational power available, an important question is how to exploit the available computational power to improve Go play. One factor hampering the utility of extensive search is that it cannot penetrate very deeply into the large search tree. Since accurate evaluation of resulting positions is difficult (especially if some are at different depths), and comparison of similar positions is even more difficult, shallow search is likely to be ineffective. It would be preferable to generalize from a limited set of future lines of play, to guide further search and improve the accuracy of a heuristic result. It may be possible to achieve this by using coevolution in real-time. When viewed as a tree search method, coevolution evaluates alternative moves near the root of a search tree by generalizing from the outcomes of complete games played down promising branches of the tree. The coevolutionary experiment described above only does this for a tree rooted at the initial empty board, but it may be possible to start coevolution from an arbitrary starting position to produce strategies that are specialized for it. If this coevolution is done in real-time during play, moves can be selected by these specialists. Coevolution can begin again (seeded with strategies from the current position) at the next move to be made. Since variation among possible outcomes is limited once a game's position has significant structure, even computationally limited strategies may be useful for this purpose.

IV.D Designing Drugs to Overcome Resistance

Drug resistance is a major obstacle to the effectiveness of many antiviral drugs. The current drug design methodology can address this problem after resistance emerges by targeting new designs to resistant forms of a virus. The virus, however, may respond to the new design with previously unseen forms of resistance.

By using computational models of drug interaction with the virus, we may treat drug design as an adversarial problem. We can anticipate and account for resistant mutants during the drug design process, and search for drugs that overcome this resistance.

Here, we focus on HIV protease inhibitors. The HIV protease enzyme[6] serves an essential role during HIV replication by cleaving polyproteins manufactured from viral genetic material into the individual proteins required by HIV. HIV protease is a homodimer, meaning that it is composed of two identical proteins. Chemical activity takes place at a localized active site in the protease. Protease inhibitors bind to this active site, preventing the normal function of HIV protease and interrupting virus replication.

The emergence of resistance limits the effectiveness of such drugs. Mutants of HIV protease that are resistant to an inhibitor have a modified shape that prevents binding by the inhibitor, while retaining the required cleavage ability of the protease. In clinical tests, mutants have arisen that are cross-resistant to a panel of inhibitors [27].

This application depends on a computational model that can predict whether or not a given inhibitor will be effective against a particular mutant form of HIV protease. For example, molecular docking software predicts geometry and energy of binding between molecules [65]. This could be used to evaluate the binding of an inhibitor to a particular mutant.

[6]Proteins are long sequences of amino acids that fold into a specific 3-D shape. Enzymes are composed of proteins, and have a chemical function dependent on this shape.

Given such a model, coevolutionary methods can be used to design robust inhibitors that bind to all feasible protease mutants. Since HIV protease serves a required function in HIV replication, feasible mutants must maintain this functionality. This may be checked by using the model to evaluate the binding of the mutant enzyme to the natural substrates that it must cleave. If this binding ability is retained, then the mutant will still cleave these required natural substrates. Feasible protease mutants are those that are shown by this binding test to retain their chemical activity.

Inhibitors range over some combinatorial class of easily synthesized molecules. Most existing protease inhibitors are "peptidemimetics," i.e. they chemically resemble a peptide,[7], with the bond that would normally be cleaved by protease replaced with an uncleavable bond. This class of inhibitors has a simple representation as a sequence of amino acids; experiments described here use this representation.

Several aspects of this adversarial problem make it difficult to approach. One problem is the construction of a model with sufficient accuracy and speed; molecular docking tools typically obtain accuracy at the expense of speed. Another difficulty is the determination of 3-D protease structure. The structure for the native HIV's protease is known,[8] but mutations to the protease may have complex global impacts on its structure. Predicting these changes computationally is a very difficult problem.

As an easier first step, we consider a fast, simplified model of binding and protease activity. We also restrict protease mutations to those in the active site of the protease, where the chemical activity is localized. These simplifications allow use of the theoretically complete coevolutionary method described in Section II.D. The exact results obtainable with this approach are a desirable feature of initial experiments in this domain. Such results may give some idea as to the existence of perfect inhibitors, the size of the space of active mutants, and specification number for this problem. This should help design methods for obtaining more accurate results on this problem.

[7]A peptide is molecule made up of a short sequence of amino acids.

[8]Native refers to the unmutated form.

The simplified model should capture some aspects of HIV protease binding, but it is still very approximate and resulting inhibitor designs are not yet expected to be practically useful. Therefore, we omit chemical details below and focus on the form of the model and the structure of the competition. Work described here is the first step in research that will continue with more accurate models; several goals for future work are discussed below in Section IV.D.5.

IV.D.1 Simple Model

The simplified model of interaction was developed in collaboration with David Goodsell of the Scripps Research Institute. It is based on simple properties of the geometry of the HIV protease active site. Inhibitor binding is evaluated using the idea that molecules tend to bind when they have complementary shapes that fit together well.

Most existing protease inhibitors are peptidemimetics [26]. We also consider this class of inhibitors, represented as hexapeptides (sequences of six amino acids). When bound to HIV protease, the position of such inhibitors tends to be well-conserved. Each of the six amino acids fits into a roughly defined pocket in the active site of the protease. In our model, each of these pockets is a subsite defined by the set of protease amino acids that contact the inhibitor amino acid in that pocket.[9] The protease amino acids contacting these six subsites are allowed to mutate. Figure IV.14 shows a schematic of a hexapeptide inhibitor in the active site, with representative protease amino acids along one of the subsites.

Inhibitors and mutants are evaluated with a simple volume-based geometric match. As detailed below, we obtain an "average volume" of the contents of each subsite for the substrates that are known to bind with the native protease [21]. We then penalize deviations from this average volume. This is a simple evaluation of geometric fit. If the contents of a subsite are much larger than the average volume,

[9] "Contact" is defined here to mean that Beta carbons are within 6 angstroms.

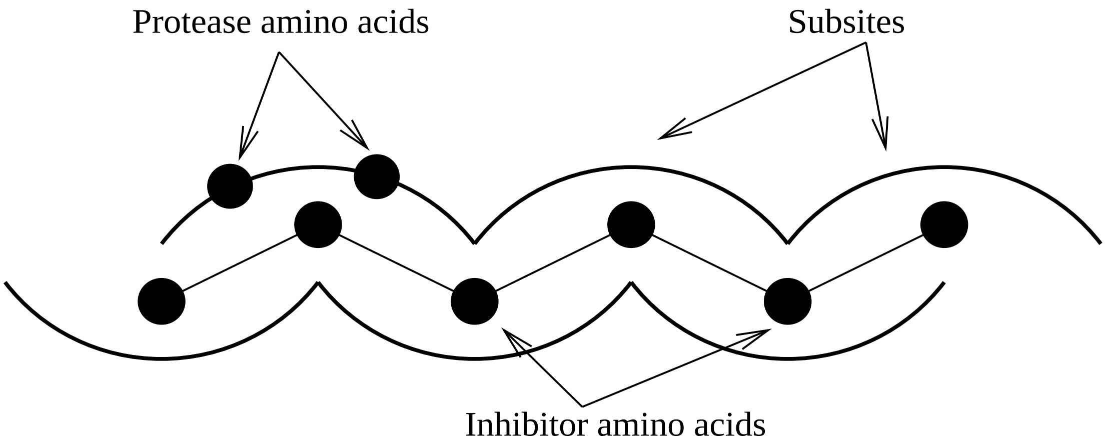


Figure IV.14: Hexapeptide in Active Site

there is likely to be an unfavorable overlap among the contents. If, on the other hand, the contents of the subsite are much smaller than the average volume, then there is a large gap between the inhibitor and protease so that they do not closely fit.

The van der Waals sidechain volume of each amino acid is used in volume computations. The "subsite volume" is the sum of the volumes for the inhibitor amino acid in that subsite, and each of the protease amino acids in that subsite. The "average volume" in a subsite is the average of the subsite volume over the known substrates, bound to the native protease. The scale of important volume deviations in a subsite is set by the range of natural variation of volume in that subsite: the standard deviation of the average volume is also computed for each subsite, and is used to normalize the magnitude of volume deviations.

When evaluating the binding of a particular inhibitor to a particular mutant, a penalty is summed over all subsites. In a single subsite, the penalty is

$$\frac{(\text{subsite volume} - \text{average subsite volume})^2}{(\text{subsite standard deviation})^2}$$

The sum of the six subsite penalties for interaction between an inhibitor and protease will be called the "binding penalty."

Mutants need to be constrained to those that retain their ability to bind to the natural substrates. To evaluate this in a simple way that is appropriate to the model, we evaluate binding to the average substrate. In each subsite, a penalty is computed as

$$\frac{(\sum \text{protease sidechain volumes in subsite} - \sum \text{native sidechain volumes in subsite})^2}{(\text{subsite standard deviation})^2}$$

This is summed over all subsites to obtain an "activity penalty."

We (somewhat arbitrarily) dictate that volumes may deviate by one standard deviation per subsite on average: a threshold of 6 is set for activity penalty. Mutants with activity penalty at most 6 are deemed active, and those with penalty above 6 are deemed inactive and are not allowed.

