Moshe Sipper

Evolved to Win

ISBN: 978-1-4709-7283-7

To Dalit, Alon, and Assaf, with all my love

About the Author

Moshe Sipper is a Professor of Computer Science at Ben-Gurion University of the Negev, Israel. He received his B.A. degree from the Technion—Israel Institute of Technology, and his M.Sc. and Ph.D. degrees from Tel Aviv University, all in computer science. During 1995-2001 he was a Senior Researcher at the Swiss Federal Institute of Technology in Lausanne.

Dr. Sipper's current research focuses on evolutionary computation, mainly as applied to games and software development. At some point or other he also did research in the following areas: bio-inspired computing, cellular automata, cellular computing, artificial self-replication, evolvable hardware, artificial life, artificial neural networks, fuzzy logic, and robotics.

Dr. Sipper has published over 140 scientific papers, and is the author of two books: *Machine Nature: The Coming Age of Bio-Inspired Computing* and *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. He is an Associate Editor of the *IEEE Transactions on Computational Intelligence and AI in Games* and *Genetic Programming and Evolvable Machines*, an Editorial Board Member of *Memetic Computing*, and a past Associate Editor of the *IEEE Transactions on Evolutionary Computation*.

Dr. Sipper won the 1999 EPFL Latsis Prize, the 2008 BGU Toronto Prize for Academic Excellence in Research, and five HUMIE Awards—Human-Competitive Results Produced by Genetic and Evolutionary Computation (Gold, 2011; Bronze, 2009; Bronze, 2008; Silver, 2007; Bronze, 2005).

Preface

The application of computational intelligence techniques within the vast domain of games has been increasing at a breathtaking speed. Over the past several years my research group has produced a plethora of results in numerous games of different natures, evidencing the success and efficiency of evolutionary algorithms in general—and genetic programming in particular—at producing top-notch, human-competitive game strategies. At some point, as I surveyed (with a modicum of pride, I must admit) our results, a seed of an idea began to germinate in my mind, finally growing into a persistent inner voice that said: "Write a book".

I blithely ignored that voice.

But, spurred on by my bright students, the results kept accumulating and that inner voice grew rather clamorous. Then, in July 2011 we won a Gold HUMIE Award for our work on the game of FreeCell. Following a Silver and three Bronzes, this was our fifth such award, four of which were awarded for work done in the domain of games. The inner voice was now threatening to burst out of my head and write the book by itself.

And so I relented, the results of which act are now set before your eyes. I have attempted to lay down a coherent piece of writing that provides the reader with both a guide on the application of the evolutionary paradigm within the domain of games as well as a tour de force through the ludic landscape.

In all likelihood I could have found a traditional publisher for the book. However, I preferred that it be freely available online for easy, speedy access, with anyone wishing a handsome, printed copy able to order it inexpensively through `www.lulu.com`.

I wish to offer my heartfelt thanks to Yaniv Azaria, Amit Benbassat, Achiya Elyasaf, Ami Hauptman, John Koza, Natalio Krasnogor, Sean Luke, Yehonatan Shichel, and Lee Spector.

"You have to learn the rules of the game," said Albert Einstein. "And then you have to play better than anyone else."

So be it.

November 2011 *Moshe Sipper*

Contents

Part I

Rules of the Game

It was on a bitterly cold night and frosty morning, towards the end of the winter of '97, that I was awakened by a tugging at my shoulder. It was Holmes. The candle in his hand shone upon his eager, stooping face, and told me at a glance that something was amiss.

"Come, Watson, come!" he cried. "The game is afoot. Not a word! Into your clothes and come!"

—Arthur Conan Doyle
"The Adventure of the Abbey Grange"

Chapter 1

Setting the Pieces

Ever since the dawn of artificial intelligence (AI) in the 1950s games have been part and parcel of this lively field. In 1957, a year after the Dartmouth Conference that marked the official birth of AI, Alex Bernstein designed a program for the IBM 704 that played two amateur games of chess. In 1958, Allen Newell, J. C. Shaw, and Herbert Simon introduced a more sophisticated chess program (beaten in thirty-five moves by a ten-year-old beginner in its last official game played in 1960). Arthur L. Samuel of IBM spent much of the fifties working on game-playing AI programs, and by 1961 he had a checkers program that could play rather decently. In 1961 and 1963 Donald Michie described a simple trial-and-error learning system for learning how to play tic-tac-toe (or Noughts and Crosses) called MENACE (for Matchbox Educable Noughts and Crosses Engine).

Why do games attract such interest? “There are two principal reasons to continue to do research on games,” wrote Epstein [59]. “First, human fascination with game playing is long-standing and pervasive. Anthropologists have catalogued popular games in almost every culture... Games intrigue us because they address important cognitive functions... The second reason to continue game-playing research is that some difficult games remain to be won, games that people play very well but computers do not. These games clarify what our current approach lacks. They set challenges for us to meet, and they promise ample rewards.”

Studying games may thus advance our knowledge both in cognition and artificial intelligence, and, last but not least, games possess a competitive angle that coincides with our human nature, thus motivating both researcher and student alike.

During the past few years there has been an ever-increasing interest in the application of computational intelligence techniques in general, and evolutionary algorithms in particular, within the vast domain of games. I happened to stumble across this trend early on and decided to climb aboard the gamesome boat while it was still not too far from the harbor (Orlov et al. [134]; Sipper [166, 167]).

The year 2005 saw the first *IEEE Symposium on Computational Intelligence and Games*, which went on to become an annually organized event. The symposia's success and popularity led to their promotion from symposium to conference in 2010, and also spawned the journal *IEEE Transactions on Computational Intelligence and AI in Games* in 2009. In 2008, a journal showcase of evolutionary computation in games seemed to be the right thing to do—so we did it (Sipper and Giacobini [168]).

Clearly, there's a serious side to games.

1.1 Some Basic Definitions

Below we provide some basic definitions that have to do with games, based on Hauptman [77].

Perhaps the most basic kind of game is the *combinatorial game*: a two-player, perfect-information game, with no chance elements (Fraenkel [67]). In Parts II and IV of this book we will encounter *generalized combinatorial games*, a class of games defined by Hearn [84], characterized by: 1) a finite (albeit large) number of positions; 2) a number of players that may vary from zero to more than two; and 3) the possibility (and ease) of determining all legal moves from a given position.

Combinatorial games represent an excellent problem domain, for several reasons: 1) these games have specific sets of rules that constrain the possible behaviors of the players (i.e., legal moves); 2) they have a definite goal—to win the game—for the players to aim for; 3) rewards are given to players that best exhibit certain behaviors (game-playing strategies) under the constraints of finite resources (game pieces) that allow them to achieve the goal; and 4) these games have enough subtleties that allow a wide range of complex behaviors to emerge, represented by the diverse environment of players.

The problem of *solving* a combinatorial game is defined as determining the best move for a player, for *any* given game position. Solving a *specific* position, as opposed to solving the entire game, is the problem of determining the best sequence of moves starting from that position (the two problems are identical if this is the starting position of the game).

The most prominent solution methodology, which has become virtually synonymous with solving combinatorial games within the field of AI, is that of performing *search*. Junghanns [95] claimed that success in writing programs for games is strongly correlated with the understanding of how to make search work for them.

A *game graph* (or *game tree*, since loops are typically avoided) is defined as a graph representing *all* possible game configurations (the nodes), connected by legal moves (the edges). While it is theoretically possible to solve any combinatorial game by constructing the entire game tree, this remains infeasible in practice for most interesting games due to the tree's prohibitive size. Instead, an informed search algorithm,

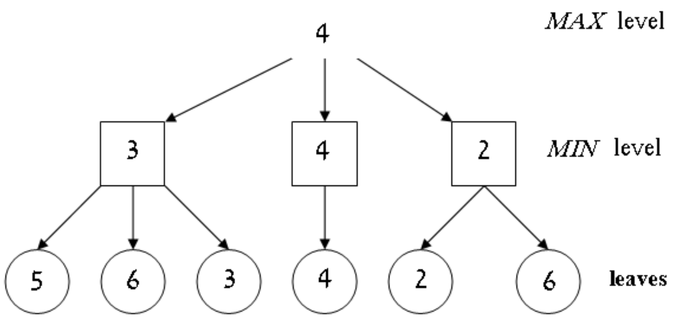


Figure 1.1 An example of a game-search tree with 3 levels. MAX nodes select the maximal value of their children, while MIN nodes select the minimal value of their children. In this example the root node, representing the player's current position, will select the move to the position indicated by the node labeled "4."

equipped with knowledge of the problem at hand (called *domain knowledge*), is typically applied in order to discover a path leading from the initial game configuration (or from a given one we wish to solve) to one of the winning positions (or *goals*) located within the game tree. The subtree traversed by the search algorithm is known as the *search tree*.

The *minimax algorithm* is perhaps the best-known algorithm for choosing the next move in a two-player game (Russell and Norvig [151]). It aims to find a good next move for the current player, such that no matter what the opponent does thereafter, the player's chances of winning the game are as high as possible. The algorithm performs a depth-first search (the depth is usually predetermined), applying an evaluation function to the leaves of the tree, and propagating these values upward according to the minimax principal: at MAX nodes, select the maximal value, and at MIN nodes—the minimal value. The value is ultimately propagated to the position from which the search had started (the root node). The evaluation function applied to a leaf computes a value that indicates how good it would be for a player to reach that position (Figure 1.1).

1.2 The Road Ahead

The book is divided into five parts.

Part I provides a short introduction to game research and a basic primer on evolutionary computation.

Part II presents research into the classic area of board games, beginning with checkers, and followed by chess (with some preliminary results on Reversi as well). My choice to embark upon our journey with these two games was not a haphazard one, but was made because the two are classic games with a time-honored history—both amongst players and within the field of AI—and they possess the typical attributes of adversarial games:

- Two players.
- Players alternate moves.

- Zero-sum: one player's loss is the other's gain.
- Perfect information: both players have access to complete information about the state of the game, with no information hidden from either player.
- Clear rules for legal moves, with no uncertain transitions involved.
- Well-defined outcomes (Win/Loss/Draw).
- No chance involved (e.g., using dice) .

This latter attribute—chance—is repealed in the final chapter of Part II, where we tackle the game of backgammon.

Part III further departs from the classic attributes listed above, delving, nonetheless, into a popular arena, whose outreach grows yearly: simulation games. These involve games that are not played between two players manipulating pieces on a board, but rather between many players in a simulated environment. They are not combinatorial in nature: players interact in a continuous manner rather than by taking turns, the number of moves is well-nigh infinite, and the game takes place in a virtual environment. We will encounter two such games: Robocode, wherein robotic tanks fight to destruction, and the Robot Auto Racing Simulator (RARS), wherein virtual car races are held.

Part IV showcases our ability to handle two difficult single-players games, also known as puzzles: Rush Hour, where one needs to clear a parking lot replete with vehicles such that a target car can make its way to the exit, and FreeCell, a highly popular card game available on any Windows-based computer.

Part V ends the book, with a discussion on the advantages of applying evolutionary algorithms—and in particular genetic programming—to games, and with some tips for the evolutionary computation practitioner wishing to tackle games.

The appendix delineates several interesting games explored in the context of undergraduate projects I have supervised over the years. These might represent directions for future research.

Each chapter on a game ends with a short discussion pertaining to the lessons gleaned from that particular chapter. In some places, in the interest of providing a fluid presentation, I have opted to omit some of the minute technical details (especially the various parameter settings), which the interested reader can find in the referenced papers.

From time to time we shall exchange the "hat" of evolutionary designer for that of a "molecular biologist" in an effort to gain a deeper understanding of our evolved code (more on this issue in Chapter 5).

As of 2004, one of the major annual events in the field of evolutionary algorithms, the Genetic and Evolutionary Computation Conference (GECCO), boasts a competition that awards prizes to human-competitive results: The "HUMIES". As noted at the competition site:[1] "Techniques of genetic and evolutionary computation are being increasingly applied to difficult real-world problems—often yielding results that are not merely interesting, but competitive with the work of creative and inventive humans."

Before moving on to introducing evolutionary algorithms, it is worth mentioning that several of the works presented in this book have won a HUMIE award:[2]

- A Bronze Award in 2005 for our work on chess, backgammon, and Robocode, presented in Chapters 4, 7, and 8.
- A Silver Award in 2007 for our work on the evolution of search algorithms for the game of chess, presented in Chapter 6.
- A Bronze Award in 2009 for our work on the Rush Hour puzzle, presented in Chapter 10.
- A Gold Award in 2011 for our work on the FreeCell puzzle, presented in Chapter 11.

[1] `http://www.human-competitive.org`

[2] We won an additional award in 2008 for work outside the area of games, wherein we applied a genetic algorithm to the automatic classification of defects in patterned silicon wafers (Glazer and Sipper [70]).

Chapter 2

Evolutionary Computation in a Nutshell

In the 1950s and 1960s several researchers began independently studying evolutionary systems with the idea that evolution could be used as a design and optimization tool for engineering and computational problems. Central to all the different methodologies was the notion of solving problems by evolving an initially random population of candidate solutions, through the application of operators inspired by natural genetics and natural selection, such that in time "fitter" (i.e., better) solutions emerge. In time, these efforts coalesced into the field of *Evolutionary Computation* or *Evolutionary Algorithms*. The field has experienced impressive growth over the past two decades, drawing in thousands of researchers who have put forward an astonishing array of arduous problems from diverse domain, all beautifully solved by some form or other of evolutionary algorithm.

This chapter is a brief and by no means complete introduction to the field. Many excellent texts are available for the reader wishing to gain a deeper and broader perspective, be it theoretical or practical (Eiben and Smith [52]; Koza [108]; Luke [118]; Poli et al. [140]; Sipper [165]; Tettamanzi and Tomassini [180]). Below, I will cover genetic algorithms—perhaps the most well-known subfield of evolutionary algorithms—and genetic programming, on which most of the work in this book is based.

2.1 Genetic Algorithms

A genetic algorithm (GA) is an iterative procedure that involves a population of individuals, each represented by a finite string of symbols, known as the *genome*, encoding a possible solution in a given problem space. This space, referred to as the *search space*, comprises all possible solutions to the problem at hand. Generally speaking, the genetic algorithm is applied to spaces that are too large to be exhaustively searched. The most common—though by no means only—form of genomic representation is that of a binary string, or bitstring (we shall stick to this representation in the explanation below).

The basic genetic algorithm proceeds as follows: an initial population of individuals is generated at random or heuristically. In every evolutionary step, known as a *generation*, the individuals in the current population are *decoded* and *evaluated* according to some predefined quality criterion, referred to as the *fitness*, or *fitness function*. To form a new population (the next generation), individuals are *selected* to become "parents" according to their fitness. Many selection procedures are currently in use, the three most well known being fitness-proportionate selection, rank selection, and tournament selection. Under fitness-proportionate selection, individuals are selected with a probability proportional to their relative fitness. Under rank selection, individuals are first sorted according to their fitness, from the highest-fit individual to the lowest-fit; the probability of an individual's being selected then depends on its position in the ranking, with higher-ranking individuals ascribed a higher selection probability. In tournament selection a number of individuals are chosen at random from the population, compared with each other, and the best of them is chosen to be the parent. While selection procedures differ in the amount of selection pressure, namely, the degree to which fit individuals can proliferate, they all share in common the basic notion that a higher-fit individual is more likely (to some degree or other) to be selected.

Selection alone cannot introduce any new individuals into the population, i.e., it cannot find new points in the search space; these are generated by genetically inspired operators, of which the most well known are *crossover* and *mutation*. Crossover is performed between two selected individuals (parents) by exchanging parts of their genomes (i.e., encodings) to form two new individuals—the offspring. In its simplest form substrings are exchanged after a randomly selected crossover point. The mutation operator is carried out by flipping bits at random with some (small) probability. A genetic algorithm is a stochastic iterative process that is not guaranteed to converge. The termination condition may be specified as some fixed, maximal number of generations or as the attainment of an acceptable fitness level. Algorithm 2.1 presents the standard genetic algorithm in pseudocode format while Figure 2.1 provides an example.