The success of an inhibitor against a particular mutant is evaluated relative to that mutant's activity penalty. If activity penalty is low, the mutant is very active and a tightly-binding inhibitor is required. This is relaxed somewhat if activity penalty is higher. We set a threshold on the allowed difference of binding penalty − activity penalty. A variety of choices for this threshold are discussed below.

To retain some chemical realism, protease mutations are restricted to amino acids that are chemically similar (mainly according to charge) to the corresponding native amino acids. There are 15 possible choices at protease sites which have an uncharged amino acid in native HIV, and 4 possible choices at protease sites which have a charged amino acid in native HIV.[10] Applying a similar restriction, inhibitors are limited to those composed of 15 possible amino acids.[11]

IV.D.2 Model Properties

To see whether this approximate model has any ability to predict binding, we consider hexapeptides for which cleavage by native protease has been experimentally determined. For these peptides, cleavage is closely related to binding. We compute the

[10] Protease sites that have an uncharged amino acid in native HIV are allowed to mutate to any of the uncharged amino acids, except histidine. Protease sites that have aspartate in native HIV are allowed to mutate to glutamate, asparagine, or glutamine; this gives a total of 4 choices at these locations. The catalytic residue is not allowed to mutate.

[11] Any of the uncharged amino acids, except histidine.

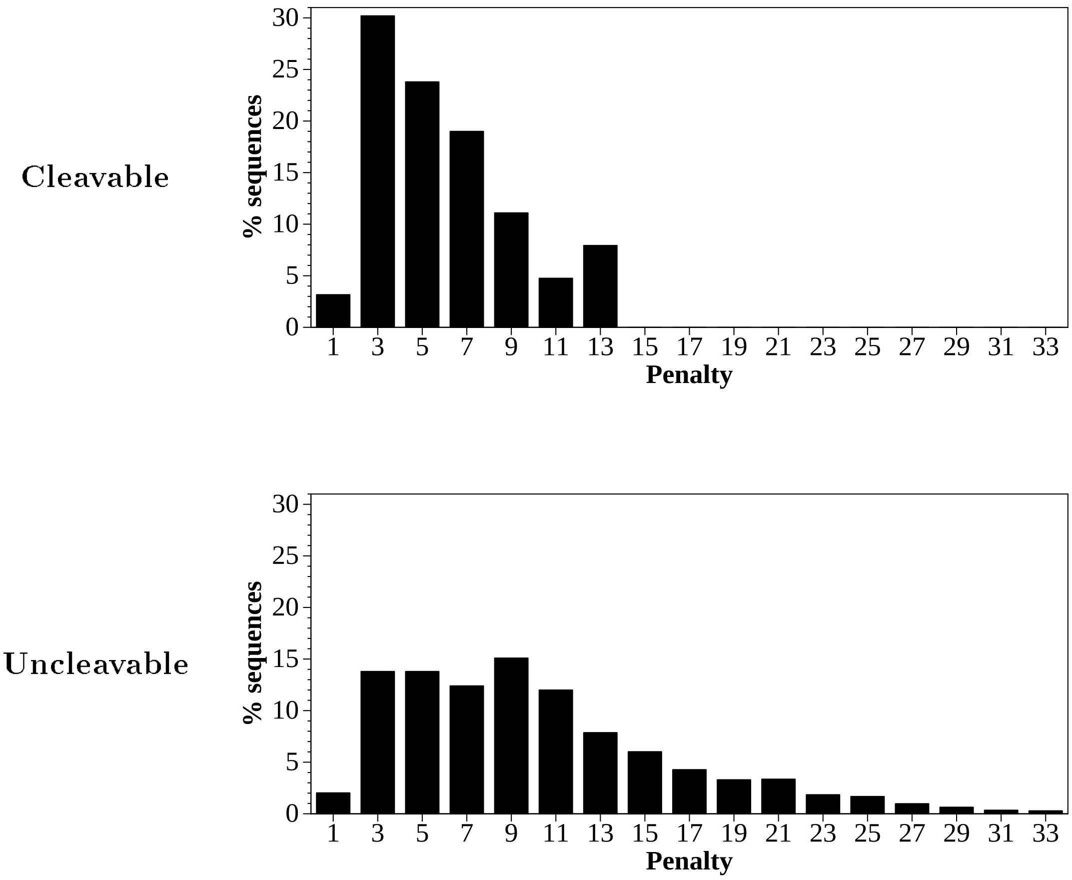


Figure IV.15: Binding Penalties for Known Hexapeptides

binding penalty for these sequences against the native protease. For 63 substrates that the protease is known to cleave, the average binding penalty is 6.00, with a standard deviation around the average of 3.35.[12] For 1727 hexapeptides that are not cleaved by the protease, average binding penalty is 10.23, with a standard deviation of 6.39. Figure refhivhistograms shows the distribution of penalties for these cleavable and uncleavable peptides. Each bin in the histograms covers a width of 2 along the penalty axis, with the first bin starting at a penalty of zero. These results show that even this simple model has some ability to separate sequences that bind from those

[12]These are dimensionless quantities, since we normalize squared volume change by squared standard deviation of volume.

that do not.

There are ten relevant mutable amino acids that affect the active site. Since the protease is a homodimer, a mutation to any one of these ten amino acids affects both proteins in the enzyme, changing the active site of the protease in two places. With the restrictions on mutations mentioned above, there are 15 possible amino acids at 8 of the mutable locations, and 4 possible amino acids at 2 of the mutable locations, for a total of about 41 billion possible protease mutants. Of these, about 250 million, or 0.6%, are active according to an activity penalty threshold of 6. This fairly high percentage indicates that, for heuristic search purposes, it should be feasible to take the simple approach of imposing the activity constraint *after* generating arbitrary protease mutants.

There are far fewer inhibitors than protease mutants. With 15 possible amino acids at each of the six locations in the hexapeptide, there are about 11 million possible inhibitors. Setting a threshold of 2 on allowable difference between activity penalty and binding penalty for a successful inhibitor, there are 46960 inhibitors that bind to the native substrate (which has an activity penalty of 0, by definition).

IV.D.3 Methods

The simple form of this model allows us to define exact strategy learning algorithms and use the covering competitive algorithm (as defined in Section II.D) to solve the problem. Recall that a strategy learning algorithm takes an explicitly defined set of opponents, and finds individuals that defeat all of them. In this problem, one strategy learning algorithm finds inhibitors that are effective against a given set of mutants, and another finds mutants that are resistant to a given set of inhibitors. The covering competitive algorithm then uses these repeatedly to construct progressively stronger inhibitors and mutants. Each step, it finds an inhibitor that defeats all mutants from prior steps, and a small set of mutants that together defeat all inhibitors from prior steps. It is guaranteed to eventually find a perfect inhibitor (effective against all possible resistant mutants) if one exists, or prove that none exists.

Recall that a teaching set is a set of test cases, such that at least one exposes a flaw in each imperfect solution. In this problem, a teaching set would consist of a set of mutants, with at least one resistant to each possible inhibitor that is not completely robust. Specification number was defined to be the size of the minimal teaching set; covering competitive algorithm performance depends on specification number. These experiments should give some idea of the specification number for this problem.

Strategy learning algorithms for finding inhibitors and protease mutants do a complete search of possible solutions. Bounds on penalties are used to discard infeasible partial solutions; this allows massive pruning that makes the search practical. Though these algorithms would not be polynomial-time in the worst case as the number of subsites grows, they are fast enough to be practical for the particular model described above. This practical problem-specific efficiency is probably the best we can hope for; even simple versions of these strategy learning problems are instances of the *maximum satisfying linear subsystem* problem, which is NP-complete and hard to approximate [7].

We describe the strategy learning algorithms below. Since they are highly specific to this particular model, we do not provide extensive details.

Finding Inhibitors

In this model, the design of an inhibitor effective against a *single* mutant is very fast and easy. Against a fixed mutant, binding penalty contributions from each subsite can be minimized independently of one another. In each subsite, we select the inhibitor amino acid with the lowest binding penalty against the mutant. Combining the amino acids that are best in each of the six subsites yields the hexapeptide with lowest binding penalty against the fixed mutant.

Against a larger panel of mutants, inhibitor design is not so simple. In each inhibitor subsite, different amino acids are optimal against each mutant. Choices among these must be balanced to create an inhibitor with adequately low binding penalty against all given mutants.

Consider the subsites to be numbered from 1 to 6. The inhibitor is built up starting from subsite 1, and proceeding through the subsites one by one, backtracking when necessary.[13] While working on subsite i, subsites numbered less than i are completed while those numbered greater than i are not yet fixed. Each possible amino acid is tried at subsite i. To check feasibility of each possibility, a best-case completion is computed individually against each mutant as described above. If all of these completions have acceptable binding penalty, search proceeds to subsite $i+1$ (or terminates successfully, if i is the last subsite). If some completion has an unacceptable binding penalty, then no possible completion of the partial inhibitor will work, so it is rejected. In this case, search tries the next amino acid at subsite i, backtracking to subsite $i-1$ (or terminating with failure, if i is the first subsite) if all amino acids have been exhausted at i.

Most unworkable partial inhibitors are shown infeasible after a few subsites are built. This pruning makes the algorithm very fast, requiring only a few seconds of computation time on a 100 MHz PowerPC processor.

Finding Protease Mutants

The algorithm for finding mutants is similar but much slower, due to the activity penalty constraint, the larger number of mutable protease amino acids, and the nonlinear interactions between them. The nonlinear interactions arise because subsite penalties depend nonlinearly on the protease amino acids in the subsite.

Search builds partial protease mutants, as described for inhibitors. Subsites (which include multiple protease amino acids) are handled as a unit, since protease amino acid choices within a subsite interact nonlinearly. Since each subsite includes a large number of possible combinations of protease amino acid choices, search is

[13]Actually, to help avoid long transitive chains, a random ordering is chosen for the subsites on each execution of the inhibitor-finding algorithm. While this may not completely satisfy the (p,q)-randomization criterion (Section II.D), it does seem to improve the total time for the covering competitive algorithm.

expensive. Fortunately, interactions between subsites do not contribute to this non-linear complexity. Only a few protease amino acids are shared between subsites. We choose an ordering for the subsites that contains all shared protease amino acids in the *first* subsite of the ordering. Once search makes a choice of protease amino acids in this first subsite, shared protease amino acids are all determined and therefore do not affect search.

Each time a subsite is completed, we check for feasible protease completions (separately against each inhibitor) to see if search may be pruned. This check for feasible completions bounds the binding penalty of the best-case completion against each inhibitor, while meeting the activity constraint. To do this, activity penalty is discretized into several levels, and a table is constructed giving the best binding penalty at each activity penalty level. This table is built at successive subsites using dynamic programming. The partial protease mutant is rejected if there does not exist an activity level that allows each inhibitor-specific completion to successfully escape binding by its inhibitor.

Since specification number may be larger than one, the algorithm needs to be able to find sets of mutants that cover all prior inhibitors. A greedy set-covering approximation[22] is used for this purpose: new mutants are chosen to cover as many remaining uncovered inhibitors as possible. This proceeds until all inhibitors are covered.

This algorithm is computationally expensive, often requiring several hours on a 100MHz PowerPC processor. This is still adequately fast to execute the covering competitive algorithm.

Algorithm Summary

The following summarizes the use of the inhibitor-finding algorithm and mutant-finding algorithm described above in the covering competitive algorithm.

```
Initialize inhibitor set I to the empty set
Initialize protease mutant set M to the native protease

/* Loop exits with return of perfect inhibitor, or failure if none exists.  */
Do:
    Call inhibitor-finder on M
    If no inhibitor covers M:
        Return failure /* No perfect inhibitor exists */
    Else:  Add returned inhibitor to I

    /* Greedy set-cover of inhibitors.  Track uncovered inhibitors with I'. */
    Initialize I' to I
    Do while I' is not empty:
        /* Mutant-finder covers as much of I' as possible with one mutant.  */
        Call mutant-finder on I'
        If no member of I' can be defeated:
            Return I' /* I' consists of a perfect inhibitor */
        Else:
            Remove from I' inhibitors defeated by returned mutant
            Add returned mutant to M
```

IV.D.4 Simple Model Results

Several choices were explored for the threshold on difference between activity penalty and binding penalty, to study the structure of the competition within the model. Limits on a reasonable range of variation for this threshold can be established before running the covering competitive algorithm.

At a threshold of 0, effective inhibitors must have binding penalty no greater than activity penalty. Since the native protease has an activity penalty of 0, and no inhibitor can exactly match the average subsite volumes to attain a binding penalty of 0, the native protease is resistant to all inhibitors at this (unrealistic) choice of threshold.

Inhibitors that are close to average subsite volumes should do well in this model, since active mutants must have a low penalty against the average substrate. By taking the amino acid with volume closest to the average volume in each subsite, we obtain an inhibitor that is as close to average as possible. At a threshold of 1.125 on the penalty difference, the mutant-finding algorithm shows that no mutants are resistant to this inhibitor; it is perfect at this threshold (and for larger thresholds).

At some point between this threshold and 1.0625, this average inhibitor loses its perfection and acquires a resistant mutant; there may still be a perfect inhibitor at this point, but it can no longer be constructed by taking the best approximation of the average volumes.

Thresh	**Steps**	**Last Mut**	**Tot Mut**	**Perfect Inhib?**	**Perfect Mut?**
0	-	-	-	No	Yes
0.5	11	2	13	No	No
0.75	10	2	14	No	No
1.0	11	3	22	No	No
1.03125	11	2	18	Yes	No
1.0625	8	2	12	Yes	No
1.125	-	-	-	Yes	No

Table IV.3: Covering Competitive Algorithm Results

Between thresholds of 0 and 1.125, we use the covering competitive algorithm to determine whether perfect inhibitors exist. Table IV.3 summarizes results of running the covering competitive algorithm at various threshold. The boundary cases discussed above are also included. "Steps" gives the number of steps in the execution of the covering competitive algorithm before a final result (positive or negative) is obtained; note that one step includes finding both an inhibitor and a set of protease mutants. "Last Mut" gives the number of mutants required to cover all prior inhibitors on the last step of the covering competitive algorithm. "Tot Mut" gives the total number of mutants found during execution of the covering competitive algorithm.

At a threshold of 1.0 the covering competitive algorithm proceeds for 11 steps. For the last 3 steps, three mutants are required to cover the inhibitors found - no single mutant is resistant to all inhibitors. On the last step, no single inhibitor covers all prior mutants. At this threshold, there are resistance mutations effective against any inhibitor.

Since there is no inhibitor with binding penalty within 1.0 of activity penalty for all mutants, perfect inhibitors cannot exist at smaller thresholds, either. Run at thresholds of 0.5 and 0.75 showed that neither perfectly resistant mutants nor perfect inhibitors exist at these points. For very small thresholds (for example at a threshold of 0 as described above), there will be a single mutant resistant to all inhibitors. Such a perfect mutant may be difficult to find. With small thresholds, most mutants will have only a few tightly-binding inhibitors successful against them. These inhibitors bind to very few mutants, so large sets of them are required to cover large sets of mutants. This suggests that specification number can get large for the inverse problem of looking for perfect mutants. Fortunately, this does not affect the difficulty of finding robust inhibitors.

At a threshold of 1.03125, the covering competitive algorithm finds a perfect inhibitor in 11 steps. The inhibitor found by this run is also perfect for thresholds larger than 1.03125, of course. This threshold is close to the point where perfect inhibitors first come into existence. Only 21 inhibitors defeat all of the 18 mutants found during the run at this threshold. All of these, except for the single perfect inhibitor found during the run, were shown to be imperfect by use of the mutant finding algorithm against this remaining set of 20 inhibitors. Including the three extra mutants required to show the imperfection of these 20 inhibitors, we get a set of 21 mutants that form a teaching set; only the single perfect inhibitor defeats all of these 21 mutants. This places an upper bound of 21 on specification number at this threshold.

This model is too approximate to expect chemically realistic results, but the results obtained with the covering competitive algorithm are encouraging. Given a model of inhibitor binding and protease activity, the covering competitive algorithm is able to determine whether robust inhibitors exist, and find them if they do exist. It does this in a very small number of steps while covering inhibitors with small sets of mutants. This, and the upper bound of 21 on specification number at the lowest threshold where a perfect inhibitor was found, indicate that issues of teaching set size are not a major difficulty under this model. If the model can be tuned for accuracy, exact results should still be obtainable, and would be more realistic. Even if a different, more accurate model is substituted, enough of the structure of the competition may survive for the problem to be tractable for coevolution.

IV.D.5 Future Work and Long-Term Possibilities

Simple models such as the one described here have a major advantage, which is that they permit exact results to be obtained with the theoretically complete methods from Chapter II. If a perfect inhibitor exists, we can find it and prove it is perfect (within the limitations and accuracy of the model). If no perfect inhibitor exists, we can exactly prove this as well. Since any model of this complicated biological system is bound to be approximate, it is very desirable to solve the adversarial problem exactly to avoid additional approximation error. Exact strategy learning algorithms will not exist for more complex models, and we will have to resort to approximate coevolutionary heuristics.

We may be able to use simplified models to study general questions about resistance and inhibitor design in the abstract, without obtaining results that are specifically accurate for any particular system. As an example, consider the form of successful inhibitors in the model described above. They are likely to resemble the average substrate, since the activity constraint requires protease mutants to have a low penalty for this average substrate. This feature comes from the simple form of the model. We can consider an improved model that evaluates activity against several natural substrates independently, rather than aggregating them into an average. Even under this improved model, the "average" of these natural substrates may be a good inhibitor. Alternatively, inhibitors that closely mimic just one of the natural substrates may work well. This is an important issue for enzymes like HIV protease that bind to a large, complex set of substrates. We plan to use an extended simple model to consider, abstractly, the question of which type of inhibitors are best for such enzymes.

Since experimental data is available for inhibitor and substrate binding to

HIV protease, it is possible to tune parameters to fit the simple model to this data. By doing this, we will be able to obtain results with much more chemical realism, while retaining the ability to find exact solutions to the adversarial problem.