2.2 Genetic Programming

Genetic programming (GP) can be succinctly described as a genetic algorithm wherein the population contains programs rather than bitstrings. A popular representation is that of LISP (treelike) S-expressions, each such expression constructed

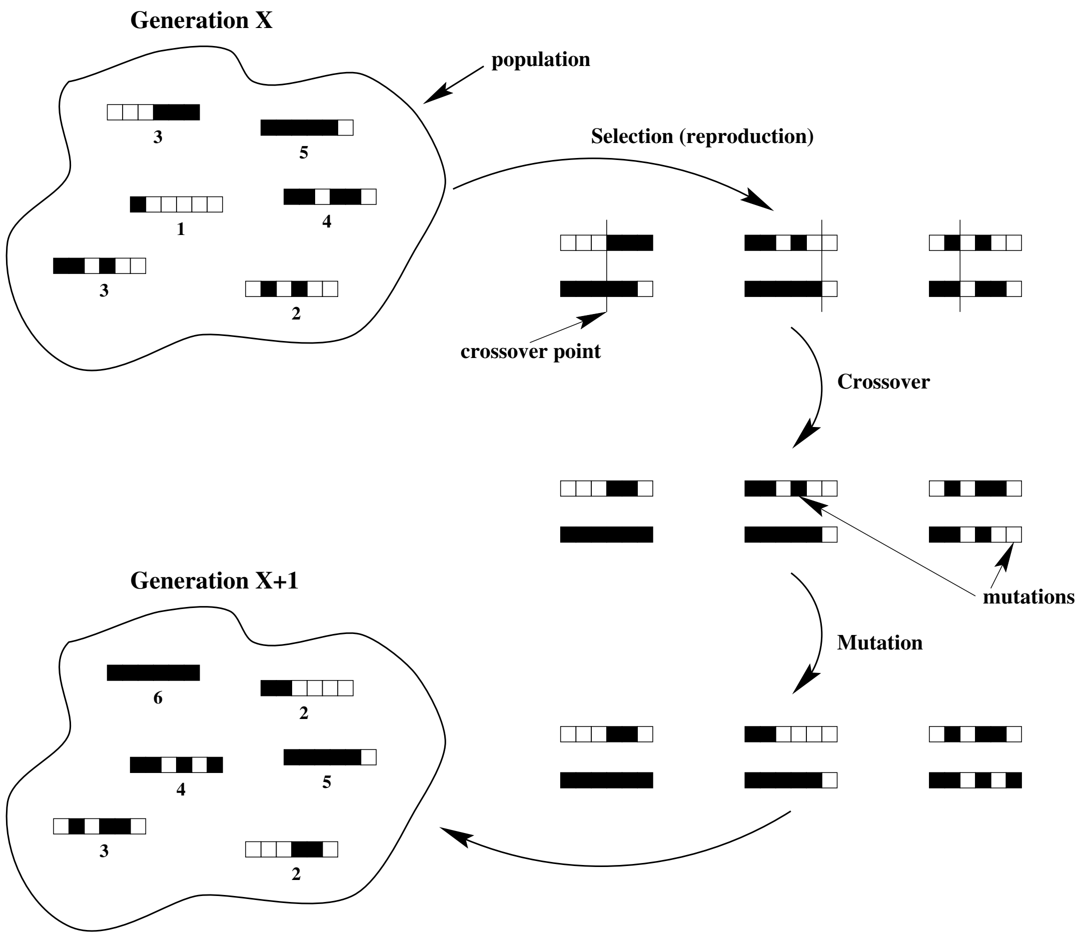


Figure 2.1 Demonstration of an evolutionary algorithm over one generation. The population consists of six individuals, each represented by an artificial genome containing six genes. A gene can take on one of two values (marked by black and white boxes). In this simple example, the fitness of an individual equals the number of black boxes (genes) in its genome (fitness values are displayed below the genomes). Selection is performed probabilistically: the higher an individual's fitness, the better its chance of being selected. Thus, some parents get selected more than once while others not at all. Each selected pair of parents is recombined to produce two offspring, an operation known as crossover. This is done by exchanging all genes to the right of a randomly selected crossover point. Mutation is then applied by flipping gene values with low probability. Note that application of the genetic operators on the population of generation X has yielded a perfect individual, with a fitness value of 6, at generation $X + 1$. Furthermore, the average fitness of the population, computed over all individuals, has increased.

Algorithm 2.1 Pseudocode of the basic genetic algorithm

```
1: $g \leftarrow 0$ // generation counter
2: Initialize population $P(g)$
3: Evaluate population $P(g)$ // compute fitness values
4: **while** not done **do**
5:     $g \leftarrow g + 1$
6:     Select $P(g)$ from $P(g-1)$
7:     Crossover $P(g)$
8:     Mutate $P(g)$
9:     Evaluate $P(g)$
10: **end while**
```

from *functions* and *terminals*. The functions are often arithmetic and logic operators that receive a number of arguments as input and compute a result as output; the terminals are zero-argument functions that serve both as constants and as sensors. Sensors are a special type of function that query the domain environment.[1]

The main flow of a GP run (Figure 2.2) is similar to that of a GA, with much of the difference between the two methods arising from the added complexity of GP's treelike representation. There are several ways for creating the initial random population, based on the recursive development of individual trees (Poli et al. [140] provide a succinct account of two of the simplest—and earliest—initialization methods, *full* and *grow*, and a widely used combination of the two known as *ramped half-and-half*). Once an initial population of randomly generated expressions has been created, each individual is evaluated in the domain environment and assigned a fitness value, after which genetic operators are applied, the common ones being:

- *Reproduction*: Copy one individual to the next generation with no modifications. The main purpose of this operator is to preserve a small number of good individuals.

- *Crossover* can be performed in many ways, one popular example of which is subtree crossover: randomly select an internal node in each of the two individuals and then swap the subtrees rooted at these nodes (Figure 2.3).

- *Mutation* can be performed in many ways, one popular example of which is subtree mutation: randomly select one node from the tree, delete the subtree rooted at that node, and then grow a new subtree (Figure 2.3).

In order to better control the structure of GP-produced programs, most of the works reported on in this book use *Strongly Typed Genetic Programming* (STGP) (Montana [122]; Poli et al. [140]). This method allows the user to assign a type to a tree

[1]Throughout this book we shall constantly encounter two types of trees, not to be conflated: game search trees (Chapter 1) and the trees representing GP individuals.

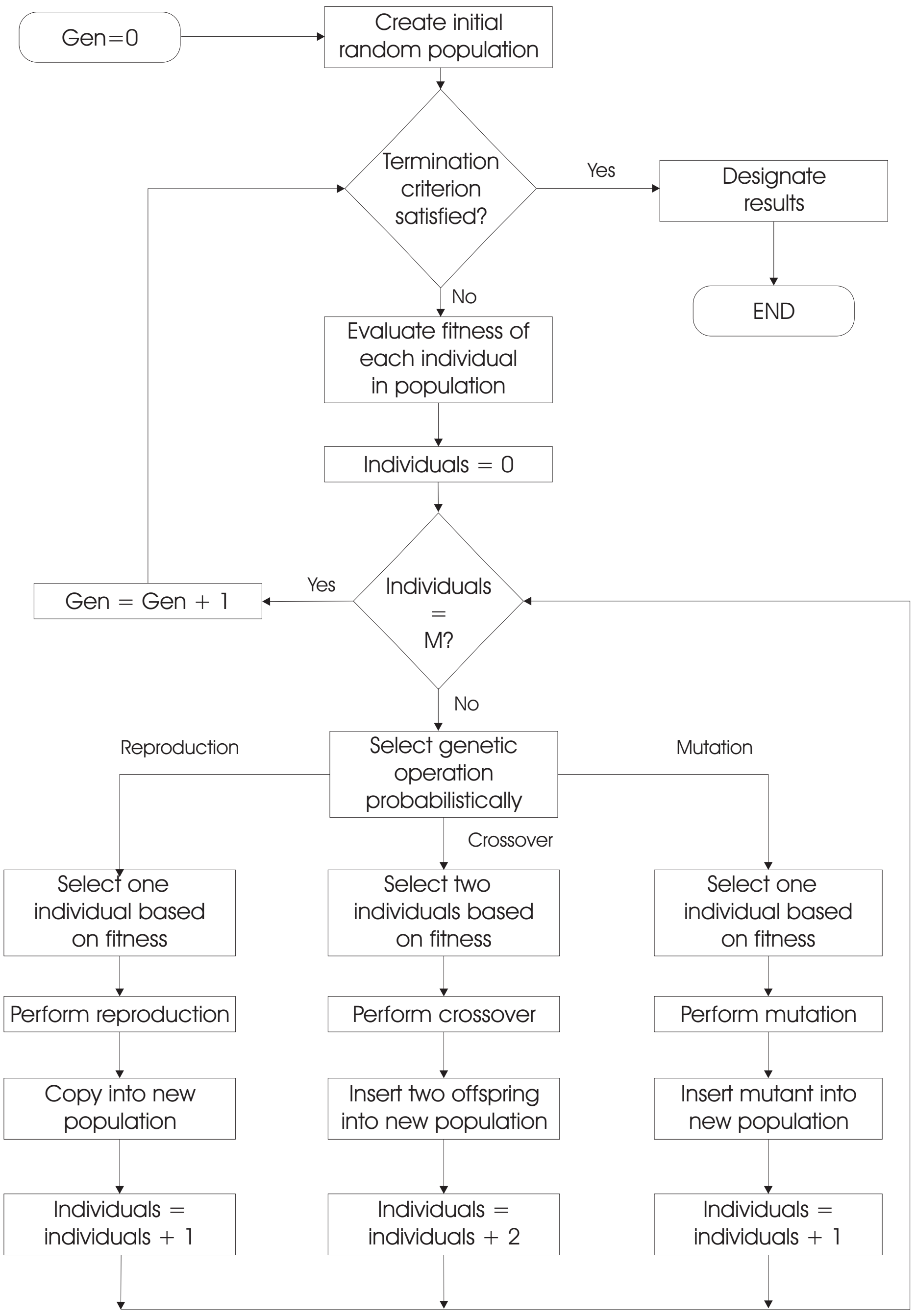

Gen=0
Create initial random population
Termination criterion satisfied?
Yes
Designate results
END
No
Evaluate fitness of each individual in population
Individuals = 0
Individuals = M?
Yes
Gen = Gen + 1
No
Reproduction
Select genetic operation probabilistically
Mutation
Crossover
Select one individual based on fitness
Select two individuals based on fitness
Select one individual based on fitness
Perform reproduction
Perform crossover
Perform mutation
Copy into new population
Insert two offspring into new population
Insert mutant into new population
Individuals = individuals + 1
Individuals = individuals + 2
Individuals = individuals + 1

Figure 2.2 Generic genetic programming flowchart (based on Koza [108]). M is the population size and Gen is the generation counter. The termination criterion can be the completion of a fixed number of generations or the discovery of a good-enough individual.

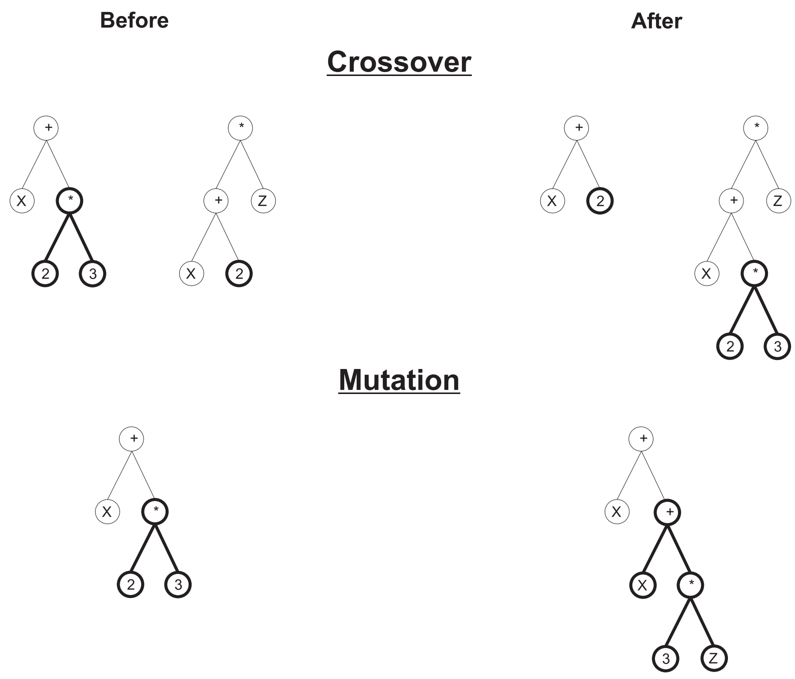


Figure 2.3 Top: Subtree crossover operator in GP. Two subtrees (marked in bold) are selected from the parents (left) and swapped to create two offspring (right). Bottom: Subtree mutation operator. A subtree (marked in bold) is selected from the parent individual (left), removed, and a new subtree is grown (right).

edge. Each function is assigned both a return type and a type for each of its arguments; each terminal is assigned a return type. Assigning more than one type per edge is also possible. All trees must be constructed according to these conventions, and only compatible types are allowed to interact (e.g., during crossover).

We also made heavy use of the ECJ software package of Luke [119].

Over the years, genetic programming has been shown to handle most (if not all) basic constructs of common programming languages, including functions, iteration, recursion, variables, and arrays. Recently, it has even proven possible to evolve programs in actual programming languages such as Java (Orlov and Sipper [131, 132, 133]).

Thus, using GP, one can evolve a complex computational structure able to solve a given (hard) problem. In particular, we will show throughout this book that (tree-based) GP is well suited to evolving winning game-playing strategies.

Part II

Board Games

A computer once beat me at chess, but it was no match for me at kick boxing.

—Emo Philips

Chapter 3

Lose Checkers

Developing players for board games has been part of AI research since the very beginning of the field. Board games have precise, easily formalized rules that render them easy to model in a programming environment. This chapter focuses on the game of lose checkers, which possesses the basic attributes of an adversarial game that we encountered in Chapter 1, including: deterministic, full-knowledge, perfect-information, and zero-sum.

We apply tree-based GP to evolving players for lose checkers (Benbassat and Sipper [17]). Our guide in developing our algorithmic setup, aside from previous research into games and GP, is nature itself. Evolution by natural selection is first and foremost nature's algorithm, and as such will serve as a source for ideas. Though it is by no means assured that an idea that works in the natural world will work in our synthetic environment, it can be seen as evidence that it might. We are mindful of evolutionary theory, particularly as pertaining to the gene-centered view of evolution. This view, presented by Williams [183] and expanded upon by Dawkins [48], focuses on the gene as the unit of selection. It is from this point of view that we consider how to adapt the ideas borrowed from nature into our synthetic GP environment.

At the end of this chapter we will present some preliminary results on two other board games: 10x10 American checkers and Reversi. While also classic adversarial games, they present, nonetheless, different challenges.

3.1 Checkers

Many variants of the game of checkers exist, several of them played by a great number of people (including tournament play). Practically all checkers variants are two-player games that contain only two types of pieces set on an $n \times n$ board. The most well-known variant of checkers is American checkers. It offers a relatively small search space (roughly 10^{20} legal positions compared to the 10^{43}–10^{50} estimated for chess) with a relatively small branching factor. It is fairly easy to write a competent[1] computer player for American checkers using minimax search and a simple evaluation function. The generic evaluation function for checkers is a piece differential that assigns extra value to kings on the board. This sort of player was used by Chellapilla and Fogel [37] to test their own evolved player.

American checkers shares its domain with another, somewhat less-popular variant of checkers, known as lose checkers. The basic rules of lose checkers are the same as American checkers (though the existence of different organizations may cause some difference in the peripheral rules). The objective, however, is quite different. A losing position for a player in American checkers is a winning position for that player in lose checkers and vice versa (i.e., one wins by losing all pieces or remaining with no legal move). Hlynka and Schaeffer [90] observed that, unlike the case of American checkers, lose checkers lacks an intuitive state evaluation function. Surprisingly (and regrettably) the inverse of the standard, piece differential-based checkers evaluation function is woefully ineffective. In some cases lose checkers computer players rely solely on optimized deep search and an endgame state database, having the evaluation function return a random value for states not in the database.

The years since Strachey [173] first designed an American checkers-playing algorithm saw a flurry of activity on checkers programs. Notable progress was made by Samuel [154, 155], who was the first to use machine learning to create a competent checkers program. Samuel's program managed to beat a competent human player in 1964. In 1989 a team of researchers from the University of Alberta led by Jonathan Schaeffer began working on an American checkers program called Chinook. By 1990 it was clear that Chinook's level of play was comparable to that of the best human players when it won second place in the U.S. checkers championship without losing a single game. Chinook continued to grow in strength, establishing its dominance (Schaeffer et al. [158]). In 2007, Schaeffer et al. [159] solved checkers and became the first to completely solve a major board game.

Board games attract considerable interest from AI researchers and the field of evolutionary algorithms is no exception to this rule. Over the years many board games have been tackled with the evolutionary approach. A GA with genomes representing artificial neural networks was used by Moriarty and Miikkulainen [123] to attack the game of Reversi (or Othello), resulting in a competent player that employed sophisticated mobility play. Artificial neural network-based American checkers players were evolved by Chellapilla and Fogel [36, 37] using a GA, their long runs resulting in expert-level play. In the subsequent chapters of this part of the book we will describe our own research, showing how GP can tackle chess and backgammon.

[1]By "competent" we mean players that show a level of skill in their play comparable to some human players (i.e., are not trivially bad) and yet do not exhibit the level of play of the strongest players (be they computer or human). As it is often hard to compare levels of play between different games, we find this fluid definition of "competence" to be suitable.

Node name	Return type	Return value
`ERC`	F	Ephemeral Random Constant
`False`	B	Boolean *false* value
`One`	F	1
`True`	B	Boolean *true* value
`Zero`	F	0

Table 3.1 Basic terminal nodes. F: floating point, B: Boolean.

To date, there has been limited research interest in lose checkers, all of it quite recent (Hlynka and Schaeffer [90]; Smith and Sailer [170]). This work concentrates either on search (Hlynka and Schaeffer [90]) or on finding a good evaluation function (Smith and Sailer [170]). Though both of these approaches give rise to strong players, they can also be seen as preliminary attempts that offer much room for improvement. The mere fact that it is difficult to handcraft a good evaluation function for lose checkers allows for the claim that any good evaluation function is in fact human competitive. If capable human programmers resort to having their evaluation function return random values, then any improvement on random is worth noting.