Beyond the simple model, we plan to use molecular docking software to obtain more accurate binding evaluations for use in competitive coevolution. Autodock [65] is an accurate molecular docking tool developed in Arthur Olson's group at the Scripps Research Institute. This software calculates several energy terms to evaluate the energy of a candidate docking. It assumes a fixed protein position, and uses a genetic algorithm-local search hybrid to optimize the position and internal rotation angles of a small drug molecule [83]. Extensive development of Autodock has produced a system that has been shown to make accurate predictions of both binding strength and geometry in systems like HIV protease [65]. Although it is currently somewhat slow to use for all competition evaluations during competitive coevolution (a single docking run takes about five minutes on a contemporary workstation), problem-specific optimizations may be possible. For example, the inhibitor must bind to the active site for results to be meaningful. Rather than do a full global search, we may be able to obtain accurate results by placing the inhibitor in the active site and doing a more restricted local search during molecular docking. Since many similar dockings will be done during coevolution, good starting points for new inhibitors and mutants should be available from successful dockings of closely related inhibitors and mutants from earlier in coevolution.

For even more accuracy, it will be possible to obtain experimental binding evaluations in the lab. We choose inhibitors from a class that can be generated from a combinatorial library. This allows hundreds of such inhibitors to be tested experimentally in a reasonable amount of time.

The simple models, Autodock, and experimental evaluations are three points along a cost/accuracy tradeoff. It should be possible to integrate these during coevolution. This has been done in a GA context with simple staging methods that use computational simulation for evaluations early in a run, and more accurate real-world experiments for later evaluations [68]. In general, less costly models should be used to carefully choose competitions to evaluate with more accurate, more expensive methods.

In addition to integrating different types of evaluations, it is desirable to integrate the exact results from the covering competitive algorithm and a simple model, with heuristic results from coevolution using a more accurate model. One way to do this might be the active tuning of the simple model with Autodock during execution of the covering competitive algorithm. Results obtained using a simple model are checked with Autodock, and mismatches are used to tune model parameters.

Accuracy is a central issue when solving adversarial problems with competition results obtained from models and simulations. This is a general difficulty with using computational models, of course, but the effects may be controllable in the context of adversarial problems. Since perfect solutions must perform well against worst-case tests, we are already demanding robustness. Even without additional effort, this broad robustness may guard against small inaccuracies in the model. Beyond this, it should be possible to establish a range of uncertainty for a model, then incorporate choices from this range of uncertainty into the test cases. We then demand that solutions perform well, for all test cases and for all variations on the model within its range of uncertainty. In the context of inhibitor design, this might be useful for addressing issues like uncertainty in protease conformation. A mutant would be allowed to specify its own choice of conformation within the range of conformational uncertainty. Robust inhibitors must bind to mutants over the (hopefully small) range of uncertainty in their conformation. Rather than solve the computationally intractable conformation problem, we push the burden of conformational uncertainty onto the inhibitor and demand greater robustness of our designs.

In the long term, the coevolutionary methods discussed in this section should offer a significant tool for studying drug resistance, but there are several limitations to this approach. It will be very difficult to capture the entire range of possible viral responses using a computational model, so novel forms of resistance could still emerge against coevolved designs. In the near future, computational limitations on accuracy are likely to weaken guarantees of robustness, even over the range of resistance mutations that are modelled. The models used in this work do not address practical issues in drug design such as toxicity and cost of manufacture, so designs that emerge from coevolutionary methods would need to be refined for actual use. Despite such limitations, coevolutionary methods may provide a powerful tool for the initial stages of drug design. Beyond the design of robust inhibitors, the methods allow study of resistant mutants of particular enzymes, and may help address more general scientific questions about the nature of resistance.

Chapter V

Conclusions

V.A Biological Implications

In Chapter I, we discussed biological evidence that suggests coevolution has features useful for computational problem-solving. Chapters II, III, and IV then presented extensive theory and computational experiments that use coevolutionary methods to solve adversarial problems. Although the computational experiments in Chapters III and IV are not focused on biological modelling, it is still worth going back to explore coevolutionary concepts from these experiments that may be relevant to biological understanding.

The biological literature on coevolution is well developed [38]. Existing work explores a wide variety of evolutionary phenomena that can occur, and includes simple mathematical models. Computational experiments may be able to contribute to this field by providing examples of coevolution that are concrete, and that include a complex search component. The combination of these two factors may yield insights that are difficult to obtain by other means. For example, biological evidence is often vague, and we cannot do controlled experiments over long evolutionary time scales. Analytically tractable mathematical models permit detailed analysis, but typically consider a handful of genes, with a simple dependence of fitness on these genes. Since the fitness of biological organisms depends on complicated developmental programs and rich environments, it may be difficult to consider the complexity of evolutionary search in such simplified models. By doing concrete computational experiments which include a difficult underlying search problem, we may hope to obtain new insights into coevolution.

The computational experiments in this thesis use forms of coevolution that are probably too distant from natural coevolution to be very useful as models. We consider strictly zero-sum competitions between two populations. Such experiments do not explore more complex interactions involving greater numbers of species, and exclude the cooperation that is important in many biological interactions. Coevolving populations are restricted to have synchronized generations; natural coevolutionary interactions such as host-parasite systems often have radically disparate evolutionary rates that probably have a major impact on the behavior of coevolution. Some of our methods for extending computational coevolution probably have no biological correlate at all. Since our experiments lack realism, we do not claim that observed phenomena are indicative of what should be seen in biology. Instead, we offer concepts coming out of these experiments that may find use in biological explanations. Further study of these concepts could include both biological and more realistic computational observations, and would be necessary to establish whether they have any value.

Below, we consider several coevolutionary ideas motivated by computational experiments. We first consider factors that can drive long-term arms races in natural coevolution. We then discuss pedagogy in coevolution. Pedagogy is important in balance-preserving feedback loops based on genetic drift, and in the evolution of drug resistance. Some of the concepts introduced here are fairly speculative; we merely wish to suggest that they may be worthy of further study.

V.B Arms Races

The original concept of the biological arms race included the idea of reciprocal adaptations, and the idea of generally improved competitive ability for tightly interacting species [29]. But it didn't include a complete definition that would allow a test for the existence of a continued arms race over arbitrarily long time scales. Here, we have provided a rough version of such a definition: ability to defeat prior opponents (which are ancestors of the current generation) must increase as evolution proceeds. This is satisfied if the *fraction* of opposition from prior generations that can be defeated remains high and roughly constant as evolution proceeds. This seems natural; if newly evolved individuals generally lose the ability to compete successfully against old opponents, the process has no directionality. There may still be reciprocal adaptations among current competitors, but there are not general increases in competitive ability. The continued arms race requires this directionality.

This definition of the arms race is central to computational coevolution. It offers theoretical guarantees of progress in coevolutionary-like processes (Chapter II). It is the basis for the hall of fame, which promotes this continual progress in practical problem-solving. The idea of directionality may be useful for studying arms races in biological coevolution. Some version of arms race diagrams (Section III.G.5) may be conceptually helpful for this purpose.

V.B.1 Time Scales of Arms Races

In computational coevolution, we desire long-term arms races directed towards problem-solving. Competitive fitness sharing maintains an adequate cover of current opposing niches while search looks for novel adaptations that permit the defeat of larger sets of opponents. This permits short-term advances in the arms race. Beyond such single novel adaptations, the diversity present in a population gives it a temporary memory that encourages advances in the arms race over a slightly longer span of time. A successful novel individual cannot lose the ability to compete successfully against very recent opponents, because relatives of these recent opponents are still present in the opposing population and will rise again if given a chance. Even this diversity becomes inadequate over longer periods of time, once the current opposition no longer tests for the traits required for successful competition against old ancestor opponents. At this point, the hall of fame becomes important. By explicitly testing against ancestral opponents, we directly promote arms race progress over long time scales.

Over what time scales is it reasonable to expect directional arms races in natural coevolution? In a complex ecosystem with ongoing speciation in opponents, and spatial variation in each opposing species, there may be a great amount of diversity that extends the temporary memory described above. Still, such mechanisms probably have minimal effect over spans of time comparable to the length of evolutionary history. After 100 million years, the ecology is so different that it is unrealistic to expect new adaptations to be successful against radically different ancestral species. In general, we do not expect protracted arms races over such long time scales.

V.B.2 Univariate Phenotypic Traits

We consider a possible class of exceptions to this general lack of long-term arms races in biological coevolution. Suppose coevolution between two competing species depends strongly on a univariate phenotypic trait that has a generally monotonic effect on competitive ability. Any genetic changes that affect this phenotypic trait must respect this monotonicity. Over the short term, coevolution should tend to increase competitive ability by changing this trait in the proper direction. But since this proper direction never changes (since the effect of the trait is monotonic), and there is only one axis of variation, these short term changes all move in the same direction and add up to long-term directional change in the trait. Because of the monotonic effect on competitive ability, this should result in a long-term arms race.

One example of this is running speed in mammalian predator and prey species. Using anatomical indicators of running speed, a fairly steady increase in running speed has been observed in North American ungulates and their predators over the last 60 million years [13]. Although there may be costs involved, we generally expect faster running speed to be better for competition between predators and prey. The long-term increase in running speed may have required a wide variety of changes at the genetic level at various times. But because competition success depends in a simple, monotonic way on running speed, we see a protracted arms race.