3.2 Evolutionary Setup

In our approach, the individuals in the population acted as board-evaluation functions, to be combined with a standard game-search algorithm (e.g., alpha-beta). The value they returned for a given board state was seen as an indication of how good that board state was for the player whose turn it was to play. The evolutionary algorithm was written in Java and is based on the Strongly Typed GP framework (Chapter 2). The two types we implemented were the Boolean type and a floating-point type. We also offered support for multi-tree individuals, as will be discussed below. We implemented the basic crossover and mutation operators described by Koza [108]. On top of this, another form of crossover was implemented—which we designated "one-way crossover"—as well as a local mutation operator. The setup is detailed below.

3.2.1 Basic terminal nodes

We implemented several basic domain-independent terminal nodes, presented in Table 3.1.

The only node in Table 3.1 that requires further explanation is the `ERC` (Ephemeral Random Constant). The concept of `ERC` was first introduced by Koza [108]. An `ERC` returns a value that is decided randomly when the node is created, and thereupon does not change unless an `ERC` mutation operator is used. We set the return value of an `ERC` to a random value in the range $[-5, 5)$.

Table 3.2 Domain-specific terminal nodes that deal with board characteristics.

Node name	Type	Return value
`EnemyKingCount`	F	The enemy's king count
`EnemyManCount`	F	The enemy's man count
`EnemyPieceCount`	F	The enemy's piece count
`FriendlyKingCount`	F	The player's king count
`FriendlyManCount`	F	The player's man count
`FriendlyPieceCount`	F	The player's piece count
`KingCount`	F	`FriendlyKingCount` − `EnemeyKingCount`
`ManCount`	F	`FriendlyManCount` − `EnemeyManCount`
`PieceCount`	F	`FriendlyPieceCount` − `EnemeyPieceCount`
`KingFactor`	F	King factor value
`Mobility`	F	The number of plies available to the player

Table 3.3 Domain-specific terminal nodes that deal with square characteristics. They all receive two parameters—X and Y—the row and column of the square, respectively.

Node name	Type	Return value
`IsEmptySquare`(X,Y)	B	True iff square empty
`IsFriendlyPiece`(X,Y)	B	True iff square occupied by friendly piece
`IsKingPiece`(X,Y)	B	True iff square occupied by king
`IsManPiece`(X,Y)	B	True iff square occupied by man

3.2.2 Domain-specific terminal nodes

The domain-specific terminal nodes are listed in two tables: Table 3.2 shows nodes describing characteristics that have to do with the board in its entirety, and Table 3.3 shows nodes describing characteristics of a certain square on the board.

The `KingFactor` terminal (Table 3.2) is a constant set to 1.4. It signifies the ratio between the value of a king and the value of a man in material evaluation of boards in American checkers. It was included in some of the runs and also played a role in calculating the return value of the piece-count nodes. A king-count terminal returns the number of kings the respective player has, or a difference between the two players' king counts. A man-count terminal returns the number of men the respective player has, or a difference between the two players' man counts. In much the same way a piece-count node returns the number of the respective player's men on the board and adds to it the number of that player's kings multiplied by the king factor. Again there is a node that returns the difference between the two players' piece counts.

The mobility node was a late addition that greatly increased the playing ability of the fitter individuals in the population. This terminal, returning the number of plies[2] available to the player, allowed individuals to more easily adopt a mobility-based,

[2]In two-player sequential games, a *ply* refers to one turn taken by one of the players. The word is used to clarify what is meant when one might otherwise say "turn". In standard chess and checkers terminology, one *move* consists of a turn by each player; therefore a ply is a half-move.

Table 3.4 Function nodes. F_i: floating-point parameter, B_i: Boolean parameter.

Node name	Type	Return value
AND(B_1,B_2)	B	Logical AND of parameters
LowerEqual(F_1,F_2)	B	True iff $F_1 \leq F_2$
NAND(B_1,B_2)	B	Logical NAND of parameters
NOR(B_1,B_2)	B	Logical NOR of parameters
NOTG(B_1,B_2)	B	Logical NOT of B_1
OR(B_1,B_2)	B	Logical OR of parameters
IFT(B_1,F_1,F_2)	F	F_1 if B_1 is true and F_2 otherwise
Minus(F_1,F_2)	F	$F_1 - F_2$
MultERC(F_1)	F	F_1 multiplied by preset random number
NullJ(F_1,F_2)	F	F_1
Plus(F_1,F_2)	F	$F_1 + F_2$

game-state evaluation function.

The square-specific nodes all returned Boolean values. They were very basic, and represented no expert human knowledge about the game. In general, one could say that all the domain-specific nodes used little in the way of human knowledge about the game, with the possible exception of the king factor and mobility terminals. This goes against what has traditionally been done when GP is applied to board games (as we will see in the subsequent chapters). This is partly due to the difficulty in finding useful board attributes for evaluating game states in lose checkers, but there is another, more fundamental, reason. Not introducing game-specific knowledge into the domain-specific nodes meant the GP algorithm defined was itself not game-specific, and thus more flexible (it is worth noting that mobility is a universal principle in playing board games, and therefore the mobility terminal can be seen as not being game-specific). As defined, the algorithm can be used on the two games played in the American checkers domain. A very slight change in the genetic program and the appropriate game program can render our system applicable to any variant of checkers (the number of conceivable checkers variants that are at least computationally interesting is virtually unlimited). Our system can also be applied with little adaptation to other board games that have no more than two types of pieces, such as Reversi (which we will encounter at the end of this chapter), and possibly even Go, the holy grail of AI board-game research.

3.2.3 Function nodes

We defined several basic domain-independent functions, presented in Table 3.4, and no domain-specific functions.

The functions implemented include logic functions, basic arithmetic functions, one relational function, and one conditional statement. The conditional expression rendered natural control flow possible and allowed us to compare values and return

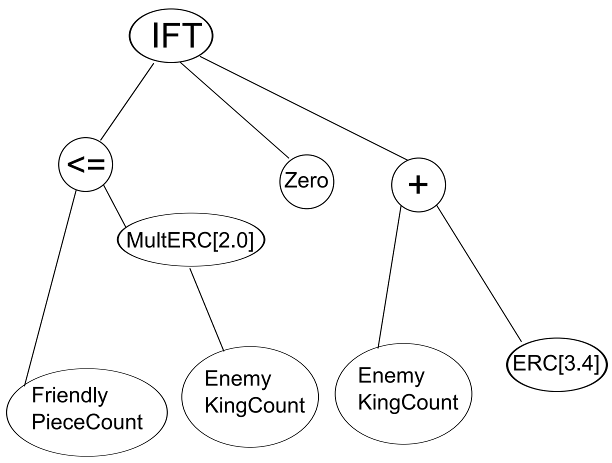


Figure 3.1 Sample tree for lose checkers board evaluation.

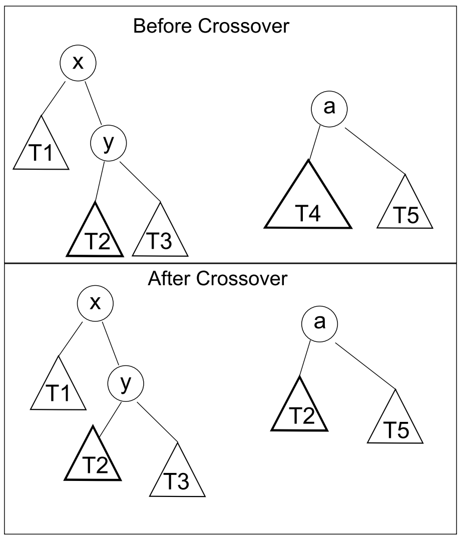


Figure 3.2 One-way crossover: Subtree T2 in donor tree (left) replaces subtree T4 in receiver tree (right). The donor tree remains unchanged.

a value accordingly. In Figure 3.1 we see an example of this. The tree depicted in the figure returns 0 if the friendly piece count is less than double the number of enemy kings on the board, and the number of enemy kings plus 3.4 otherwise (3.4 is an `ERC`).

3.2.4 One-way crossover

One-way crossover, as opposed to the typical two-way version, does not consist of two individuals swapping parts of their genomes, but rather of one individual inserting a copy of part of its genome into another individual, without receiving any genetic information in return. This can be seen as akin to an act of “aggression”, where one individual pushes its genes upon another, as opposed to the generic two-way crossover operators that are more cooperative in nature. In our case, the one-way crossover is done by randomly selecting a subtree in both participating individuals, and then inserting a copy of the selected subtree from the first individual in place of the selected subtree from the second individual. An example is shown in Figure 3.2.

This type of crossover operator is uni-directional. There is a donor and a receiver of genetic material. This directionality can be used to make one-way crossover more than a random operator, as in our case where we opted to select the individual with higher fitness as the donor. This sort of nonrandom genetic operator favors the fitter individuals as they have a better chance of surviving it. Algorithm 3.1 shows the pseudocode representing how crossover is handled in our system. As can be seen, one-way crossover is expected to be chosen about half the time on average, giving the fitter individuals a survival advantage, but the fitter individuals can still be altered due to the standard two-way crossover.

Algorithm 3.1 Crossover

```
1: Randomly choose two different previously unselected individuals from the population for crossover: $I1$ and $I2$
2: if $I1.Fitness \geq I2.Fitness$ then
3:     Perform one-way crossover with $I1$ as donor and $I2$ as receiver
4: else
5:     Perform two-way crossover with $I1$ and $I2$
6: end if
```

Using the vantage point of the gene-centered view of evolution, it is easier to see the logic of crossover in our system. In a gene-centered world we look at genes as competing with each other, the more effective ones out-reproducing the rest. This, of course, should already happen in a framework using the generic two-way crossover alone. Using one-way crossover, as we do, just strengthens this trend. In one-way crossover, the donor individual pushes a copy of one of its genes into the receiver's genome at the expense of one of the receiver's own genes. The individuals with high fitness that are more likely to get chosen as donors in one-way crossover are also more likely to contain more good genes than the less-fit individuals that get chosen as receivers. This genetic operator thus causes an increase in the frequency of the genes that lead to better fitness.

Both types of crossover used have their roots in nature. Two-way crossover is often seen as analogous to sexual reproduction. One-way crossover also has an analog in nature in the form of lateral gene transfer that exists in bacteria.

3.2.5 Local mutation

In order to afford local mutation with limited effect we modified the GP setup as follows: To each node returning a floating-point value we added a floating-point variable (initialized to 1) that served as a factor. The return value of the node was the normal return value multiplied by this factor. A local mutation would then be a small change in the node's factor value.

Whenever a node returning a floating-point value was chosen for mutation, a decision had to be made on whether to activate the traditional tree-building mutation operator, or the local factor mutation operator. Toward this end we designated a run parameter that determined the probability of opting for the local mutation operator.

3.2.6 Explicitly defined introns

In natural living systems not all DNA has a phenotypic effect. This non-coding DNA, sometimes referred to as junk DNA, is prevalent in virtually all eukaryotic genomes. In GP, so-called introns are areas of code that do not affect survival and reproduction (usually this can be replaced with "do not affect fitness"). In the context of tree-based GP the term "areas of code" applies to subtrees.

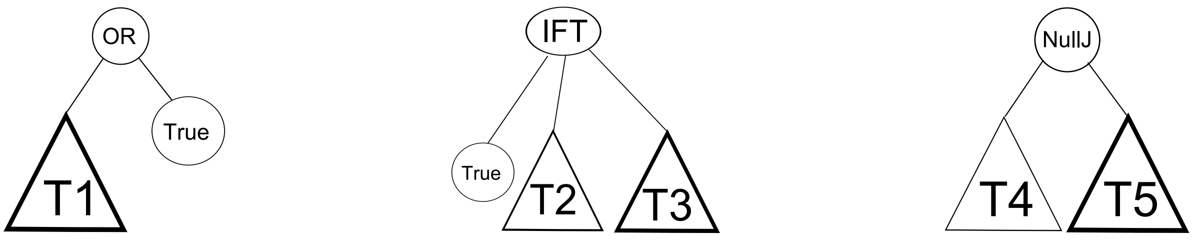
Introns occur naturally in GP, provided that the function and terminal sets allow for it. As bloat progresses, the number of nodes that are part of introns tends to increase. Luke [117] distinguished two types of subtrees that are sometimes referred to as introns in the literature:

- *Unoptimized code*: Areas of code that can be trivially simplified without modifying the individual's functionality, but not replaced with just anything.
- *Inviable code*: Subtrees that cannot be replaced by anything that can possibly change the individual's functionality.

Luke focused on inviable introns and we will do the same because unoptimized code seems to cast too wide a net and wander too far from the original meaning of the term "intron" in biology. We also make another distinction between two types of inviable-code introns:

- *Live-code introns*: Subtrees that cannot be replaced by anything that can possibly change the individual's functionality, but may still generate code that will run at some point.
- *Dead-code introns*: Subtrees whose code is never run.

Figure 3.3 exemplifies our definitions of introns in GP: $T1$ is a live-code intron, while $T3$ and $T5$ are dead-code introns. $T1$ is calculated when the individual is executed, but its return value is not relevant because the logical OR with a *true* value always returns a *true* value. $T3$, on the other hand, never gets calculated because the `IFT` function node above it always turns to $T2$ instead. $T3$ is thus dead code. Similarly, $T5$ is dead code because the `NullJ` function returns a value that is independent of its second parameter.



(a) Live-code intron (b) Implicit dead-code intron (c) Explicitly defined dead-code intron

Figure 3.3 Examples of different types of introns in GP trees.

Explicitly defined introns (EDIs) in GP are introns that reside in an area of the genome specifically designed to hold introns. As the individual runs it will simply ignore these introns. In our system EDIs exist under every `NullJ` and `NotG` node. In both functions the rightmost subtree does not affect the return value in any way. This means that every instance of one of these function nodes in an individual's tree defines an intron, which is always of the dead-code type. In Figure 3.3, $T5$ differs from $T3$ in that $T5$ is known to be an intron the moment `NullJ` is reached, and therefore the program can take it into account. When we converted individuals into C code the EDIs were simply ignored, a feat that could be accomplished with ease as they were dead-code introns that were easy to find.

Nordin et al. [129] explored EDIs in linear GP, finding that they tended to improve fitness and shorten runtime, as EDIs allow the evolutionary algorithm to protect important functional genes and save runtime used by live-code introns. Earlier work showed that using introns was also helpful in GAs (Levenick [114]).

3.2.7 Multi-tree individuals

Preliminary runs with a single tree per individual taught us a fair bit about the sort of solutions that tended to evolve. An emerging characteristic seemed to be that the majority of the population quickly focused on one design choice for the top of the tree and kept it, unchanged, in effect developing "regulatory" genes. The `IFT` function was a popular pick for the top part of the tree. This makes sense as the conditional expression is a good way to differentiate cases. We decided to reinforce this trend, using multiple trees to simulate the run of a single tree with preset nodes at its top. We chose to use ten trees per individual, all returning floating-point values. The values returned by the ten trees were manipulated to simulate the behavior of the tree shown in Figure 3.4.

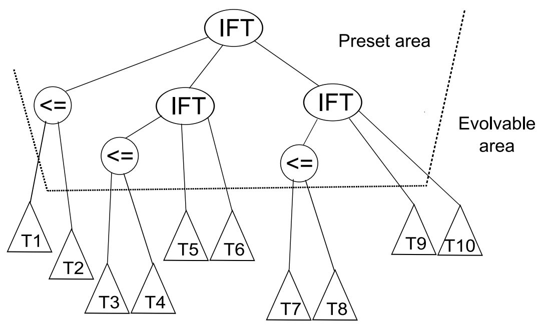


Figure 3.4 A multi-tree individual in our system with preset (non-evolving) area above the line and 10 evolving subtrees, $T_1, \ldots, T_{10}$, below the line.

3.2.8 Fitness calculation

Fitness calculation was carried out in the manner described in Algorithm 3.2. Evolving players face two types of opponents: external "guides" (described below) and their own cohorts in the population. The latter method of evaluation is known as coevolution (Runarsson and Lucas [150]), and is referred to below as the coevolution round.

Algorithm 3.2 Fitness evaluation

```
   // Parameter: GuideArr—array of guide players
1: for $i \leftarrow 1$ to GuideArr.length do
2:    for $j \leftarrow 1$ to GuideArr[i].NumOfRounds do
3:       Every individual in population deemed fit enough plays
         GuideArr[i].roundSize games against guide $i$
4:    end for
5: end for
6: Every individual in the population plays CoPlayNum games as black against
   CoPlayNum random opponents in the population
7: Assign 1 point per every game won by the individual, and 0.5 points per drawn
   game
```

The method of evaluation described requires some parameter setting, including the number of guides, their designations, the number of rounds per guide, and the number of games per round, for the guides array *GuideArr* (players played X rounds of Y games each). The algorithm also needs to know the number of co-play opponents for the coevolutionary round. In addition, a parameter for game point value for different guides, as well as for the coevolutionary round, was also required. This allowed us to ascribe greater significance to certain rounds than to others. Tweaking these parameters allows for different setups.

Guide-play rounds We implemented two types of guides: A random player and an alpha-beta player. The random player chose a move at random and was used to test initial runs. The alpha-beta player searched up to a preset depth in the game tree and used an evaluation function returning a random value for game states in which there was no clear winner (in states where win, loss, or draw was evident, the evaluation function returned an appropriate value). To save time, not all individuals were chosen for each game round. We defined a cutoff for participation in a guide-play round. Before every guide-play round began, the best individual in the population was found. Only individuals whose fitness trailed that of the best individual by no more than the cutoff value got to play. When playing against a guide each player in the population received 1 point added to its fitness for every win, and 0.5 points for every draw.