It is probably advantageous for a species to possess such univariate phenotypic traits. When coevolution demands increased competitive ability, it should be a fairly simple matter to search for some genetic change that moves such a trait in the required direction. Competitors that possess such traits should generally fare well in coevolutionary arms races that depend on the trait. Because of this, evolution may favor developmental systems that give rise to such univariate phenotypic traits. In such developmental systems, any genetic change that increases the phenotypic trait should respect monotonicity by also increasing the competitive ability for which that trait is relevant. As long as this is roughly true, the species should enjoy a coevolutionary advantage. A specific candidate example of this is given below.

This idea is quite speculative, but there is evidence that suggests coevolutionary pressures can promote the evolution of evolutionary mechanisms themselves. A survey of the use of sexual recombination by animals in a variety of ecosystems reveals that it is more prevalent when the ecosystem is biologically complex, with a wide variety of coevolutionary interactions [40]. It is less common in ecosystems with fewer predatory and pathogenic species. It is believed that the coevolutionary pressures in complex environments demand high evolutionary rates of participating species. Species that evolve more slowly due to a lack of sexual recombination are not competitively viable [89]. So, it seems that pressure for coevolutionary success can promote change in the basic mechanisms on which evolution depends.

As a possible example of a case where coevolution may have favored developmental systems with univariate phenotypic traits, consider the long-term increase in normalized brain size, in predators and prey, discussed in Chapter I. First, consider a situation where coevolving predators and prey adapt by producing novel specialized cognitive abilities relevant to the competition. In this case, there is no reason to expect cognitive responses to ancestral competitors to be maintained. Such specialized responses would become obsolete and their maintenance would be unnecessary. Old cognitive abilities could simply be replaced by new ones. There would be no need for brain size to increase in this case, since the repertoire of behaviors does not become more complex over time.

The brain is highly complex, and the evolution of novel specialized cognitive abilities may be slow and difficult. If, instead, developmental systems tend to produce a few univariate phenotypic traits that correspond to general cognitive capabilities, then evolution could proceed more rapidly. Rather than assemble complex new cognitive abilities from scratch, evolution might simply change a few genes that generally increase "learning capacity" (for example). Such general cognitive abilities should have a simple monotonic affect on competitive ability, so they could satisfy the requirement described above for a long-term arms race.

This hypothesis is no doubt highly simplified, even if there is some truth in it. But it does offer a potential explanation for the long-term increase in brain size. It is easy to imagine that univariate cognitive traits correlate with brain size. With a developmental system possessing such univariate cognitive traits, a protracted arms race would be expected, with long-term increases in brain size as a result.

The computational experiments in Chapters III and IV do not offer sufficiently complex representations to consider the evolution of developmental systems. But there have been experiments combining genetic algorithms with models of development [43], so there is some hope that computational experiments could eventually shed light on this hypothesis.

V.C Pedagogy, Drift, and Resistance

Section III.G.6 discussed the idea of pedagogy in coevolution. When no member of a population can compete successfully against a difficult opponent, competition against related opponents must provide the pedagogical fitness information needed to guide search. These related opponents are "stepping stones" that are easier to defeat, with success in defeating them bringing individuals closer to defeating the difficult opponent.

Pedagogy becomes important when search is hard. If, instead, fitness increases against a difficult opponent are easily obtained via single point mutations, then pedagogical stepping stones are not required to guide search towards the defeat of difficult opponents. Such stepping stones only matter when increased fitness is unavailable to genotypes that are easy to generate from the current population. In such a case, evolution requires fitness information from competition against the stepping stones to reach genotypes that are more successful against the difficult opponent.

In general, when there is little or no guiding fitness information from a difficult opponent, competition with other opponents is a significant determinant of the direction of evolution. If these other opponents include relevant pedagogical stepping stones, then evolution should eventually produce individuals with improved ability to compete against the difficult opponent. If, on the other hand, the defeat of these other opponents is not informative with respect to the difficult opponent, then evolution may not get any closer to (and may get further away from) improved ability to compete against the difficult opponent. So, when evolutionary innovation produces new difficult opposition, the nature of the remaining ("non-difficult") opposition helps determine the eventual outcome of coevolution.

It seems that most simple mathematical coevolutionary models do not include the possibility of a search that is so difficult that pedagogical stepping stones are required to make progress [38]. Concepts related to pedagogy and search difficulty have been discussed in the context of virus evolution [26]. For complex organisms, however, it may be difficult to understand the evolutionary landscape in sufficient detail to show the need for pedagogical stepping stones. This may make computational experiments valuable in providing understandable coevolutionary systems with sufficient complexity to make pedagogy a significant factor. The concepts described above may be useful starting points in understanding pedagogy.

Below, we consider two applications of pedagogy in biological evolution. First, we discuss the generation of pedagogical stepping stones via genetic drift. Second, we consider pedagogy in the evolution of resistance (for example, pesticide resistance in insects and drug resistance in viruses). Since the evolution of resistance is partially under human control, an understanding of pedagogy in this domain may have practical significance.

V.C.1 Drift

In Chapter III, we emphasized the importance of balance during coevolutionary competition. When one side gets far behind in computational coevolution, both the dominating side and the dominated side lose guiding fitness information. Coevolution can become slow and aimless. In biological coevolution, there are a variety of possible responses to a lack of balance. The dominated side can become extinct: herbivores may have gotten far enough ahead (in running speed and brain size) to be a major factor in the extinction of creodonts, carnivores of the early Tertiary [47]. If the side that is dominating the competition relies on exploitation of the opposition, it may have to back off from this domination to preserve the opposition for its own survival. This may contribute, for example, to the evolution of reduced virulence in parasites [62]. Finally, there may be a feedback loop that restores balance by increasing selection pressure on the dominated species and reducing selection pressure on the dominating species [79]. This difference in evolutionary rates tends to bring the species back into parity.

Such a feedback loop may not be able to operate smoothly if the fitness of competitors has a complex, noncontinuous dependence on genotypes. If there is a rapid evolutionary jump based on a novel mutation in the dominating species, both sides may lose guiding fitness information. Without informative fitness, relative selection pressure is not relevant; neither side can evolve meaningfully. The feedback loop based on differential selection pressure may not be able to operate in a straightforward manner in this case.

Such a rapid evolutionary change may create a new difficult opponent, without informative fitness or pedagogical stepping stones to guide evolution towards a response to this opponent. In computational experiments, random genetic drift becomes an important force when both sides have lost guiding fitness information in a situation such as this. Drift can create degraded versions of the difficult opponent that act as the necessary stepping stones (see Section III.G.6). Since random drift can take place in natural populations, it is possible that this is a factor in a feedback loop that preserves coevolutionary balance. When evolutionary discontinuities cause a loss of fitness information, drift can provide the pedagogical information that restores a balancing feedback loop based on relative selection pressure.

V.C.2 Resistance

Situations similar to coevolution can arise when one species makes evolutionary adaptations to human (non-evolutionary) efforts against it. Such situations arise, for example, in insect resistance to pesticides, and in drug resistance among viruses, bacteria, and parasites. The general engineering approach in these situations attempts to find chemical attacks that are effective against the opponent species, in its current form. There may be some hope of delaying the evolution of resistance if the chemical attack is novel and powerful. Beyond this vague hope, evolutionary responses in the opponent species are typically not accounted for until after they occur. Section IV.D discusses a computational methodology for predicting some of the possible opposing responses during drug design, before the drug is ever deployed.

Such a method may be important beyond improving the engineering of new drugs. When drugs that are only partially effective are deployed, resistance can arise quickly. The effectiveness of the drug is quite limited in this case, and it may have the dangerous effect of providing a stepping stone for evolution in the opposing species. Later therapies that include drugs related to the one deployed may be less effective against opponents that have already evolved resistance to the stepping stone. Such opponents have less evolutionary distance to cover in evolving resistance to the later therapy. In some cases, it may be better to avoid deploying the drug altogether, than to risk supplying such stepping stones. It may be best to wait until a more effective therapy is available, against which the evolution of resistance is unlikely among opponents that have not had the benefit of the stepping stone. To make such decisions, it is important to understand pedagogy in the context of the opponent species and the range of possible drugs that may be used against it.

As a specific example, combination therapies for HIV have recently proven quite effective. Such therapies include multiple drugs, to inhibit HIV replication at multiple points in its life cycle. It is *not* the case that resistance mutations do not exist to such combination therapies. Instead, such therapies create a difficult evolutionary problem for HIV: multiple mutations are required for effective resistance. Since such multiple mutations are rare, HIV replication can be suppressed for a very long time [26]. In patients that have already received some subset of the drugs included in the combination, prior to starting combination therapy, resistant forms of HIV may already be present that overcame the prior stepping stone. Since HIV displays cross-resistance to multiple related drugs when exposed to a single drug in a class, such stepping stones need not even be identical to any particular drug in the combination therapy. In patients infected with HIV that has been exposed to such stepping stones, HIV resistance to the combination therapy may require fewer mutations and therefore arise much more rapidly. The combination therapy may only be fully effective in patients that never received drugs related to those used in the combination therapy.

In situations like this one, humans have a substantial amount of control over the evolution of an opposing species. By providing evolutionary stepping stones, preliminary efforts at controlling the species may only hamper the effectiveness of later efforts. It may be best to avoid deployment of such stepping stones even though they help in the short term. Given a detailed understanding of the opposing species, it may even be possible to deploy "anti-stepping stones" that guide evolution of the opponent *away* from resistance to a planned later effort against it. Such measures might actually be harmful in the short-term, but could have the benefit of great effectiveness in the long run by making the evolution of resistance extremely unlikely.