Coevolution rounds In a co-play round, each member of the population in turn played Black in a number of games equal to the parameter *CoPlayNum* against *CoPlayNum* random opponents from the population playing White. The opponents were chosen in a way that ensured that each individual also played exactly *CoPlayNum* games as White. This was done to make sure that no individuals received a disproportionately high fitness value by being chosen as opponents more times than others. When playing a co-play game, as when playing against a guide, each player in the population received 1 point added to its fitness for every win, and 0.5 points for every draw.

3.2.9 Selection and procreation

The change in population from one generation to the next was divided into two stages: A selection stage and a procreation stage. In the selection stage the parents of the next generation were selected (some more than once) according to their fitness. In the procreation stage, genetic operators were applied to the parents in order to create the next generation.

Selection was done by the following simple method: Of two individuals chosen at random, a copy of the fitter individual was selected as a parent for the procreation stage (this is known as tournament selection with tournament size 2). The pseudocode for the selection process is given in Algorithm 3.3.

Algorithm 3.3 Selection

```
1: repeat
2:    Randomly choose two different individuals from population : $I1$ and $I2$
3:    if $I1.Fitness > I2.Fitness$ then
4:       Select a copy of $I1$ for parent population
5:    else
6:       Select a copy of $I2$ for parent population
7:    end if
8: until number of parents selected is equal to original population size
```

The crossover and mutation probabilities were denoted p_{xo} and p_m, respectively . Every individual was chosen for crossover (with a previously unchosen individual) with probability p_{xo} and self-replicated with probability $1 - p_{xo}$. The implementation and choice of specific crossover operator was as in Algorithm 3.1. After crossover every individual underwent mutation with probability p_m. This is slightly different from traditional GP, where an individual undergoes either mutation or crossover but not both (Figure 2.2). However our system is in line with the GA tradition where crossover and mutation act independently of each other.

3.2.10 Players

Our system supported two kinds of GP players. The first kind of player examined all legal moves and used the GP individual to assign scores to the different moves, choosing the one that scored highest. This method is essentially a minimax search of depth 1. The second kind of player mixed GP game-state evaluation with a minimax search. It used the alpha-beta search algorithm implemented for the guides, but instead of evaluating non-terminal states randomly it did so using the GP individual. This method added search power to our players, but resulted in a program wherein deeper search created more game states to be evaluated, taking more time.

3.2.11 Run parameters

The run parameters were as follows:

- Number of generations: 100–200.
- Population size: 100–200.
- Crossover probability: typically 0.8.
- Mutation probability: 0.1, or 0.2 if local mutation was used.
- Local mutation ratio: 0 or 0.5.
- Maximum depth of GP tree: 15 without search, 12–14 with search of depth 3, 10 with search of depth 4.
- Designation of player to serve as benchmark for the best player of each generation.
- Designation of search depth used by GP players during run.

Table 3.5 Relative levels of play for different benchmark (guide) players. Here and in the subsequent tables, $\alpha\beta i$ refers to an alpha-beta player using a search depth of i and a random evaluation function (unless the state reached can easily be determined to be win, loss, or draw).

1st Player	2nd Player	1st Player win ratio
$\alpha\beta2$	random	0.9665
$\alpha\beta3$	$\alpha\beta2$	0.8502
$\alpha\beta5$	$\alpha\beta3$	0.82535
$\alpha\beta7$	$\alpha\beta5$	0.8478
$\alpha\beta3$	$\alpha\beta4$	0.76925
$\alpha\beta3$	$\alpha\beta6$	0.6171
$\alpha\beta3$	$\alpha\beta8$	0.41270
$\alpha\beta5$	$\alpha\beta6$	0.7652
$\alpha\beta5$	$\alpha\beta8$	0.55620

3.3 Results

We divided the experiments into two sets, the first using no search, the second with search and mobility incorporated into the evolutionary algorithm. The same hand-crafted machine players that were used as guides in fitness evaluation also served as the benchmark players. Before beginning the evolutionary experiments, we first evaluated our guide players by testing them against each other in matches of 10,000 games (with players alternating between playing Black and White). For search depths greater than 2 we focused on guide players using odd search depths, as those using even search depths above 2 proved weaker than the players using lower odd search depths. Every one of the alpha-beta players participated in a match with a weaker player and a stronger player (except for the strongest searcher tested). Table 3.5 presents the results of these matches. Note the advantage of odd search depth over (higher) even search depth.

In all evolutionary runs that follow, evolution ran on a single core of a dual-core Xeon 5140 2.33GHz processor. All runs took from a few days to a week (we limited runtimes to 160 hours).

3.3.1 No search

The first set of experiments involved no search (meaning a lookahead of 1). When tested against the random player, the evolved move evaluators showed a significant improvement in level of play, as can be seen in Table 3.6.

Table 3.6 Results of top runs using a lookahead of 1 (i.e., no search). Here and in subsequent tables: the benchmark (post-evolutionary) score of the evolved *Player* is calculated over 1000 games against the respective *Benchmark Opponent*; the benchmark score is the number of games won plus half the number of games drawn; xRand stands for x games against the random player; yCo stands for y games of co-play. *Benchmark Opponent* herein is the random player.

Run Identifier	Fitness Evaluation	Benchmark Score
r00032A	20Rand+30Co	554.0
r00033B	30Rand+5Co	634.0
r00034A	30Rand+5Co	660.0
r00035B	30Rand+5Co	721.0
r00036A	30Rand+10Co	705.5
r00037B	30Rand+10Co	666.5

Table 3.7 Results of top runs using shallow search. *Player* uses $\alpha\beta$ search of depth 3 coupled with evolved evaluation function, while *Benchmark Opponent* uses $\alpha\beta$ search of depth 3 coupled with a random evaluation function.

Run Identifier	Fitness Evaluation	Benchmark Score
r00044A	50Co	744.0
r00046A	50Co	698.5
r00047A	50Co	765.5
r00048A	50Co	696.5
r00049A	50Co	781.5
r00056A	50Co	721.0
r00057A	50Co	786.5
r00058A	50Co	697.0
r00060A	50Co	737.0
r00061A	50Co	737.0

3.3.2 Shallow search

Using shallow search produced better results. Experiments showed that a search depth of 3 was the best choice. This allowed for improved results, while deeper search caused runs to become too slow. In order to save time the maximum GP-tree depth was more strongly restricted, and code optimizations were added to improve runtime. Attempts to use the multi-tree design with search did not yield good results, and this design was abandoned in favor of the standard single-tree version. It is quite possible that another, less wasteful, multi-tree design, would have succeeded more. The results are shown in Table 3.7.

As we saw above, random evaluation coupled with odd search depth is a powerful heuristic. We tested some of our best evolved players against players using random evaluation with even search depth values greater than 3. The results of these tests

Table 3.8 Results of top runs playing against players using random evaluation and various search depths, focusing on even depths greater than 3. *Player* uses $\alpha\beta$ search of depth 3 coupled with evolved evaluation function.

Run Identifier	vs. $\alpha\beta3$	vs. $\alpha\beta4$	vs. $\alpha\beta6$	vs. $\alpha\beta8$
r00044A	744.0	944.5	816.0	758.0
r00047A	765.5	899.0	722.5	476.0
r00049A	781.5	915.0	809.0	735.5
r00057A	786.5	909.0	745.5	399.5
r00060A	737.0	897.0	627.0	408.5
r00061A	737.0	947.0	781.5	715.5

are given in Table 3.8. As evident, the evolved players, some more than others, have acquired a playing proficiency that allows them to outmaneuver players employing a far deeper search and taking far greater time resources. It is worth noting that the player $\alpha\beta8$, which uses a search depth of 8, is decisively outperformed by three different players (that only search to a depth of 3). This player (as Table 3.5 clearly indicates) is stronger than the $\alpha\beta3$ player that served as the evolved players' original benchmark.

3.3.3 Expanding search depth

In order to produce good results against stronger players without incurring high runtime costs, we took some of the existing players evolved using shallow search of depth 3 and modified them to use deeper search with the same (evolved) evaluation function. We then tested the modified players against a benchmark player using deeper search, hoping that expanding search depth would preserve the quality of the evaluation function (Chellapilla and Fogel [36] succeeded in doing this with American checkers). In these select runs we chose the evolved individual that had the best benchmark score against the $\alpha\beta3$, altered its code to extend its search depth to 5, and ran a 1000-game match between the altered, deeper-search player and $\alpha\beta$ random-evaluation players. The results of this experiment are presented in Table 3.9.

3.3.4 Using deeper search

After optimizing the code to some extent we used a search depth of 4 in our evolutionary runs. To try and prevent exceedingly long runtimes population size, number of generations, and maximum tree depth were severely limited (to 50, 70, and 10, respectively). Table 3.10 presents the results. We see that players have evolved to beat the strong $\alpha\beta5$ player but at the cost of some overspecialization against that player.

Table 3.9 Results of top evolved players after depth expansion. Note that *Player* was evolved using a search depth of 3, but is here tested with its depth extended to 5, versus the random-evaluation $\alpha\beta5$, $\alpha\beta6$, and $\alpha\beta8$ *Benchmark Opponents*.

Run Identifier	vs. $\alpha\beta5$	vs. $\alpha\beta6$	vs. $\alpha\beta8$
r00044A	438.5	774.0	507.5
r00047A	437.5	807.0	482.5
r00049A	420.5	856.0	449.0
r00057A	494.5	874.5	463.5
r00060A	459.0	834.0	583.5
r00061A	483.5	967.0	886.0

Table 3.10 Results of top runs playing against players using random evaluation and various search depths. *Player* uses $\alpha\beta$ search of depth 4 coupled with evolved evaluation function.

Run Identifier	Fitness Evaluation	vs. $\alpha\beta5$	vs. $\alpha\beta6$	vs. $\alpha\beta8$
r00064A	20$\alpha\beta$5+20Co	582.0	603.5	395.0
r00065A	20$\alpha\beta$5+20Co	537.0	782.5	561.5
r00066A	20$\alpha\beta$5+20Co	567.0	757.5	483.5
r00067A	20$\alpha\beta$5+20Co	598.5	723.0	385.5
r00068A	20$\alpha\beta$5+20Co	548.0	787.0	524.0
r00069A	20$\alpha\beta$5+20Co	573.5	715.5	523.0
r00070A	20$\alpha\beta$5+20Co	577.0	691.5	476.0
r00071A	20$\alpha\beta$5+20Co	551.5	582.5	401.5

3.3.5 Reversi and 10x10 checkers

Having obtained such good results for lose checkers we asked ourselves whether, with little adjustment, we could tackle two other board games: 10x10 checkers and Reversi (Benbassat and Sipper [18]). An expansion of the rule set of American Checkers (the standard, non-losing variant) to a 10x10 board creates a significantly higher branching factor and higher game complexity (due, in addition to the increased branching factor, to the existence of 30 pieces instead of 24 on the opening board). Reversi, also known as Othello, is a popular game with a rich research history (Chong et al. [39]; Eskin and Siegel [60]; Lee and Mahajan [113]; Moriarty and Miikkulainen [123]; Rosenbloom [149]). Though a board game played on an 8x8 board, it differs widely from the checkers variants in that it is a piece-placing game rather than a piece-moving game. In Reversi the number of pieces on the board increases during play, rather than decreasing as with checkers. The number of moves (not counting the rare pass moves) in Reversi is limited by the board's size, making it a short game.

For these two games we built opponent players by using the simple yet effective method of material evaluation (piece counting) to evaluate board states. We had hand-crafted players randomly alternate between two different material evaluation functions in order to generate a playing strategy that was not entirely predictable.[3] The average score of $\alpha\beta2$ playing 10x10 checkers against the random player was 0.99885, and $\alpha\beta3$ scored an average of 0.5229 against $\alpha\beta2$.

As in most evolutionary computation systems, so in ours, the lion's share of runtime is spent on calculating fitness values. As discussed above, fitness calculations in our case were based on an individual's playing multiple games, all of whose results were independent of all other games. This opens the door for parallelism, as the games used in determining fitness can be run in parallel on different cores. Indeed, once we began running parallel fitness evaluations, runtimes were curtailed substantively. Considering the growing trend in recent years towards multi-core and multi-processor parallel computation, the easy applicability of parallelism in evolutionary computation is of significant value when considering whether or not to use the evolutionary approach (see also Chapter 7).

This is still work in progress. Our preliminary best runs for 10x10 checkers are presented in Table 3.11. As the table demonstrates, our players were able to outperform the $\alpha\beta3$ player using a search depth of 2 (runs 92–95), and to overwhelm it using a search depth of 3 (runs 84–85).

The average score of $\alpha\beta2$ playing Reversi against the random player was 0.8471, $\alpha\beta3$ scored an average of 0.6004 against $\alpha\beta2$, $\alpha\beta5$ scored an average of 0.7509 against $\alpha\beta3$, and $\alpha\beta7$ scored an average of 0.7509 against $\alpha\beta5$.

[3]All functions used were material evaluation functions (friendly piece count minus opponent piece count). The difference was the bonus given for special pieces: In checkers the king bonus was different, while in Reversi the corner bonus was different.

Table 3.11 10x10 checkers: Results of top runs using shallow search. *Player* uses $\alpha\beta$ search of depth 3 (runs 84, 85) or 2 (runs 92–95) coupled with evolved evaluation function, while *Benchmark Opponent* uses $\alpha\beta$ search of depth 3 coupled with a material evaluation function.

Run Identifier	Fitness Evaluation	Benchmark Score
r00084	50Co	889.0
r00085	50Co	927.0
r00092	25Co	732.0
r00093	25Co	615.5
r00094	25Co	554.0
r00095	25Co	631.0

Table 3.12 Reversi: Results of top runs using shallow search. *Player* uses $\alpha\beta$ search of depth 4 coupled with evolved evaluation function, while *Benchmark Opponent* uses $\alpha\beta$ search of depths 5 and 7 coupled with a material evaluation function.

Run Identifier	Fitness Evaluation	Benchmark Score vs. $\alpha\beta5$	Benchmark Score vs. $\alpha\beta7$
r00100	25Co	875.0	758.5
r00101	25Co	957.5	803.0
r00102	40Co	942.5	640.5
r00103	40Co	905.5	711.5
r00108	40Co	956.0	760.0
r00109	40Co	912.5	826.0
r00110	40Co	953.5	730.5
r00111	40Co	961.0	815.5

Our preliminary best runs for Reversi are presented in Table 3.12. As the table demonstrates, our players were able to outperform the $\alpha\beta5$ and $\alpha\beta7$ players using a search depth of 4.

3.4 Discussion

We presented a GP approach for discovering effective strategies for playing the game of lose checkers, also showing preliminary promising results for two additional board games. Guided by the gene-centered view of evolution, which describes evolution as a process in which segments of self-replicating units of information compete for dominance in their genetic environment, we introduced several new ideas and adaptations of existing ideas for augmenting the GP approach. Having evolved successful players, we established that tree-based GP is applicable to board-state evaluation in lose checkers—a full, nontrivial board game.

A number of possible avenues of further research may be considered:

1. Applying the GP approach to other computationally interesting checkers variants, as preliminarily shown for 10x10 checkers. This has the advantage that our system needs to be changed only minimally and most effort can be invested in applying new methods to improve the evolutionary algorithm's performance.

2. Applying the GP approach to other board games, as we preliminarily did with Reversi. This work is, as far as we know, the first time tree-based GP was used to evolve players for a full board game, and one of the very few attempts at applying the evolutionary approach to board games in general. Our attempt demonstrates that a game like lose checkers can be tackled even with little domain knowledge and expertise.

3. GP can be applied to guiding search itself in games and puzzles, deciding which nodes to develop first (we'll see this in Chapter 6).

4. GP can also be applied to more-complicated games that are not full-knowledge, or contain a stochastic element, as we shall see further on in this book.

5. Investigate other fields where search is used in conjunction with heuristic functions, such as planning in AI (we'll encounter this line of research in Part IV).

As long as the strategies for solving the problem can be defined and the quality of solvers can be evaluated in reasonable time, there is an opening for using GP to evolve a strong problem-solving program. All that is required is that solvers be evaluated in such a way that those solvers that are closer to doing the job right get higher fitness, and that the search space defined by the GP setup be such that good solvers tend to be clumped together (i.e., be similar to each other), or that the same code segments tend to appear in many good solvers, so as to allow the gradual change and improvement that is a hallmark of the evolutionary process.

Chapter 4

Chess: Endgames

The game of chess has always been viewed as an intellectual game par excellence, “a touchstone of the intellect,” according to Goethe. The game’s complexity stems from two main sources. First, the size of the search space: after the opening phase, each player has to select the next move from approximately 25 possible moves on average. Since a single game typically consists of a few dozen moves, the search space is enormous. A second source of complexity stems from the amount of information contained in a single board. Since each player starts with 16 pieces of 6 different types, and as the board comprises 64 squares, evaluating a single board (a “position”) entails elaborate computation, even without looking ahead.