V.D Future Work

In addition to ideas for future work presented above in relevant sections, several additional methods, uses for coevolution, and coevolutionary phenomena are worth future exploration.

V.D.1 Further Methods for Competitive Coevolution

Spatially Distributed Coevolution

The spatial distribution of populations is likely to be a useful technique for coevolutionary heuristics. Spatially distributed populations have been successfully used in non-coevolutionary genetic algorithms to efficiently parallelize evolution, and maintain multiple solution niches during search [63].

One type of distributed evolution breaks a population up into separate subpopulation "islands", with small amounts of migration among islands maintaining communication between subpopulations. A second type of spatial distribution places individuals on a grid (which may be one-dimensional, two-dimensional, or higher-dimensional); mating and reproductive competition take place among individuals that are nearby on the grid.

Spatially distributed populations have also been used with competitive coevolution, in Hillis' experiments on sorting network design [44]. In this work, both solutions and test cases are distributed on a two-dimensional grid, and competition takes place between solutions and test cases at the same spatial location on the grid. An island version of coevolution would also be possible.

Distributed populations may be able to serve several roles in coevolution. First, the restriction of competition to local opposition provides opportunities for specialization. While general solutions that handle all opposition are required in the long-term, specialization in the short term can be useful to guide search. Individual test cases do not even necessarily need to cover a significant range of possible candidate solutions. Specialization among test cases may allow them to provide more informative testing for local candidate solutions.

A second role for spatial distribution is the maintenance of diverse niches during coevolution. Competitive fitness sharing (see Section III.C.1) helps provide this by maintaining adequate diversity to cover opposition. But pressure for diversity may not exist early in coevolution, when a richly structured competition requiring large covers has not yet been set up. It also may not exist late in coevolution, if populations are out of balance. As a simple approach to maintaining diversity through these periods during controller/patient evolution in Section IV.A, a large patient population was used. But spatial distribution naturally maintains multiple genetically distinct subpopulations, which should help when sharing is unable to preserve adequate useful diversity. Preliminary experiments indicate that a simple island distribution, even without localized competitive interactions, is useful for improving coevolutionary performance. This should be further explored, especially for use in problems where adequate diversity is difficult to maintain.

Efficient Testing

A second class of techniques that should be straightforward to use in coevolution involve reductions in testing work for computational efficiency. As a simple example, competition outcomes during fitness testing are computed either for use in tournament selection (usually in a tournament of size two) or for finding the best individuals in a population to use as elitists. For both of these purposes, it is often unnecessary to compute the outcome for all competitions. In a selection tournament, testing of an individual can stop when it is no longer possible for it to win the tournament (even if it won all remaining competitions against the parasite sample). When maximizing performance to find an elitist, an individual does not need to be tested further once it has been proven worse than the best-so-far. These simple methods should provide small computational savings with no impact on coevolutionary behavior.

Beyond this, less precise methods should be useful for stopping testing early when existing evidence indicates that an individual is likely to be poor. Brood selection does this in a simple manner; offspring are evaluated on a limited sample, and only those that appear promising are admitted to the final population for complete fitness testing. But it should be possible to refine this two-round competition to smoothly adjust the amount of testing performed as a function of the relative estimated performance of individuals being compared. It may be possible to analytically derive such methods using simple statistical assumptions.

V.D.2 New Uses for Coevolution

Coevolving against Model Uncertainty

In Section IV.D.5, we discussed the issue of model accuracy in the drug design problem. Due to model approximations and computational limitations, evaluations of binding during the coevolution of inhibitors and resistant mutants are necessarily of limited accuracy. It was suggested that the coevolutionary approach to generating solutions that are robust against a range of test cases might also be used to generate solutions that are robust across the range of model uncertainty.

This idea could be tested more simply in a non-coevolutionary optimization context. Consider the optimization of a design, using a simulation to obtain a (continuous) evaluation of the design. A threshold of acceptable performance defines the point at which optimization has solved the problem. For example, we might use optimization to design a controller to navigate a robot through an environment, using a simulation of the environment [68]. The objective function optimized could be the time required to navigate through the environment, with a threshold giving acceptable time.

The simulation of a physical environment is necessarily approximate. In the robot example, positions and sizes of obstacles may not have been exactly measured, and robot response to control inputs may be variable or imprecisely calibrated. These sources of uncertainty contribute to errors in final design. By studying these sources of uncertainty in the model used to evaluate designs, we can extend the model to give ranges of uncertainty on relevant parameters. For example, ranges of variation or uncertainty in obstacle positions and robot responses could be estimated by comparing measurements used in simulation to true values and behavior.

Once these ranges of uncertainty are established, a test case can be defined as a set of choices for parameter offsets chosen from these ranges of uncertainty. Coevolution can then be used to find designs that meet the required performance threshold across the range of possible test cases. If such a robust design exists, we have higher expectations of its real-world performance than we would of non-coevolved designs. The robust design remains correct over the entire range of possible simulation error, while no such guarantee is available for the non-coevolved design.

Optimization of designs that are robust over a range of model uncertainty should be possible in situations where models approximate real-world performance, as well as situations where uncertainty is due to approximations made for the sake of computational efficiency. The space of possible designs needs to be flexible enough to include conservative designs that can meet the performance threshold over a range of model error, and the range of possible uncertainty must not be so wide as to prohibit the existence of robust designs. If these conditions can be met, coevolution may make simulated performance evaluations a more realistic tool for design problems.

Integrating Coevolution with Tree Search

Conventional game-playing programs make heavy use of tree search to evaluate the consequences of various lines of play. While evaluation of non-terminal positions is usually heuristic, tree search is often exact; within a finite horizon, the results of complete tree search are obtained. [1] In games such as Go with large search trees, such complete tree-search methods may be wasteful. Many lines of play are quite similar, so that a more intelligent search method could generalize from examination

[1]Though alpha-beta pruning and related methods usually allow significant pruning of search without affecting results.

of a small number of example lines of play.

To some extent, coevolution of game strategies can be viewed as this sort of generalizing tree search. Each competition explores a single line of play down to the end of the game. Playing to the end of the game removes inaccuracy due to misevaluation of terminal positions; other game-playing techniques such as rollouts in backgammon rely on this to obtain accurate evaluations of positions [95]. Entire strategies are evaluated based on a sample of a few such lines of play. The coevolutionary process improves strategies based on information obtained from these lines of play, so that later search focuses on lines that include strong moves and responses. Section IV.C.7 discusses the idea of using coevolution in real-time as a tree-search method.

Restriction to single, unbranched lines of play may limit the accuracy of results in some games. In the Go experiment described in Section IV.C, coevolved strategies were augmented with limited tactical search. Even this restricted tree search was important in overcoming the tunnel vision of coevolution's exploration of unbranched lines of play.

Coevolution in games may benefit from further integration with tree search. One simple way of doing this is to use tree search during endgames, where exact results may be obtained without using heuristic evaluation functions. When games played between coevolving strategies reach a point where tree search can determine the exact value of the position, play can be halted. The removal of the burden of the endgame from coevolution should make search easier, and allow more accurate strategy evaluations throughout a coevolutionary run.

Exact endgame search techniques are well-developed, due to their application in computing complete solutions to games [5, 6]. Methods have been developed that allow extremely fast identification of won positions, many moves away from the actual end of the game. Solutions to Qubic and Go-Moku depended on such techniques [5, 6]. These games may be good test domains for the integration of coevolution with endgame tree search.

V.D.3 Informative Parasites

Coevolutionary search is guided by the information provided by competition against parasites. Effective search requires that parasites be informative. Important basic questions in this context are what it means for parasites to be "informative," and how to choose informative parasites.

We may attempt to define what it means for a single parasite to be informative. At extreme ends of the competitive ability of parasites, such a definition seems possible. If a parasite defeats all hosts, then it does not help distinguish any hosts for purposes of guiding search. Parasites that lose to all hosts are similarly unhelpful. Such parasites are uninformative in the context of their current opposition, because they give a uniform result against all members of this opposition.

Between these extremes, parasites do provide fitness information by defeating some hosts while losing to others. But the question of whether a single isolated parasite is informative may be malformed. For coevolutionary progress to continue, we demand that hosts generalize by defeating a range of parasitic opposition, rather than specializing in the defeat of a single parasite. Testing hosts against a single isolated parasite offers no information about whether the host generalizes or specializes. Because of this, isolated parasites provide little information useful for guiding host evolution.

Groups of multiple parasites, on the other hand, may be quite informative. If some hosts defeat every member of a particular group of parasites, while other hosts specialize in the defeat of a small subset of the group, then these parasites help distinguish generalist hosts from specialists. Such groups of parasites may be considered informative in a coevolutionary context.

Since fitness testing treats parasites as individual units in the methods of Chapter III, simply summing results over parasites, we are not using all of the information available from competition against informative groups of multiple parasites. Competitive fitness sharing (Section III.C.1) rewards hosts for defeating parasites that few other hosts can defeat. As a first step towards using competition information against groups of multiple parasites, this may be extended to reward hosts that defeat *pairs* of parasites that few other hosts can defeat. This should begin to reward hosts that generalize across parasites in novel ways. Preliminary tests on Nim indicate that this provides at least a small computational advantage.

V.D.4 Displaying Covering Sets

The arms race diagrams developed in Section III.G.5 are useful for displaying arms race progress when both populations are successful in producing new individuals with broad competitive ability. In adversarial problems where specification number is substantially larger than one, individual test cases eventually lose this broad competitive ability. This may be seen in the 2DTTT and 3DTTT test problems of Chapter III, and in sorting network design (Section IV.B). For such problems, individual test cases cannot retain ability to cover a large fraction of candidate solutions. The arms race diagram displays this lack of ability, but provides little information about coevolutionary behavior beyond this for these problems.

Even if individual test cases lose the ability to cover all solutions, larger sets of test cases may retain this ability and allow coevolution to succeed. Arms race diagrams may be extensible by extracting these covering sets each generation, and displaying these to show whether populations retain the ability to cover all prior competitors with multiple individuals. The display of covering sets should also be useful in the solution population for problems in which single solutions are inadequate during coevolution.

V.E Summary

Competitive coevolution is able to solve adversarial problems by producing progressively better solutions and more difficult test cases in an evolutionary arms race. The capabilities of coevolution, and the behavior underlying its performance, have been explored with both theoretical and heuristic methods.

Results in Chapter II show that the central idea of coevolution - finding new solutions and test cases to defeat prior ones - is able to efficiently search through large spaces of solutions and test cases under the right conditions. These conditions include a low specification number so that perfect solutions may be practically distinguished, the production of new solutions and test cases that are effective against all prior opposition, and randomization to avoid perversely slow climbs up exponentially long transitive chains. For simple applications such as the simplified drug design problem in Section IV.D, exact methods from Chapter II can be efficiently applied to obtain complete solutions.

Experiments with heuristic coevolutionary methods in Chapter III show that simple forms of coevolution can fail in several ways. Coevolution can lose diversity, allowing the emergence of specialists rather than driving the evolution of robust solutions. Over many generations, coevolutionary progress can break down as new individuals are produced that are unable to defeat older opposition. Coevolutionary balance can be lost when one population gets far ahead of the other; both populations lose guiding fitness information in such situations.

New methods for competitive coevolution are able to overcome these problems and make coevolution more computationally efficient. Competitive fitness sharing preserves diversity by maintaining a cover of all parasites in each host population. Shared sampling allows effective fitness evaluation during coevolution by selecting strong, diverse samples of parasites. The hall of fame encourages arms race progress by retaining individuals from earlier generations for later use as parasites. The phantom parasite helps restore coevolutionary balance. Brood selection, a technique from genetic programming, can substantially reduce the amount of computational effort required for testing. With these methods, coevolution is able to efficiently solve several test problems that cannot be solved efficiently without them. Measurement techniques reveal the impact of these new methods on the phenomena underlying coevolutionary behavior.

Several applications illustrate the broad applicability of coevolution, and reveal diverse modes of coevolutionary operation. In the coevolution of controllers and plants (Section IV.A), continuous parameters and nonlinear constraints help create a difficult search problem. Coevolution is able overcome a natural imbalance in this problem to eventually produce a small set of plants, for which no single perfect controller seems to exist. The coevolution of sorting networks (Section IV.B illustrates the behavior of coevolution when specification number is high; test cases are able to track a changing set of flaws in an evolving population of solutions, without being able to evolve a complete teaching set. The game of Go is used to explore the behavior of coevolution in a more open-ended domain (Section IV.C), where we do not expect a final, perfect solution. Coevolution succeeds in pushing strategies to the limits of their representational capabilities, and in producing a competitive level of play. Finally, a simple model of HIV protease interaction with protease inhibitors is used with the exact theoretical methods of Chapter II to find inhibitors that are robust across possible resistance mutations. Future work on models with more chemical realism should provide a significant tool for drug design.

Bibliography

[1] Computer go ladder. Available online at http://www.cgl.ucsf.edu/go/ladder.html.

[2] Computer go mailing list. Information available online at http://cs.anu.edu.au/ Lex.Weaver/COMPUTER-GO/.

[3] D.H. Ackley. ccr: A network of worlds for research. In *Artificial Life V*, 1996.

[4] J.S. Albus. Mechanisms of planning and problem solving in the brain. *Mathematical Biosciences*, 45:247–293, 1979.

[5] L.V. Allis, H.J. Van Den Herik, and P.H. Huntjens. Go-Moku solved by new search techniques. *Computational Intelligence*, 12(1), 1996.

[6] L.V. Allis, M. Van Der Meulen, and H. Jaap Van Den Herik. Proof-number search. *Artificial Intelligence*, 66(1), 1994.

[7] E. Amaldi and V. Kann. The complexity and approximability of finding maximum feasible subsystems of linear relations. *Theoretical Computer Science*, 147:181–210, 1995.

[8] P. J. Angeline and J. B. Pollack. Competitive environments evolve better solutions for complex tasks. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann, 1993.

[9] D. Angluin. Negative results for equivalence queries. *Machine Learning*, 5(2), 1990.

[10] M. Anthony, G. Brightwell, D. Cohen, and J. Shawe-Taylor. On exact specification by examples. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, 1992.

[11] American Go Association. AGA computer go. Available online at http://www.usgo.org/computer/, 1997.

[12] R. Axelrod. Evolution of strategies in the Iterated Prisoner's Dilemma. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*. Morgan Kaufmann, 1989.

[13] R.T. Bakker. The deer flees, the wolf pursues: incongruencies in predator-prey coevolution. In D.J. Futuyma and M. Slatkin, editors, *Coevolution*. Sinauer, 1983.

[14] E.B. Baum, D. Boneh, and C. Garrett. Where genetic algorithms excel. This is an expanded and improved version of "On Genetic Algorithms" which appeared in the Proceedings of the Eighth Annual Conference on Computational Learning Theory, ACM., 1995.

[15] M.A. Bedau and N.H. Packard. Measurement of evolutionary activity, teleology, and life. In *Artificial Life II*. Addison-Wesley, 1991.

[16] R.K. Belew and M. Mitchell. *Adaptive Individuals in Evolving Populations: Models and Algorithms*. Addison-Wesley, 1996.

[17] R.K. Belew and M.D. Vose, editors. *Foundations of Genetic Algorithms 4*. Morgan Kaufmann, 1997.

[18] E.R. Berlekamp, J.H. Conway, and R.K. Guy. *Winning Ways for Your Mathematical Plays*. Academic Press, 1982.

[19] N.H. Bshouty, R. Cleve, S. Kannan, and C. Tamon. Oracles and queries that are sufficient for exact learning. In *Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory, COLT 94*. ACM, 1994.

[20] J. Burmeister and J. Wiles. The challenge of go as a domain for ai research: A comparison between go and chess. In *Proceedings of the Third Australian and New Zealand Conference on Intelligent Information Systems*. IEEE Western Australia Section, 1995.

[21] K.C. Chou, A.G. Tomasselli, I.M. Reardon, and R.L. Heinrikson. Predicting human immunodeficiency virus protease cleavage sites in proteins by a discriminant function method. *Proteins*, 24(1):51–72, 1996.

[22] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.

[23] D. Cliff and G. F. Miller. Tracking the red queen: Measurements of adaptive progress in co-evolutionary simulations. In F. Moran, A. Moreno, J. J. Merelo, and P. Chacon, editors, *Advances in Artificial Life. Third European Conference on Artificial Life Proceedings.* Springer-Verlag, 1995.

[24] D. Cohn, L. Atlas, and R. Ladner. Improving generalization with active learning. *Machine Learning*, 15(2):201–221, May 1994.

[25] D.A. Cohn, Z. Ghahramani, and M.I. Jordan. Active learning with statistical models. *Journal of Artificial Intelligence Research*, 4:129–145, 1996.

[26] J.H. Condra and E.A. Emini. Preventing hiv-1 drug resistance. *Science & Medicine*, January/February 1997.

[27] J.H. Condra, William A. Schleif, et al. In vivo emergence of hiv-1 variants resistant to multiple protease inhibitors. *Nature*, 374:569–571, April 1995.

[28] P. Darwen and X. Yao. Automatic modularisation by speciation. In *Proceedings of the Third IEEE International Conference on Evolutionary Computation (ICEC'96)*, 1996.

[29] R. Dawkins and J.R. Krebs. Arms races between and within species. *Proceedings of the Royal Society of London B*, 205:489–511, September 1979.

[30] T. Durham. A program with a touch of Zen. In *Computing Horizons*. Addison-Wesley, 1985.

[31] M. Enzenberger. The integration of a priori knowledge into a go playing neural network. ftp://ftp.cgl.ucsf.edu/pub/pett/go/ladder/NeuroGo.ps.gz, 1996.

[32] S.L. Epstein. Toward an ideal trainer. *Machine Learning*, 15(3):251–277, June 1994.

[33] L.J. Eshelman, editor. *Proceedings of the Sixth International Conference on Genetic Algorithms*. Morgan Kaufmann, 1995.

[34] A.S. Fraenkel, M.R. Garey, D.S. Johnson, T. Schafer, and Y. Yesha. The complexity of checkers on an n x n board - preliminary report. In *FOCS 19*, 1978.

[35] R.J. Freund and W.J. Wilson. *Statistical Methods*. Academic Press, 1997.