Computer programs capable of playing the game of chess have been designed for more than 50 years, starting with the first working program that was reported in 1958 (Bernstein and de V. Roberts [20]). According to Russell and Norvig [151], from 1965 to 1994 there was an almost linear increase in the strength of computer chess programs—as measured in their performance in human-rated tournaments. This increase culminated in the defeat in 1997 of Gary Kasparov—the former World Chess Champion—by IBM’s special-purpose chess engine, Deep Blue (DeCoste [49]; Deep Blue [50]).

Deep Blue, and its offspring Deeper Blue, relied mainly on brute-force methods to gain an advantage over the opponent, by traversing as deeply as possible the game tree (Campbell et al. [28]). Although these programs achieved amazing performance levels, Chomsky [38] criticized this aspect of game-playing research as being “about as interesting as the fact that a bulldozer can lift more than some weight lifter.”

The number of feasible games possible (i.e., the size of the game tree), given a board configuration, is astronomical, even if one limits oneself to endgames. While endgames typically contain but a few pieces, the problem of evaluation is still hard, as the pieces are usually free to move all over the board, resulting in complex game trees—both deep and with high branching factors. Thus, we cannot rely on brute-force methods alone. We need to develop better ways to approximate the outcome of games with “smart” evaluation functions. The automated learning of evaluation functions is a promising research area if we are to produce stronger artificial players (Kendall and Whitwell [101]).

We shall apply GP to evolving board-evaluation functions (Hauptman and Sipper [79]; Sipper et al. [169]).

4.1 Previous Work

Ferrer and Martin [63] had a computer play the ancient Egyptian board game of Senet, by evolving board-evaluation functions using tournament-style fitness evaluation. Gross et al. [72] introduced a system that integrated GP and Evolution Strategies to learn to play chess. This system did not learn from scratch, but instead a “scaffolding” algorithm that could perform the task already was improved by means of evolutionary techniques.

Kendall and Whitwell [101] used evolutionary algorithms to tune evaluation-function parameters. The resulting individuals were successfully matched against commercial chess programs, but only when the lookahead for the commercial program was strictly limited.

In the work of Fogel et al. [66], a genetic algorithm was employed to improve the tuning of parameters that governed a set of features regarding board evaluation. Evaluation functions were structured as a linear combination of: 1) the sum of the material values attributed to each player; 2) values derived from tables indicating the worth of having certain pieces at certain values—“positional value tables”; and 3) three neural networks: one for each player’s front two rows, and one for the central 16 squares. Games were played using a search of depth 4–6 ply, with alpha-beta pruning (Knuth and Moore [103]). The best evolved neural network achieved an above-Master level of performance, estimated at 2437.

These works used simple board-evaluation functions as the building blocks for the evolutionary algorithm. For example, some typical functions used by Gross et al. [72] were: material values for the different pieces, penalty for bishops in initial positions, bonus for pawns in the center of the chessboard, penalty for doubled pawns and for backward pawns, castling bonus if this move was taken and penalty if it was not, and rook bonus for an open line or on the same line of a passed pawn. Kendall and Whitwell [101] used fewer board-evaluation functions, focusing on the weights of the remaining pieces.

More recently, David-Tabibi et al. [47] demonstrated how genetic algorithms could be used to reverse engineer an evaluation function’s parameters. Using an appropriate expert (or mentor) they evolved a program that was on a par with top tournament-playing chess programs. The performance gain was achieved by evolving a program that mimicked the behavior of a superior expert.

4.2 Evolving Chess Endgame Strategies

We evolved endgame strategies using Koza-style GP (Koza [108, 109]). Each individual—a LISP-like tree expression—represented a strategy, the purpose of which was to evaluate a given board configuration and generate a real-valued score. We used simple Boolean functions (AND, OR, NOT) and conditional statements, with terminals used to analyze certain features of the game position. We included a large number of terminals, varying from simple ones (such as the number of moves for the player's king), to more complex features (for example, the number of pieces attacking a given piece). A full description of functions and terminals used is given in Section 4.2.3.

In order to better control the structure of our programs we used Strongly Typed GP (Chapter 2), also used in the checkers system.

4.2.1 Board evaluation

Our purpose was to create a program that analyzes single nodes thoroughly, in a way reminiscent of human thinking, and therefore we did not perform deep lookahead.

We evolved individuals represented as LISP programs. Each such program received a chess endgame position as input, and, according to its sensors (terminals) and functions, returned an evaluation of the board, in the form of a real value.

Our chess endgame players consisted of an evolved LISP expression, coupled to a program that generated all possible (legal) next-moves and fed them to the expression. The next-move with the highest score was selected (ties were broken stochastically). The player also identified when the game was over (either by a draw or a win).

4.2.2 Tree topology

Our programs played chess endgames consisting of kings, queens, and rooks. Each game started from a different (random) legal position, in which no piece was attacked, e.g., two kings, two rooks, and two queens in a KQRKQR endgame. Although at first each program was evolved to play a different type of endgame (KRKR, KRRKRR, KQKQ, KQRKQR, etc.), which implies using different game strategies, the same set of terminals and functions was used for all types. Moreover, this set was also used for our more complex runs, in which GP chess players were evolved to play several types of endgames. Our ultimate aim was the evolution of general-purpose strategies.

As most chess players would agree, playing a winning position (e.g., with material advantage) is very different than playing a losing position, or an even one. For this reason we defined each individual to include three trees: an advantage tree, an even tree, and a disadvantage tree. These trees were used according to the current status of the board. The disadvantage tree was smaller, since achieving a stalemate and avoiding exchanges requires less complicated reasoning. Most terminals and functions were used for all trees.

4.2.3 Tree nodes

While evaluating a position, an expert chess player considers various aspects of the board. Some are simple, while others require a deep understanding of the game. Chase and Simon [35] found that experts recalled meaningful chess formations better than novices. This led them to hypothesize that chess skill depends on a large knowledge base, indexed through thousands of familiar chess patterns.

We assumed that complex aspects of the game board are comprised of simpler units, which require less game knowledge, and are to be combined in some way. Our chess programs used terminals, which represented those relatively simple aspects, and functions, which incorporated no game knowledge, but supplied methods of combining those aspects. As we used Strongly Typed GP, all functions and terminals were assigned one or more of two data types: *Float* and *Boolean*. We also included a third data type, named *Query*, which could be used as either Float or Boolean.

The function set used included the If function, and simple Boolean functions. Although our trees returned a real number we omitted arithmetic functions, for several reasons. First, a large part of contemporary research in the field of machine learning and game theory (in particular for perfect-information games) revolves around inducing logic rules for learning games (for example, see Bain [12]; Bonanno [21]; Fürnkranz [69]). Second, according to the expert players we consulted, while evaluating positions involves considering various aspects of the board, some more important than others, performing logic operations on these aspects seems natural, while performing mathematical operations does not. Third, we observed that numeric functions sometimes returned extremely large values, which interfered with subtle calculations. Therefore the scheme we used was a (carefully ordered) series of Boolean queries, each returning a fixed value (either an `ERC` or a numeric terminal, see below). Table 4.1 lists the complete set of functions.

Table 4.1 Function set of GP individual. B: Boolean, F: Float.

Node name	Type	Return value
If3(B_1,F_1,F_2)	F	If B_1 is non-zero, return F_1, else return F_2
Or2(B_1,B_2)	B	Return 1 if at least one of B_1, B_2 is non-zero, 0 otherwise
Or3(B_1,B_2,B_3)	B	Return 1 if at least one of B_1, B_2, B_3 is non-zero, 0 otherwise
And2(B_1,B_2)	B	Return 1 only if B_1 and B_2 are non-zero, 0 otherwise
And3(B_1,B_2,B_3)	B	Return 1 only if B_1, B_2, and B_3 are non-zero, 0 otherwise
Not(B_1)	B	Return 0 if B_1 is non-zero, 1 otherwise
Smaller(F_1,F_2)	B	Return 1 if F_1 is smaller than F_2, 0 otherwise
Equal(F_1,F_2)	B	Return 1 if F_1 is equal to F_2, 0 otherwise

4.2.4 Terminal set

We developed most of our terminals by consulting several high-ranking chess players.[1] The terminal set examined various aspects of the chessboard, and can be divided into 3 groups:

Float values were created using the ERC (Ephemeral Random Constant) mechanism (discussed in the previous chapter). An ERC was chosen at random to be one of the following six values: $\pm 1 \cdot \{\frac{1}{2}, \frac{1}{3}, \frac{1}{4}\} \cdot MAX$ (MAX was empirically set to 1000), and the inverses of these numbers. This guarantees that when a value is returned after some group of features has been identified, it will be distinct enough to engender the outcome.

Simple terminals analyzed relatively simple aspects of the board, such as the number of possible moves for each king, and the number of attacked pieces for each player. These terminals were derived by breaking relatively complex aspects of the board into simpler notions. More complex terminals belonged to the next group (see below). For example, a player should capture its opponent's piece if it is not sufficiently protected, meaning that the number of attacking pieces the player controls is greater than the number of pieces protecting the opponent's piece, and the material value of the defending pieces is equal to or greater than the player's. Adjudicating these considerations is not simple, and therefore a terminal that performs this entire computational feat by itself belongs to the next group of complex terminals.

The simple terminals comprising this second group were derived by refining the logical resolution of the previous paragraphs' reasoning: Is an opponent's piece attacked? How many of the player's pieces are attacking that piece? How many pieces

[1]The highest-ranking player we consulted was Boris Gutkin, ELO 2410, International Master, and fully qualified chess teacher.

are protecting a given opponent's piece? What is the material value of pieces attacking and defending a given opponent's piece? All these questions were embodied as terminals within the second group. The ability to easily embody such reasoning within the GP setup, as functions and terminals, is a major asset of GP, an issue we shall elaborate upon in Chapter 12.

Other terminals were also derived in a similar manner. Table 4.2 provides a complete list of simple terminals. Note that some of the terminals are inverted—we would like terminals to always return positive (or *true*) values, since these values represent a favorable position. This is why, for example, we used a terminal evaluating the player's king's *distance* from the edges of the board (generally a favorable feature for endgames), while using a terminal evaluating the *proximity* of the opponent's king to the edges (again, a positive feature).

Complex terminals check the same aspects of the board a human player would. Examples include: considering the capture of a piece; checking if the current position is a draw, a mate, or a stalemate (especially important for non-even boards); checking if there is a mate in one or two moves (this is the most complex terminal); the material value of the position; and, comparing the material value of the position to the original board—this is important since it is easier to consider change than to evaluate the board in an absolute manner. Table 4.3 provides a full list of complex terminals.

Since some of these terminals are hard to compute, and most appear more than once in the individual's trees, we used a memoization scheme to save time (Abelson et al. [1]): After the first calculation of each terminal the result is stored so that further calls to the same terminal (on the same board) do not repeat the calculation. Memoization greatly reduced the evolutionary runtime.

4.2.5 Fitness evaluation

We used a competitive evaluation scheme, with the fitness of an individual being determined by its success against its peers. We applied the random-2-ways method (Angeline and Pollack [7]; Panait and Luke [135]), in which each individual plays against a fixed number of randomly selected peers. Each of these encounters entailed a fixed number of games, each starting from a randomly generated position in which no piece was attacked.

The score for each game was derived from the outcome of the game. Players that managed to mate their opponents received more points than those that achieved only a material advantage. Draws were rewarded by a score of low value and losses entailed no points at all.

The scoring method was based on the one used in chess tournaments: victory—1 point, draw—$\frac{1}{2}$ point, loss—0 points. In order to better differentiate our players, we

Table 4.2 Simple terminals. Opp: opponent, My: player.

Node name	Type	Return value
`NotMyKingInCheck`	B	Is the player's king not being checked?
`IsOppKingInCheck`	B	Is the opponent's king being checked?
`MyKingDistEdges`	F	The player's king's distance from the edges of the board
`OppKingProximityToEdges`	F	The player's king's proximity to the edges of the board
`NumMyPiecesNotAttacked`	F	The number of the player's pieces that are not attacked
`NumOppPiecesAttacked`	F	The number of the opponent's attacked pieces
`ValueMyPiecesAttacking`	F	The material value of the player's pieces which are attacking
`ValueOppPiecesAttacking`	F	The material value of the opponent's pieces which are attacking
`IsMyQueenNotAttacked`	B	Is the player's queen not attacked?
`AreMyPiecesUnattacked`	B	Are all of the player's pieces not attacked?
`IsOppQueenAttacked`	B	Is the opponent's queen attacked?
`IsMyFork`	B	Is the player creating a fork?
`IsOppNotFork`	B	Is the opponent not creating a fork?
`NumMovesMyKing`	F	The number of legal moves for the player's king
`NumNotMovesOppKing`	F	The number of illegal moves for the opponent's king
`MyKingProxRook`	F	Proximity of player's king and rook(s)
`OppKingDistRook`	F	Distance between opponent's king and rook(s)
`MyPiecesSameLine`	B	Are two or more of the player's pieces protecting each other?
`OppPiecesNotSameLine`	B	Are two or more of the opponent's pieces protecting each other?
`IsOppKingProtectingPiece`	B	Is the opponent's king protecting one of its pieces?
`IsMyKingProtectingPiece`	B	Is the player's king protecting one of its pieces?

Table 4.3 Complex terminals. Opp: opponent, My: player. Some of these terminals perform lookahead, while others compare with the original board.

Node name	Type	Return value
`EvaluateMaterial`	F	The material value of the board
`IsMaterialIncrease`	B	Did the player capture a piece?
`IsMate`	B	Is this a mate position?
`IsMateInOne`	B	Can the opponent mate the player after this move?
`OppPieceCanBeCaptured`	B	Is it possible to capture one of the opponent's pieces without retaliation?
`MyPieceCannotBeCaptured`	B	Is it not possible to capture one of the player's pieces without retaliation?
`IsOppKingStuck`	B	Do all legal moves for the opponent's king advance it closer to the edges?
`IsMyKingNotStuck`	B	Is there a legal move for the player's king that advances it away from the edges?
`IsOppKingBehindPiece`	B	Is the opponent's king two or more squares behind one of its pieces?
`IsMyKingNotBehindPiece`	B	Is the player's king not two or more squares behind one of its pieces?
`IsOppPiecePinned`	B	Is one or more of the opponent's pieces pinned?
`IsMyPieceNotPinned`	B	Are all the player's pieces not pinned?

rewarded $\frac{3}{4}$ points for a material advantage (without mating the opponent).

The final score was the sum of all scores a player received, divided by the number of games. This way, a player that always mated its opponent received a perfect score of 1. The score for a player that played against an opponent of comparable strength (where most games end in a draw) was 0.5 on average.

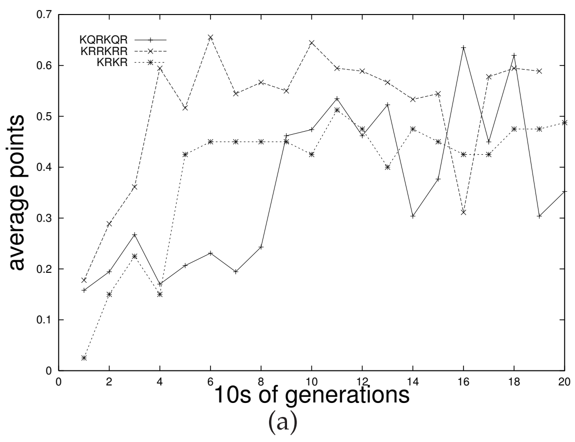
The final fitness for each player was the sum of all points earned in the entire tournament for that generation.

4.2.6 Run parameters

We used the standard reproduction, crossover, and mutation operators. The major parameters were: population size—80, generation count—between 150 and 250, reproduction probability—0.35, crossover probability—0.5, and mutation probability—0.15 (including ERC mutation).

4.3 Results

We conducted three sets of experiments to test our evolving chess players:



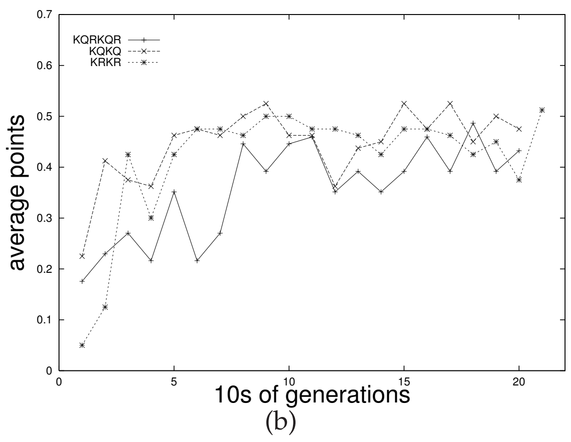


Figure 4.1 (a) Results against a strategy defined by a chess Master. The three graphs show the average score over time of the best run of 50 runs carried out, for three types of endgames: KRKR, KRRKRR, KRQKRQ. A point represents the score of the best individual at that time, pitted in a 150-game tournament against the human-defined strategy. (b) Results against Crafty. The three graphs show the average score over time of the best run of 15 runs carried out, for three types of endgames: KRKR, KQKQ, KQRKQR. A point represents the score of the best individual at that time, pitted in a 50-game tournament against Crafty.

1. Competing against a human-defined strategy.
2. Competing against a world-class chess engine.
3. Multiple-endgame runs.

4.3.1 Competing against a human-defined strategy

As noted above, we developed most of our terminals by consulting several high-ranking (human) chess players. In order to evaluate our system we wished to test our evolved strategies against some of these players. Because we needed to play thousands of games in every run, these could not be conducted manually, but instead we programmed a good strategy, based on the guidance from the players we consulted. We wrote this evaluation program using the functions and terminals of our GP system.