[36] Y. Freund and R.E. Schapire. Game theory, on-line prediction and boosting. In *Proceedings of the Ninth Annual Conference on Computational Learning Theory*, 1996.

[37] D. Fudenberg and D.K. Levine. *Theory of Learning in Games*. published online at http://levine.sscnet.ucla.edu/Papers/CONTENTS.HTM, 1997.

[38] D.J. Futuyma and M. Slatkin. *Coevolution*. Sinauer, 1983.

[39] A. Giordana and F. Neri. Search-intensive concept induction. *Evolutionary Computation*, 3(4), 1995.

[40] R.R. Glesener and D. Tilman. Sexuality and the components of environmental uncertainty. *American Naturalist*, 1978.

[41] D.E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In John J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*. L. Erlbaum Assoc., 1987.

[42] S.A Goldman and M.J. Kearns. On the complexity of teaching. In M.K. Warmuth and Leslie Valiant, editors, *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*. Morgan Kaufmann, 1991.

[43] W.E. Hart, T.E. Kammeyer, and R.K. Belew. The role of development in genetic algorithms. In D. Whitley and M. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 315–332. Morgan Kaufmann, 1995.

[44] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In C. Langton et al., editor, *Artificial Life II*. Addison-Wesley, 1991.

[45] J.H. Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press, 1992.

[46] Japan Computer Shogi Association and Japan Computer Go Association. *Game Programming Workshop in Japan '95*, 1995.

[47] H.J. Jerison. *Evolution of the Brain and Intelligence*. Academic Press, 1973.

[48] M. Jerrum, L. Valiant, and V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43(2), 1986.

[49] H. Juillé. Evolution of non-deterministic incremental algorithms as a new approach for search in state spaces. In Larry J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*. Morgan Kaufmann, 1995.

[50] T.E. Kammeyer, R.K. Belew, and S.G. Williamson. Evolving compare-exchange networks using grammars. *Artificial Life*, 2(2), 1995.

[51] M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[52] D.E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison Wesley, 1973.

[53] J.R. Koza. *Genetic programming : on the programming of computers by means of natural selection*. MIT Press, 1992.

[54] M. Land and R.K. Belew. Towards a self-replicating language for computation. In J.R. McDonnell, R.G. Reynolds, and D.B. Fogel, editors, *Evolutionary Programming IV*. MIT Press, 1995.

[55] D.B. Lenat. AM: Discovery in mathematics as heuristic search. In R. Davis and D.B. Lenat, editors, *Knowledge-Based Systems in Artificial Intelligence*, pages 1–225. McGraw-Hill, 1982.

[56] D. Levy. *Computer Games*. Springer-Verlag, 1988.

[57] K. Lindgren and M.G. Nordahl. Artificial food webs. In *Artificial Life III*, 1994.

[58] R.J. Lipton and N.E. Young. Simple strategies for large zero-sum games with applications to complexity theory. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing. STOC '94*. ACM, 1994.

[59] N. Littlestone. Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm. *Machine Learning*, 2(4), 1988.

[60] W. Maass. On-line learning with an oblivious environment and the power of randomization. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, 1991.

[61] S.W. Mahfoud. Population size and genetic drift in fitness sharing. In L.D. Whitley and M.D. Vose, editors, *Foundations of Genetic Algorithms 3*. Morgan Kaufmann, 1995.

[62] R.M. May and R.M. Anderson. Parasite-host coevolution. In D.J. Futuyma and M. Slatkin, editors, *Coevolution*. Sinauer, 1983.

[63] J. McInerney. *Biologically Influenced Algorithms and Parallelism in Non-Linear Optimization*. PhD thesis, University of California, San Diego, 1992.

[64] A. Mehlmann. *Applied Differential Games*. Plenum, 1988.

[65] G.M. Morris, D.S. Goodsell, R. Huey, and A.J. Olson. Distributed automated docking of flexible ligands to proteins - parallel applications of autodock 2.4. *Journal of Computer-Aided Molecular Design*, 10:293–304, 1996.

[66] Martin Mueller. Explorer. Available online at http://nobi.ethz.ch/martin/explorer.html.

[67] Martin Mueller. *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory*. PhD thesis, Institute for Theoretical Computer Science, ETH Zurich, 1995.

[68] S. Nolfi, D. Floreano, O. Miglino, and F. Mondada. How to evolve autonomous robots: different approaches in evolutionary robotics. In *Artificial Life IV*. MIT Press, 1994.

[69] C.K. Oei, D.E. Goldberg, and S.J. Chang. Tournament selection, niching, and the preservation of diversity. Technical Report IlliGAL Report 91011, Urbana: University of Illinois, Illinois Genetic Algorithms Laboratory, 1991.

[70] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[71] C.H. Papadimitriou and M. Yannakakis. On complexity as bounded rationality. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing. STOC '94*. ACM, 1994.

[72] I. Parberry. Single-exception sorting networks and the computational complexity of optimal sorting network verification. *Mathematical Systems Theory*, 23(2):81–93, 1990.

[73] J. Paredis. Coevolutionary computation. *Artificial Life*, 2(355-375), 1995.

[74] M. Plutowki and H. White. Selecting concise training sets from clean data. *IEEE Transactions on Neural Networks*, 4(2):305–318, March 1993.

[75] J. Pollack, A. Blair, and M. Land. Coevolution of a backgammon player. In *Artificial Life V*, 1996.

[76] M.A. Potter. *The Design and Analysis of a Computational Model of Cooperative Coevolution*. PhD thesis, George Mason University, 1997.

[77] W. Reitman and B. Wilcox. The structure and performance of the INTERIM.2 Go program. In *Proceedings of Sixth International Joint Conference on Artificial Intelligence*. IJCAII, 1979.

[78] Craig W. Reynolds. Competition, coevolution, and the game of tag. In Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV*. MIT Press, 1994. Used competitive co-evolution successfully to evolve strategies for the game of tag, treated as a differential game on a plane.

[79] M.L. Rosenzweig. Evolution of the predator isocline. *Evolution*, 27:84–94, 1973.

[80] C.D. Rosin and R.K. Belew. Methods for competitive co-evolution: Finding opponents worth beating. In L.J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*. Morgan Kaufmann, 1995.

[81] C.D. Rosin and R.K. Belew. A competitive approach to game learning. In *Proceedings of the Ninth Annual Conference on Computational Learning Theory*, 1996.

[82] C.D. Rosin and R.K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1), 1997.

[83] C.D. Rosin, R.S. Halliday, W.E. Hart, and R.K. Belew. A comparison of global and local search methods in drug docking. In *Proceedings of the Seventh International Conference on Genetic Algorithms*, 1997.

[84] A. Samuel. Some studies in machine learning using the game of checkers. In E.A. Feigenbaum and J. Feldman, editors, *Computers and Thought*. McGraw-Hill, 1963.

[85] N. Schraudolph, P. Dayan, and T.J. Sejnowski. Temporal difference learning of position evaluation in the game of go. In *Advances in Neural Information Processing Systems 6*, 1994.

[86] A.V. Sebald and J. Schlenzig. Minimax design of neural net controllers for highly uncertain plants. *IEEE Transactions on Neural Networks*, 5(1):73–82, January 1994.

[87] A.V. Sebald, G. Schnurer, M. Parti, N.T. Smith, and M.L. Quinn. A dynamic empirical model of the human response to sodium nitroprusside during cardiac surgery. In *Proceedings, 10th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, volume 2, pages 517–518, 1988.

[88] Karl Sims. Evolving 3d morphology and behavior by competition. In Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV*. MIT Press, 1994.

[89] J. Maynard Smith. *The Evolution of Sex*. Cambridge University Press, 1978.

[90] R.E. Smith, S. Forrest, and A.S. Perelson. Searching for diverse, cooperative populations with genetic algorithms. *Evolutionary Computation*, 1(2), 1993.

[91] Chuen-Tsai Sun, Ying-Hong Liao, Jing-Yi Lu, and Fu-May Zheng. Genetic algorithm learning in game playing with multiple coaches. In *Proceedings of the First IEEE Conference on Evolutionary Computation*. IEEE, 1994.

[92] P.D. Surry and N.J. Radcliffe. Real representations. In R.K. Belew and M.D. Vose, editors, *Foundations of Genetic Algorithms 4*. Morgan Kaufmann, 1997.

[93] W.A. Tackett and A. Carmi. The unique implications of brood selection for genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation*. IEEE, 1994.

[94] C.J. Tan. Deep Blue: computer chess and massively parallel systems. In *Conference Proceedings of the 1995 International Conference on Supercomputing*. ACM, 1995.

[95] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, March 1995.

[96] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[97] S. Walker, R. Lister, and T. Downs. Temporal difference, non-determinism, and noise: a case study on the 'Othello' board game. In M. Marinaro and P.G. Morasso, editors, *ICANN '94. Proceedings of the International Conference on Artificial Neural Networks*. Springer-Verlag, 1994.

[98] L. Weaver and T. Bossomaier. Evolution of neural networks to play the game of dots-and-boxes. Poster session presented at Artificial Life V, May 1996.

[99] A.S. Wu and R.K. Lindsay. Empirical studies of the genetic algorithm with non-coding segments. *Evolutionary Computation*, 3(2), 1995.