During evolution our chess programs competed against each other. However, every 10 generations the best individual was extracted and pitted in a 150-game tournament against the human-defined strategy. The results are depicted in Figure 4.1(a), showing runs for KRKR-, KRRKRR-, and KQRKQR-type endgames.

We observe that starting from a low level of performance, chess players evolved to play as good as high-ranking humans for all groups of endgames, in one case even going beyond a draw to win (KQRKQR endgame, where a high score of 0.63 was attained). Improvement was rapid, typically requiring only a few dozen generations.

4.3.2 Competing against a world-class chess engine

Having attained good results against a human-defined strategy based on expert chess players, we went one step further and competed against a highly powerful chess engine. For this task, we used the Crafty engine (version 19.01) by Hyatt.[2] Crafty is a state-of-the-art chess engine, using a typical brute-force approach, with a fast evaluation function, NegaScout search, and all the standard enhancements (Jiang and Buro [94]). Crafty finished second at the 12th World Computer Speed Chess Championship, held at Bar-Ilan University in 2004. According to the Computer Chess Rating Lists website,[3] Crafty has a rating of 2846 points, which places it at the human Grandmaster level. Crafty is thus, undoubtedly, a worthy opponent.

As expected, Crafty proved to be a formidable opponent, constantly mating the GP opponent at early generations. However, during the process of evolution, substantial improvement was observed to occur. As shown in Figure 4.1(b), our program managed to achieve near-draw scores, even for the complex KQRKQR endgame. Considering that our evolved 2-lookahead (counting terminals) programs competed against a world-class chess player, our method seems quite viable and promising.

4.3.3 Multiple-endgame runs

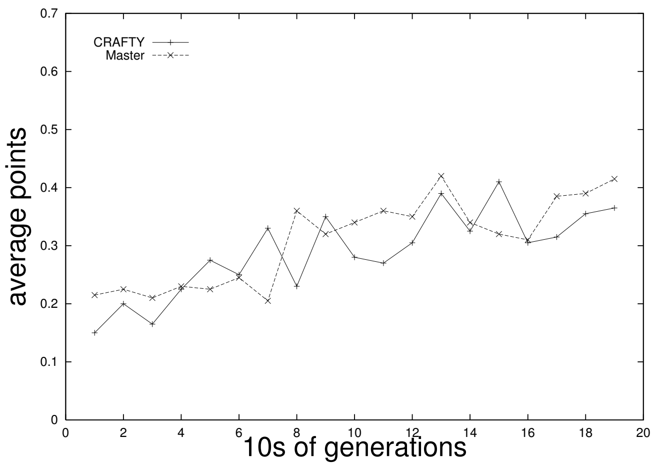
Aiming for general-purpose strategies, this third experimental suite involved the playing of one game *of each type* (rather than a single type)—both during evolution and in the test tournaments. Evolved players were pitted against the Master-defined strategy and Crafty. As can be seen in Figure 4.2, near-draw scores were achieved under these conditions as well. We observed that performance kept improving and are confident that it would continue doing so with added computational resources.

4.4 Discussion

We presented a method by which chess endgame players may be evolved to successfully hold their own against excellent opponents. One of the major prima facie problems with our scheme is its complexity, as evidenced by the terminal and function tables in Section 4.2. In the time-honored tradition of computer science, we argue that this is not a bug but rather a feature—to be more precise, a somewhat overlooked feature of genetic programming. We believe that GP represents a viable means to automatic programming, and perhaps more generally to machine intelligence, in no small part due to its being *cooperative with humans*.

[2]Crafty's source code is available at `ftp://ftp.cis.uab.edu/pub/hyatt`.

[3]`http://www.computerchess.org.uk/ccrl/`



	%Wins	%Advs	%Draws
Master	6	4	80
Crafty	2	2	77

Figure 4.2 Left: Results for multiple-endgame runs—wherein all endgames were used during evolution—against both Crafty and the Master-defined strategy. Each graph shows the average score over time of the best run of 20 runs carried out. A point represents the score of the best individual at that time, pitted in a 50-game tournament against Crafty or a 150-game tournament against Master. Right: Percent of wins, material advantage without mating (having a higher point count than the opponent), and draws for best tournament of run (i.e., fitness peak of graph).

We did not design our genome (Tables 4.1, 4.2, and 4.3) in one fell swoop, but rather through an incremental, interactive process, whereby man (represented by the humble authors of this work) and machine (represented by man's university's computers) worked hand-in-keyboard. To wit, we began our experimentation with small sets of functions and terminals, which were revised and added upon through our examination of evolved players and their performance, and through consultation with high-ranking chess players. GP's design includes cooperativeness, often overlooked, which is perhaps one of its major boons. We shall have much more to say about this in Chapter 12.

Note that the number of terminals we used is small, compared to the number of patterns used by chess experts when evaluating a position, which, according to Simon and Gilmartin [163], is close to 100,000. Since most pattern-based programs nowadays are considered to be far from competitive (see Fürnkranz [69]), the results we obtained may imply that we have made a step towards developing a program that has more in common with the way humans play chess.

In the next chapter we turn to analyzing the performance of our best evolved player.

Chapter 5

Chess: Emergence and Complexity

Genetic programming (GP) has been shown to successfully produce solutions to hard problems from numerous domains, and yet an understanding of the evolved "spaghetti code" is usually lacking. Indeed, it seems a GPer must wear two hats: that of an evolutionary designer, and that of a "molecular biologist" (Sipper [165]). The latter hat allows one to gain more *qualitative* information regarding the evolved individuals' capabilities, going beyond the rather laconic *quantitative* performance scores.

We wore the first hat in Chapter 4, presenting successful chess endgame players evolved via GP. In this chapter we wish to wear the second hat—that of the "molecular biologist"—in an attempt to understand the resultant intelligence, hidden within the innards of our evolved programs (Hauptman and Sipper [78, 80]) (we shall wear this hat again in Chapters 7 and 9, and we also wore it in Wolfson et al. [185], albeit in a domain unrelated to games). A key observation is that large GP individuals (i.e., typically containing between tens to thousands of functions and terminals) fall within the framework of *Complex Systems*, "being within the mesoscopic domain—containing more than a few, and less than too many [interacting] parts." (Bar-Yam [13])

The purpose of the experiments described in this chapter is to attempt to answer several questions regarding the qualities of our strongest evolved players. Do they display *emergent* playing capabilities, surpassing the game knowledge imbued into them? If so, should these capabilities be attributed to specific terminals in the GP trees of the strongest players, to simple combinations of terminals, or to the entire individual? What strategies were developed exactly? And, how do they compensate for the lack of deep search?

In an attempt to answer these questions we perform two suites of experiments. First, we examine a short endgame played by one of the strongest individuals evolved in Chapter 4, and draw some conclusions regarding its playing capabilities. Second, we propose novel measures for determining the effectiveness of GP building blocks (both single terminals and groups of terminals), and use them to analyze their impact

```
(If3 (Or2 (Not (Not IsOppKingInCheck)) (And2 (Or3 (Or2 IsOppKingInCheck
IsOppKingInCheck) (Or3 IsMyKingNotStuck IsMyFork OppPieceCanBeCaptured) (Not
IsMyFork)) (Not NotMyKingInCheck))) (If3 (Or3 (Not (And3 OppPieceCanBeCaptured
MyPieceCannotBeCaptured IsOppKingBehindPiece)) (Or3 (And3 NotMyKingInCheck
IsOppKingBehindPiece IsMyFork) (Or2 -1000*IsMateInOne NotMyKingInCheck)
(And2 IsOppKingInCheck NotMyKingInCheck)) (Or3 (Or3 (And3 NotMyKingInCheck
IsOppKingBehindPiece IsMyFork) (Or2 -1000*IsMateInOne NotMyKingInCheck)
(And2 IsOppKingInCheck NotMyKingInCheck)) (Or3 IsMyFork IsMyKingNotStuck
IsMyKingNotStuck) (And2 IsOppKingStuck OppPieceCanBeCaptured))) (If3 (Not (And2
IsMyFork 1000*IsMate)) (If3 (And2 100*Increase IsOppKingStuck) (If3 1000*IsMate
NumMyPiecesNotAttacked NumNotMovesOppKing) (If3 MyPieceCannotBeCaptured
NumNotMovesOppKing -1000*IsMateInOne)) (If3 IsOppKingBehindPiece
-1000*IsMateInOne -1000*IsMateInOne)) (If3 IsMyKingNotStuck IsOppKingBehindPiece
IsMyFork)) (If3 (Or2 (Or3 (Or2 IsOppKingInCheck IsOppKingInCheck)
(Or3 IsMyKingNotStuck IsMyFork OppPieceCanBeCaptured) (Not IsMyFork))
(And2 IsOppKingInCheck NotMyKingInCheck)) (If3 (And2 IsOppKingInCheck
NotMyKingInCheck) (If3 1000*IsMate NumNotMovesOppKing IsMyFork) (If3
-1000*IsMateInOne IsMyKingNotBehindPiece IsOppKingBehindPiece)) (If3 (Or3 (And2
IsMyKingNotStuck IsMyKingNotStuck) (Or3 IsOppKingInCheck OppPieceCanBeCaptured
IsMyFork) (And3 IsOppKingInCheck OppPieceCanBeCaptured IsOppKingStuck))
(If3 -1000*IsMateInOne IsMyKingNotBehindPiece IsOppKingBehindPiece)
(If3 (< MyKingDistEdges NumMyPiecesNotAttacked) (If3 1000*IsMate
NumNotMovesOppKing IsMyFork) (If3 -1000*IsMateInOne IsMyKingNotBehindPiece
IsOppKingBehindPiece)))))
```

Figure 5.1 Advantage tree of Gpec190.

on the evolved player's performance.

5.1 Analysis of Moves

We describe a sample game played by a strong GP-Endchess individual, obtained at generation 190, dubbed Gpec190, or Gpec for short. Gpec scored 0.485 points against Crafty and Master on average (0.5 being a draw) in multiple-endgame runs. Figure 5.1 presents Gpec's advantage tree.

In the games below Gpec plays against itself, with Crafty being used to analyze every move (i.e., Crafty also plays each board position of the game). Although Gpec played both Black and White we focus only on White's moves, since in the following examples the number of possible moves for White is on average above 30, while for Black—only 3 or 4.

Crafty here serves as a superb yardstick, allowing us to compare the scores assigned by Gpec to "real" scores (Crafty's scores were obtained by deep analysis of each move, typically lasting 25–30 seconds, at an average speed above 1500 KNodes/sec, so are therefore highly reliable). Since exact scores assigned by different chess engines to moves vary widely, calculating a correlation factor between Crafty's

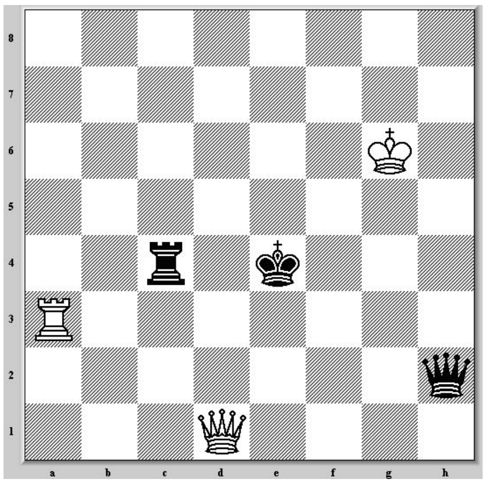


Figure 5.2 Opening position of a sample game.

scores and Gpec's scores would be futile. However, if a win (mate-in-n, or mate in n moves) exists, or there is a smaller advantage to one of the sides, (near) optimal moves are more easily identifiable (as maintaining the advantage), and distinguished from "bad" ones (losing the advantage).

We consider a KQRKQR opening position. As in this type of position the player to move first (White in our experiments) has the strategic advantage, in order to verify correct play we should check that the player maintains the advantage, especially if a mate-in-n as identified by Crafty exists, making the optimal line of play well defined. The player should assign higher scores to moves assigned higher scores by Crafty, and lower scores to moves forfeiting the advantage.

Instead of going into the details of each move, we display scoring tables for the moves considered, and their assigned scores (both by Crafty and Gpec), and only discuss some of the moves.

We discuss a sample game, for which the moves appear in Table 5.1. The starting position is given in Figure 5.2.

As can be seen, the best moves according to Crafty were always included in Gpec's highest-rated moves (top scores). However, Gpec's play was not always optimal (for example, see second move) since other, sub-optimal, good moves also received high scores. Since there were always 36 or more possible moves, it is highly unlikely that such a result would stem from mere chance.

We hereby discuss some of Gpec's tactics, and relevant terminals that may affect them:

- Capturing pieces (also known as *material*) is an integral part of any chess-playing program's strategy. Indeed, one might even construct a strong chess program based solely on material considerations and deep lookahead. However, since blindly capturing pieces is far from being a perfect strategy, an important test to a program's playing strength is its ability to avoid captur-

Table 5.1 Top-scoring possible moves for White in the sample game studied, along with scores assigned by Crafty and Gpec. Each column represents the best options for White for the given moves (according to Gpec). All moves not appearing in the table were assigned a score of 0.0 by Gpec. Moves for Black are not included since the number of possible moves is very small. Moves played by Gpec are shown in boldface. The bottom lines show the total number of possible moves for this position, and Black's reply. This game lasted six moves.

Move 1	Crafty	Gpec
Qd3+	mate-in-9	6.0
Qf3+	mate-in-9	6.0
Qg4+	6.75	6.0
Qe1+	6.7	5.0
Qb1+	0.0	4.0
possible:	39 moves	
Black:	Ke5	

Move 2	Crafty	Gpec
Ra5+	mate-in-8	6.0
Qf5+	mate-in-8	6.0
Qe3+	mate-in-13	6.0
Qxc4	6.7	4.0
Qg3+	0.0	3.0
Possible:	37 moves	
Black:	Kd5	

Move 3	Crafty	Gpec
Rd3+	mate-in-12	6.0
Ra6+	6.05	5.0
Qb6+	6.8	4.0
Qd3+	0.0	2.0
Qg3+	0.0	2.0
Possible:	39 moves	
Black:	Kc6	

Move 4	Crafty	Gpec
Qe6+	mate-in-11	4.0
Qe8+	mate-in-11	4.0
Qf3+	mate-in-15	0.0
Possible:	36 moves	
Black:	Kc5	

Move 5	Crafty	Gpec
Qd5+	mate-in-7	5.0
Qf5+	6.5	5.0
Qe3+	mate-in-8	5.0
Qc8+	mate-in-13	4.0
Possible:	40 moves	
Black:	Kb6	

Move 6	Crafty	Gpec
Rb3+	mate-in-4	6.0
Qxc4	6.05	5.0
Qd6+	6.8	4.0
Qe6+	0.0	2.0
Qd8+	0.0	2.0
Possible:	39 moves	
Black:	Ka6	

ing "poisoned" pieces (eventually leading to losing the game). Knowing that capturing a piece is wrong typically requires tapping more elaborate knowledge, or looking ahead deeper into the game tree. On the second move Gpec can capture the opponent's rook by Qxc4. This is a good move (scored 6.7 by Crafty), not a blunder, but still sub-optimal since there exist other moves leading to mate-in-8 and mate-in-13. Gpec awarded 6.0 points to the two optimal moves (leading to mate-in-8) and to Qe3+, which is slightly sub-optimal (mate-in-13) and stochastically selected Qe3+. The capture move received only 4.0 points. Preferring a move leading to mate-in-13 over a strong capturing move is by no means a trivial achievement for a program with a lookahead of 1! To sum up, Gpec has learned through emergent evolution the value of material (manifested in not losing pieces), but knows when it is less important. Relevant terminals are: `EvaluateMaterial`, `IsMaterialIncrease`, `IsMyQueenNotAttacked`, `IsMyFork`, `OppPieceAttackedUnprot`.

- Gpec has learned to identify the need to repeatedly check the opponent (most moves in the table are checking moves), which is usually crucial to maintaining the advantage. Still, there is a significant difference between various checking moves (noted both by Gpec's and Crafty's varying scores to these moves). Relevant terminals: `IsOppKingInCheck`, `IsOppKingBehindPiece`, `IsMate`.

- Cornering the opponent's king was an important factor in Gpec's decision. As can be seen in the table, moves leaving fewer free squares to the opponent's king received higher scores (this is reflected in the `NumNotMovesOpponentKing` terminal's output). While this tactic is important when trying to deliver a mate, Gpec correctly avoids such moves that jeopardize the attacking piece. Moreover, Gpec still differentiated cornering moves, assigning higher scores to moves leading to relatively closer mates (see moves 4, 5, 6). Relevant terminals: `IsOppKingStuck`, `OppKingProximityToEdges`, `IsOppKingBehindPiece`.

- Gpec preferred to attack with the queen instead of the rook (though sometimes the rook is selected to attack). This was correct in various positions (as can be seen in Crafty's scores). The queen is also more capable of delivering forks. Relevant terminals: `ValueMyPiecesAttacking`, `IsMyFork`.

The above list is only partial, but aids in grasping some of the complicated considerations involved in our evolved player's decisions.

It should be mentioned that Gpec played extremely well, though not perfectly. The main flaw was the lack of diversity in scoring—Gpec typically assigned a score of 0.0 to most non-winning moves. It is true that identifying the winning moves is usually sufficient to win, but when playing on the losing side it is still important to delay the loss as much as possible, as the opponent may make a mistake. As we know that Gpec can differentiate near-mates from further ones, this was not utilized while playing the losing side.

5.2 Strategic Emergence

In the previous section we witnessed a strong (though not perfect) level of play presented by the evolved player, Gpec. Now we turn to examining the emergent aspects of this skill. First, we will try to break it down and show that it is more than the sum of its components (GP-tree terminals). Then, we will gradually construct the strongest players we can, using increasingly larger groups of these "parts", and test their performance.

As several terminals were deemed relevant to our player's strategies, we first turn to examining them in isolation.

5.2.1 Single terminals

The "atoms" of our individual are single terminals. We assessed the playing strength of each terminal using a standard method and two novel measures of our own devise. We calculated scores by awarding 1 point per win and 0.5 points per draw—averaged over 500 games. Thus, an overall value of 1 signified a perfect score and 0.5 meant a play level more or less equal to the opponent's. The scores against Crafty and against Master were averaged to obtain the overall final score. The three methods of analysis were:

1. First, we examined the playing strength of individuals containing only the given terminal in their evaluation function (playing vs. Crafty and Master).[1] Each terminal was assigned a score reflecting its performance, marked S_{SINGLE}. Since 0.16 was the score for the random evaluation function in our experiments, terminals that scored 0.16 (or less) presented zero (or even negative) playing strength.

2. To compute the second measure, marked S_{DIS}, we "handicapped" several strong endgame players we developed (including Gpec) by disabling the given terminal (altering its function to return either a random number or zero whenever it was called, instead of the regular output). The scores reflect the average *decrease* in performance when the given terminals were thus disabled.

[1]It was possible to test the terminals this way (the entire evaluation function being the terminal) since they were all implemented such that they returned larger values for better board configurations for the player. For example, compare `NotMyKingInCheck` (returning 1 when the player's king is *not* attacked, and 0 otherwise) to `IsOppKingInCheck` (returning 1 when the opponent's king *is* attacked).

3. The third measure was the result of sets of experiments (containing 10 runs for each terminal), in which we evolved individuals containing all terminals *except for* the given terminal. Under this condition the strongest individuals evolved were recorded. We averaged their performance and subtracted it from 0.485, which is Gpec's score (to reflect the fact that if stronger individuals were evolved without the terminal, it is probably less influential). This score is presented as the last measure,[2] marked S_{NO}.

The *Strength* of each terminal was computed as the arithmetic average of all three measures: $\mathit{Strength} = (S_{SINGLE} + S_{DIS} + S_{NO})/3$.

Results are summarized in Table 5.2. As can be seen, the contribution of each "atom" to Gpec's overall success—even when measured in multiple ways—is relatively small. As noted above, S_{SINGLE} scores below 0.16 mean that the terminal is, in and of itself, worse than a random function (although a random function would score 0 on the second and third measures). As some terminals used by Gpec (for example, `IsOppKingStuck`) received zero or *negative* S_{SINGLE} scores (i.e., less than the random score of 0.16), it is highly likely that using these terminals is non-trivial for evolution with the full terminal set.

Another interesting point to note is that the terminal `NumNotMovesOppKing`, which is clearly an integral part of Gpec's strategy (due to the apparent closeness of Gpec's evaluation scores and those of this terminal), did not even make it to the top 12 (it is ranked only 15th). Also, `IsMateInOne` (ranked 2nd) is not used by Gpec and several other strong players.

We conclude that single terminals are weak and insufficient to explain the overall playing strength of a full-blown evolved strategy, even if some diversity can be seen in their *Strength* measures.

5.2.2 Terminal pairs

Next we turn to examining small "molecules" containing 2 atoms each: we selected pairs of strong terminals—the top-ranking ones from Table 5.2 (except that we avoided pairing similar terminals, such as `MyPieceCannotBeCaptured` and `AreMyPiecesUnattacked`)—and attempted to reach the maximal level of play attainable with these pairs. This was accomplished by evolving GP individuals using only one pair of terminals (per experiment), and all functions from the function set (see Tables 4.1, 4.2, and 4.3). The depth of the GP-trees was bound by 4.

Results appear in Table 5.3, reflecting the best evolved individual's playing strength against Crafty and Master for each pair. As can be seen, combining two

[2]For some terminals, this score is less significant since there are other terminals with highly similar functionality available to evolution. For example: `EvaluateMaterial` and `IsMaterialIncrease`.

Table 5.2 The 12 most influential single terminals sorted by their overall *Strength* score (right column). This score was derived from three measures: S_{SINGLE}, the average score against Crafty and Master when using only the given terminal in the evaluation function, after subtracting 0.16 (the score of the random evaluation function); S_{DIS}, the average decrease in performance of several strong players when the given terminal is still present—but disabled (returns a random value); and S_{NO}, the average score for the best evolved individuals when evolving with all terminals *except* the given one, subtracted from 0.485, which is Gpec's score. The overall *Strength* of the terminal is the average of these three measures.

Terminal	S_{SINGLE}	S_{DIS}	S_{NO}	*Strength*
`MyPieceCannotBeCaptured`	0.26	0.131	0.10	0.164
`IsMateInOne`	0.16	0.105	0.15	0.138
`AreMyPiecesUnattacked`	0.32	0.010	0.04	0.123
`NumMyPiecesNotAttacked`	0.30	0.008	0.06	0.123
`EvaluateMaterial`	0.18	0.056	0.11	0.115
`IsMate`	0.16	0.08	0.10	0.113
`IsMaterialIncrease`	0.19	0.021	0.12	0.110
`IsOppKingStuck`	0.14	0.048	0.12	0.103
`OppKingProximityToEdges`	0.16	0.106	0.02	0.096
`IsMyFork`	0.16	0.024	0.05	0.078
`IsMyKingNotStuck`	0.16	0.027	0.03	0.073
`IsOppKingInCheck`	0.04	0.027	0.10	0.057

Table 5.3 Scores for pairs of several top-ranking terminals. The top table lists the terminals used, along with their *Strength* and S_{SINGLE} scores. The bottom table shows the scores of terminal pairs. For example, the value at row 3, column 5 (0.185) is the score when the 3rd and 5th terminals together—`EvaluateMaterial` and `OppKingProximityToEdges`—comprise the evaluation function. Scores are points awarded to the strongest individuals in each run.

Index	Terminal	*Strength*	S_{SINGLE}
1	`MyPieceCannotBeCaptured`	0.164	0.26
2	`IsMateInOne`	0.138	0.16
3	`EvaluateMaterial`	0.115	0.18
4	`IsOppKingStuck`	0.103	0.14
5	`OppKingProximityToEdges`	0.096	0.16
6	`IsMyFork`	0.078	0.16
7	`IsOppKingInCheck`	0.057	0.04

Index	2	3	4	5	6	7
1	0.23	0.217	0.212	0.232	0.252	0.26
2		0.191	0.149	0.151	0.194	0.04
3			0.19	0.185	0.178	0.18
4				0.169	0.146	0.174
5					0.15	0.05
6						0.164

elements does not necessarily yield a better result. Sometimes the scores for the pair were higher than each of the individual terminals comprising it (for example, `EvaluateMaterial` and `IsOppKingStuck` received a score of 0.19, which is higher than their separate scores), but mostly this did not occur. The S_{SINGLE} score of the terminal `MyPieceCannotBeCaptured`, which was 0.26, was not improved by any pairing with another terminal (and was often even degraded).

Thus, it may be observed that emergence did not occur here—the road to improving the individual terminals' performance lies far beyond simply pairing them with other strong terminals, even when many combinations are tried by evolution.

5.2.3 Terminal groups of varying size

In light of the experiments described above we wished to better understand the connection between the number of "atoms" or "molecules" to the player's performance, and to investigate questions regarding the linearity of improvement.

And so we devised a third suite of experiments. Here the size of individuals was allowed to grow up to full-scale "poly-molecules" ("cells"), comprising all terminals. In this manner the increase in playing level (and emergent aspects) might be observed as we moved from simple units to more complex ones.

This was accomplished by increasing the size, S, of the terminal group in each

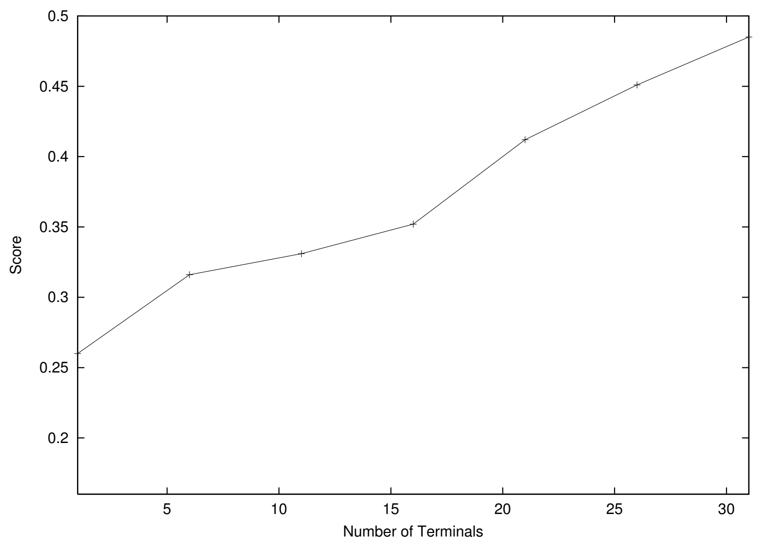


Figure 5.3 Best results obtained by evolution using an increasing number of terminals.

set of experiments. For each $S \in \{1, 6, 11, \ldots, 31\}$ (31 being the total number of different terminals), we manually selected S terminals and conducted evolutionary runs using only these terminals and with all the functions. Since weaker terminals might be randomly selected, and evolutionary runs, being partially stochastic, might sometimes fail to come up with good solutions, we repeated the process several times per S.

After each run the strongest individual's scores against both Crafty and Master were averaged for all runs (with the same S-value). Since improvement with larger groups of terminals proved more difficult, several small "cells" were constructed by hand (e.g., a few functions containing a good combination of two or more terminals were constructed), and added to the process. The results are shown in Figure 5.3.

It is important to note that although the graph seems quasi-linear, improvement is strictly non-linear, since as the level of play increases, improvement becomes more difficult. Indeed, reaching the higher scores (in the experiments with more terminals) took considerably more time and computational effort.

5.3 Discussion

We analyzed a strong evolved chess endgame player in several empirical ways. Moreover, we examined the emergent capabilities of evolved individuals, primarily in the sense that their knowledge of the game (reflected in their scores) transcended the knowledge that was infused into them.

We started by using Crafty to analyze the scores assigned by our player (Gpec) to multiple choices of moves in a short endgame, and observed that Gpec always awarded the highest score to Crafty's selected (best) move, although in a few cases other moves also received the highest score, which resulted in occasional sub-optimal play.

Next, we moved to breaking up a strategy into its component parts, and examining the parts' effect in several ways. As expected, simple units did not perform well on their own. However, their effect was more pronounced when removed (the S_{NO} measure). For example, while the `IsMateInOne` terminal scored 0.16 on its own (the same score as the random function), when it was disabled in strong players, their scores decreased on average by $S_{DIS} = 0.105$, which is a strong effect on play level (as noted above, when a player's level is high, competition becomes harsh, and every point counts).

Pairs of terminals did not prove to be much of an improvement. Although we only checked the pairs' scores (and did not conduct the more elaborate testing we did with single terminals), we were still surprised by the difficulty in joining strong terminals together correctly to use with the entire function set, even in such small groups.

As a result, considerably more computational effort was put into constructing the terminal groups of varying size. In addition, evolution was aided by several functions specifically tailored for this experiment. While this helped evolution converge faster, it may have diverted it towards local maxima. More time and effort is needed to ascertain whether evolution may find better solutions, comprised of smaller parts.

All in all, we gained insight into the emergent aspects of our evolving players' capabilities. An important conclusion is that the "leap" in performance occurs somewhere around 21 terminals, since the level of play presented by players with more terminals surpassed the capabilities of Master, which was constructed by hand, and represents our best non-evolutionary improvement.

We concluded that complex systems, presenting emergent features, may indeed be *evolved* from simple parts, given the right experimental conditions. As we are attempting to construct non-trivial heuristics (embodied as board-evaluation functions) using relatively simple parts, this is yet another proof that the method we chose is capable of producing the outcomes we seek.

Our novel methods for evaluating the so-called strength of terminals are not limited to chess, or even to games. One of the most difficult phases in constructing a GP system is that of defining the right terminals and functions, and assessing their individual "quality" may aid GPers across various domains. However, several important points must not be overlooked: First, our method can be effectively applied only after at least some strong individuals have already been constructed, so that the S_{DIS} measure may be used. Second, it may be appropriate, for some problem domains, to construct strength scores with different weights for the separate elements (S_{SINGLE}, S_{DIS}, and S_{NO}) instead of the equal weights used here. For example, if a major requirement is to evolve individuals containing as few distinct terminals as possible, S_{DIS} and S_{NO} should be assigned larger weights. Third, as the performance of the overall system is more important than that of its components, measuring the effectiveness of larger groups of terminals (rather than single ones), however more difficult, is more informative.

As stated at the beginning of this chapter, we took a less-traveled path when we decided to examine the essential features of our evolved players, beyond their numeric measures of performance. Several important insights were gained in this process, directly applicable to the next phase of our work. Perhaps the most significant insight is that more search must be incorporated into our players, to avoid observed errors, however infrequent, such as missing mates, and to scale our players to larger parts of the game. Thus, in the next chapter, we deal with the issue of adding the power of search to our evolving players.

Chapter 6

Chess: Evolving Search

Artificial intelligence for board games is widely based on developing deep, large game trees. In a two-player game, such as chess or checkers, players move in turn, each trying to win against the opponent according to specific rules. As we saw in Chapter 1, the course of the game may be modeled using a structure known as an adversarial game tree (or simply game tree), in which nodes are positions in the game and edges are moves (e.g., Rivest [146]; Stockman [172]). The complete game tree for a given game is the tree starting at the initial position (the root) and containing all possible moves (edges) from each position. Terminal nodes represent positions where the rules of the game determine whether the result is a win, a draw, or a loss.

When the game tree is too large to be generated completely, only a partial tree (called a search tree) is generated instead. This is accomplished by invoking a *search algorithm*, deciding which nodes are to be expanded at any given time, and when to terminate the search (typically at non-terminal nodes due to time constraints) (Campbell and Marsland [29]; Kaindl [100]). During the search, some nodes are evaluated by means of an *evaluation function*. This is done mostly at the leaves of the tree. Furthermore, a search can start from any position, and not just the beginning of the game.

In general, there is a tradeoff between *search* and *knowledge*, i.e., the amount of search (development of the game tree) carried out and the amount of knowledge in the leaf-node evaluator. Because deeper search yields better results but takes exponentially more time, various techniques are used to guide the search, typically pruning the game tree. While some techniques are more generic and domain independent, such as alpha-beta search (Knuth and Moore [103]; Marsland and Campbell [121]; Schaeffer and Plaat [157]) and the use of hash tables (i.e., transposition and refutation) (Beal and Smith [16]; Frey [68]; Nelson [125]; Taylor and Korf [175]), other methods rely on domain-specific knowledge. For example, quiescence search (Kaindl [99]) relies on examining capture move sequences in chess (and relevant parts of the game tree) more thoroughly than other moves. This is derived from empirical knowledge regarding the importance of capture moves. Theoretically speaking, perfect domain knowledge would render the search futile, as is the case in solved endgames in chess (Bourzutschky et al. [24]). However, constructing a full knowledge base for difficult games such as chess is still far from attainable.

While state-of-the-art chess engines integrate both search and knowledge, the scale is tipped towards generic search enhancements, rather than knowledge-based reasoning (e.g., Campbell et al. [28]; Hyatt and Newborn [93]; Newborn [126]). In this chapter we evolve a search algorithm, allowing evolution to "balance the scale" between search and knowledge. The *entire search algorithm*, based on building blocks taken from existing methods, is subject to evolution (Hauptman and Sipper [81]). Some such blocks are representative of queries performed by strong human players, allowing evolution to find ways of correctly integrating them into the search algorithm. This was previously too difficult a task to be done without optimization algorithms, as evidenced by the designers of Deep Blue (Campbell et al. [28]). Our results showed that the number of search-tree nodes required by the evolved search algorithms could be greatly reduced in many cases (Hauptman and Sipper [81]).

6.1 Previous Work

We found little work in the literature on the evolution of search algorithms. Brave [25] compared several GP methods on a planning problem involving tree search, in which a goal node was to be found in one of the leaves of a full binary tree of a given depth. While this work concluded that GP with recursive automatically defined functions (ADFs) (Koza [109]) outperformed other methods and scaled well, the problems he tackled were specifically tailored, and not real-world problems.

Hong et al. applied evolutionary algorithms to game search trees, both for single-player games (Hong et al. [91]) and two-player games (Hong et al. [92]). Each individual in the population encoded a path in the search tree, and the entire population was evolved to solve single game positions. Their results showed considerable improvement over the minimax algorithm, both in speed and accuracy, which seems promising. However, their system required that search trees have the same number of next-moves for all positions. Moreover, they did not tackle real-world games.

Gross et al. [72] evolved search for chess players using an alpha-beta algorithm as the kernel of an individual, which was enhanced by GP and Evolution Strategy modules. Thus, although the algorithmic skeleton was predetermined, the more "clever" parts of the algorithm (such as move ordering, search cut-off, and node evaluation) were evolved. Results showed a reduction in the number of nodes required by alpha-beta by an astonishing 94 percent. However, since the general framework of the algorithm was determined beforehand, the full power of evolution was not tapped. Moreover, there is no record of successfully competing against commercial programs, which are known to greatly outperform alpha-beta (with standard enhancements) on specific game-playing tasks.

We showed in Chapter 4 how to evolve chess endgame players using GP. Our evolved players successfully competed against Crafty, a world-class chess program, on various endgames. Deeper analysis of the strategies developed (Chapter 5) revealed several important shortcomings, most of which stemmed from the fact that they used deep knowledge and little search (typically, our evolved players developed only *one* level of the search tree). Simply increasing the search depth would not solve the problem, since the evolved programs examine each board very thoroughly, and scanning many boards would increase time requirements prohibitively.

And so we turn to evolution to find an optimal way to overcome this problem: How to add more search at the expense of less knowledgeable (and thus less time-consuming) node evaluators, while attaining better performance. In the experiment described in this chapter *we evolved the search algorithm itself*. While previous work on evolving search used either a scaffolding algorithm (Gross et al. [72]) or searching in toy problems (Brave [25]), we present a novel approach of *evolving the entire search algorithm*, based on building blocks taken from existing methods, integrating knowledge in the process, and applying our results to a real-world problem.

We consider all endgames, as opposed to our previous set of experiments (Chapter 4), in which we only considered a limited subset of endgames. However, an important limit has been imposed: Since efficiently searching the entire game (or endgame) tree is an extremely difficult task, we limited ourselves *for now* to searching only for game termination (or mate positions) of varying tree depths, as explained in the next section.

6.2 The Mate-in-n Problem

The mate-in-n problem in chess is defined as finding a key move such that even with the best possible counterplays, the opponent cannot avoid being mated in (or before) move n, where n counts only the player's moves and not the opponent's. This implies finding a subtree of forced moves, leading the opponent to defeat in $(2n-1)$ plies (actually, $2n$ plies, since we need an additional ply to verify a mate). Typically, for such tactical positions (where long forcing move sequences exist), chess engines search much more thoroughly, using far more resources. For example, Deep Blue searches at roughly half the usual speed in such positions (Campbell et al. [28]).

Presumably, solving the mate-in-n problem may be accomplished by performing an exhaustive search. However, because deep search is required when n is large, the number of nodes grows exponentially and a full search is next to impossible. For example, finding a mate-in-5 sequence requires searching 10 or 11 plies ahead, and more than $2 * 10^{10}$ nodes. Of course, advanced chess engines search far fewer nodes

Table 6.1 Number of nodes required to solve the mate-in-n problem by Crafty, averaged over our test examples. Depth in plies (half-moves) needed is also shown.

Mate-in	1	2	3	4	5
Depth in plies	2	4	6	8	10
Nodes examined	600	7K	50K	138K	1.6M

due to state-of-the-art search enhancements, as can be seen in Table 6.1. Still, the problem remains difficult.

A basic algorithm for solving the mate-in-n problem through exhaustive search is shown in Algorithm 6.1. First, we check if the search should terminate: successfully, if the given board is indeed a mate; in failure, if the required depth was reached and no mate was found. Then, for each of the player's moves we perform the following check: if, after making the move, *all* the opponent's moves lead (recursively) to mate-in-$(n-1)$ or better (procedure CheckOppTurn), the mating sequence was found, and we return *true*. If not, we iterate on all the player's other moves. If no move meets the condition, we return *false*.

This algorithm has much in common with several algorithms, including alpha-beta search and proof-number (pn) search (Allis et al. [5]). However, as no advanced techniques (for example, move ordering or cutoffs) are employed here, the algorithm becomes impracticable for large values of n.

In the course of our experiments we broke the algorithmic skeleton into its component building blocks, and incorporated them, along with other important elements, into the evolving GP individuals.

6.3 Evolving Mate-Solving Algorithms

Since we wish to develop intelligent—rather than exhaustive—search, our board evaluation requires special care. Human players never develop the entire tree, even when this is possible. For example, mate-in-1 problems are typically solved by only developing checking moves, and not all possible moves (since non-checking moves are necessarily non-mating moves, there is no point in looking into them). As human players only consider 2 to 3 boards per second, yet still solve deep mate-in-n problems fast (for example, Grandmasters often find winning combinations more than 10 moves ahead in mere seconds), they rely either on massive pattern recognition or on intelligent pruning, or both (Chabris and Hearst [32]).

Thus, we evolved our individuals (game search-tree algorithms) accordingly, following these guidelines:

1. Individuals only consider moves adhering to certain conditions (themselves de-

Algorithm 6.1 mate-in-n?(*board*,*depth*)

```
1:  if IsMate?(board) then
2:     return true
3:  end if
4:  if $depth = 0$ then
5:     return false
6:  end if
7:  for each $move \in$ GetNextMoves(board) do
8:     MakeMove(board,move)
9:     $result \leftarrow$ CheckOppTurn(board,depth)
10:    UndoMove(board,move)
11:    if $result$ = true then
12:       return true
13:    end if
14: end for
15: return false
Procedure CheckOppTurn(board,depth)
    // Check that all opponent's moves lead to mate-in-($n-1$)
16: for each $oppmove \in$ GetNextMoves(board) do
17:    MakeMove(board,oppmove)
18:    $result \leftarrow$ mate-in-n?(board,$depth-1$)
19:    UndoMove(board,oppmove)
20:    if $result$ = false then
21:       return false
22:    end if
23: end for
24: return true
```

veloped by evolution).

2. The amount of lookahead is left to the individual's discretion, with fitness penalties for deep lookahead (to avoid exhaustive search). Thus, we also get *evolving lookahead*.

3. Development of the game tree is asymmetrical. This helps with computation since we do not need to consider the same aspects for both players' moves.

4. Each node examined during the search is individually considered according to game knowledge, and the move sequence may be developed along a different depth.

6.3.1 Basic program architecture

Our individuals received a chessboard as input, and returned a real-valued score in the range $[-1000.0, 1000.0]$, indicating the likelihood of this board leading to a mate (higher is more likely). A representation issue was whether to evolve boards returning scores or moves (allowing a return of no move to indicate no mate has been found). An alternative approach might be evolving the individuals as move-ordering modules. However, the approach we took was more versatile and reduced the overhead of move comparison by the individual—instead of comparing moves by the GP individual, the first level of the search was done by a separate module. An evolved program thus received as input all possible board configurations reachable from the current position by making one legal move. After all options were considered by the program, the move that received the highest score was selected, and compared to the known correct solution for fitness purposes.

6.3.2 Functions and terminals

We developed most of our terminals and functions by consulting several high-ranking chess players.

Domain-specific functions These functions are listed in Table 6.2. Note that domain-specific functions typically examine if a move the player makes adheres to a given condition, which is known to lead to a mate in various positions. If so, this move is made, and evaluation continues. If not, the other child is evaluated. Also, a more generic function—`IfMyMoveExistsSuchThat`—was included to incorporate other (possibly unknown) considerations in making moves by the player. All functions undo the moves they make after evaluation of their children is completed. Since some functions are only appropriate in MAX nodes (player's turn), and others in MIN nodes (opponent's turn), some functions in the table were only used at the relevant levels. Other functions, such as `MakeBestMove`, behave differently in MAX nodes and in MIN nodes.

Sometimes functions that consider a player's move are called when it is the opponent's turn. In this case we go immediately to the *false* condition (without making a move). This solution was simpler than, for example, defining a new set of return types. Some of these functions appear as terminals also, to allow considerations to be made while it is the opponent's turn.

Generic functions These domain-independent functions are the same ones used in Table 4.1, included to allow logic and some numeric calculations.

Chess terminals A subgroup of those used in Chapter 4, shown in Table 6.3. Here, several mating aspects of the board, of varying complexity levels, are considered. From the number of possible moves for the opponent's king, through checking if the player creates a fork attacking the opponent's king, to one of the most important terminals—`IsMateInOneOrLess`. This terminal is used to allow the player to identify very close mates. Of course, repeated applications of this terminal at varying tree depths might have solved our problem but this alternative was not chosen by evolution (as shown below). Material value and material change are considered, to allow the player to make choices involving not losing pieces.

Mate terminals These were specifically constructed for the mating problem (Table 6.4). Some of these terminals resemble those from the function set to allow building different calculations with similar (important) units.

6.3.3 Two sample search algorithms

Herein we delineate two examples to demonstrate the capability of constructing mate-searching algorithms using our function and terminal sets. Consider Algorithm 6.2, a simple LISP-like program for solving mate-in-2.

When this program receives as input all boards (one at a time) after the player has moved, it will return the maximal score for the board for which the move leading to a mate (i.e., the correct move) was played. Note that the function IfForAllOpponentMoves also takes care to undo moves.

The recursive Algorithm 6.3 demonstrates how we can construct the entire basic algorithm (Algorithm 6.1).[1]

[1]The main difference between the algorithm shown here and Algorithm 6.1 (apart from returning real values) is that the main function checks the opponent's moves (since it deals with the board after the player has moved, as explained above), and the auxiliary function checks the player's moves. This is but a minor difference.

Table 6.2 Domain-specific function set of an individual program in the population. B: Boolean, F: Float. Note: all move-making functions undo the move when the function terminates.

Node name	Type	Return value
`IfMyMoveExistsSuchThat`(B,F_1,F_2)	F	If after making one of my moves B is true, make that move and return F_1, else return F_2
`IfForAllOpponentMoves`(B,F_1,F_2)	F	If after making each of the opponent's moves B is true, make an opponent's move and return F_1, else return F_2
`MakeBestMove`(F)	F	Make all moves possible, evaluate the child (F) after each move, and return the maximal (or minimal) value of all evaluations
`MakeAllMovesSuchThat`(B,F_1,F_2)	F	Make all possible moves, and remember those for which B was true. Evaluate F_1 after making each of these moves, and return the best result. If no such move exists, return F_2
`IfExistsCheckingMove`(F_1,F_2)	F	If a checking move exists, return F_1, else return F_2
`MyMoveIter`(B_1,B_2,F_1,F_2)	F	Find a player's move for which B_1 is true. Then, make all the opponent's moves, and check if for all, B_2 is true. If so, return F_1, else return F_2
`IfKingMustMove`(F_1,F_2)	F	If opponent's king must move, make a move, and return F_1, else return F_2
`IfCaptureCloseToKingMove`(F_1,F_2)	F	If player can capture close to king, make that move and return F_1, else return F_2
`IfPinCloseToKingMove`(F_1,F_2)	F	If player can pin a piece close to opponent's king, make that move and return F_1, else return F_2
`IfAttackingKingMove`(F_1,F_2)	F	If player can move a piece into a square attacking the area near opponent's king, make that move and return F_1, else return F_2
`IfClearingWayMove`(F_1,F_2)	F	If player can move a piece in such a way that another piece can check next turn, return F_1, else return F_2
`IfSuicideCheck`(B,F_1,F_2)	F	If player can check the opponent's king while losing its own piece and B is true, evaluate F_1, else return F_2

Table 6.3 Chess terminal set. Opp: opponent.

Node name	Type	Return value
`IsCheck`	B	Is the opponent's king being checked?
`OppKingProximityToEdges`	F	The player's king's proximity to the edges of the board
`NumOppPiecesAttacked`	F	The number of the opponent's attacked pieces close to its king
`IsCheckFork`	B	Is the player creating a fork attacking the opponent's king?
`NumNotMovesOppKing`	F	The number of illegal moves for the opponent's king
`NumNotMovesOppBigger`	B	Has the number of illegal moves for the opponent's king increased?
`IsOppKingProtectingPiece`	B	Is the opponent's king protecting one of its pieces?
`EvaluateMaterial`	F	The material value of the board
`IsMaterialChange`	B	Was the last move a capture move?
`IsMateInOneOrLess`	B	Is the opponent in mate, or can be in the next turn?
`IsOppKingStuck`	B	Do all legal moves for the opponent's king advance it closer to the edges?
`IsOppPiecePinned`	B	Is one or more of the opponent's pieces pinned?

Table 6.4 Mate terminal set. Opp: opponent, My: player.

Node name	Type	Return value
`IsNextMoveForced`	B	Is the opponent's next move forced (only one possible)?
`IsNextMoveForcedWithKing`	F	Opponent must move its king
`IsPinCloseToKing`	B	Is an opponent's piece pinned close to the king?
`NumMyPiecesCanCheck`	F	Number of the player's pieces capable of checking the opponent
`DidNumAttackingKingIncrease`	B	Did the number of pieces attacking the opponent's king's area increase after last move?
`IsPinCloseToKing`	B	Is an opponent's piece pinned close to the king?
`IsDiscoveredCheck`	B	Did the last move clear the way for another piece to check?
`IsDiscoveredProtectedCheck`	B	Same as above, only the checking piece is also protected

Algorithm 6.2 ScoreMove(*board*,*depth*)

```
1: if IsMate(board) then
2:    return 1000.0
3: else
4:    if depth = 0 then
5:       return -1000.0
6:    else
7:       return IfForAllOpponentMoves(IsMateInOne, 1000.0, -1000.0)
8:    end if
9: end if
```

Algorithm 6.3 ScoreMove(*board*,*depth*)

```
   // mate-in-$n$ constructed with the building blocks presented in this chapter
1: if IsMate?(board) then
2:    return 1000.0
3: else if $depth = 0$ then
4:    return -1000.0
5: else
6:    return IfForAllOpponentMoves(CheckCondition(board,depth), 1000.0, -1000.0)
7: end if
Procedure CheckCondition(board,depth)
8: return IfMyMoveExistsSuchThat(Equal(ScoreMove(board,$depth-1$),1000.0), 1000.0,
       -1000.0)
```

Note that this is just the basic algorithm, which implements exhaustive search with no enhancements or knowledge.

6.3.4 Fitness

In order to test our individuals and assign fitness values we used a pool of 100 mate-in-*n* problems of varying depths (i.e., values of *n*). The easier 50 problems ($n = 1, 2, 3$) were taken from Polgar's Book (Polgar [139]), while those with larger *n* values ($n \geq 4$) were taken from various issues of the Israeli Chess Federation Newsletter.[2]

Special care was taken to ensure that all of the deeper problems could not be solved trivially (e.g., if there are only a few pieces left on the board, or when the opponent's king can be easily pushed towards the edges). We used Crafty's feature of counting nodes in the game tree, and made sure that the amount of search required to solve all problems was close to the average values given in Table 6.1.

The fitness score was assigned according to an individual's (search algorithm's) success in solving a random sample of problems of all depths, taken from the pool. An individual's fitness was defined as:

$$fitness = \sum_{i=1}^{s \cdot Max_n} Correctness_i \cdot 2^{n_i} \cdot Boards_i,$$

where:

- i, n, and s are the problem, the depth, and the sample size (set to 5 per n), respectively. Max_n is the maximal depth we worked with (which was 5).
- $Correctness_i \in [0, 1]$ represents the correctness of the move, with a right move

[2] `http://www.chess.org.il`

ascribed a score of 1.0 and a wrong move ascribed a score of 0.0. If the correct piece was selected, this score was 0.5^d, where d is the distance (in squares) between the correct destination and the chosen destination for the piece. If the correct square was attacked, but with the wrong piece, *Correctness* was set to 0.1. In the later stages of each run (after more than 75% of the problems were solved by the best individuals), this fitness component was either 0.0 or 1.0.

- n_i is the depth of problem i. The difficulty of finding mating moves increases exponentially with depth, hence the exponent in the second component of the sum.

- $Boards_i$ is the number of boards examined by Crafty for this problem, divided by the number examined by the individual.[3] For small n values this factor was only used at later stages of evolution.

We used the standard reproduction, crossover, and mutation operators. The major evolutionary parameters were: population size—70, generation count—between 100 and 150, reproduction probability—0.35, crossover probability—0.5, and mutation probability—0.15 (including Ephemeral Random Constants—ERCs). The relatively small population size helped to maintain shorter runtimes, although in this manner possibly more runs were needed to attain our results.

6.4 Results

After each run we extracted the top individual (i.e., the one that obtained the best fitness throughout the run) and tested its performance on a separate problem set (the *test set*), containing 10 unseen problems per depth. The results from the ten best runs show that all problems up to $n = 4$ were solved completely in most of the runs, and most $n = 5$ problems were also solved.

We do not present herein a detailed analysis of runs but focus on the most important issue, namely, the number of nodes examined by our evolved search algorithms. As stated above, mates can be found with exhaustive search and little knowledge, but the number of nodes would be prohibitive. Table 6.5 compares the number of nodes examined by our best evolved algorithm with the number of nodes required by Crafty. As can be seen, a reduction of 47% is achieved for the most difficult case ($n = 5$). Note that *improvement is not over the basic alpha-beta algorithm, but over a world-class program using all standard enhancements*.

We did not carry out, at this juncture, a comparison of search runtimes, as our evolved programs were far from being programmatically optimal (i.e., in terms of

[3]In order to better control runtimes, if an individual examined more than 1.5 times the number of boards examined by Crafty, the search tree was truncated, although the returned score was still used.

Table 6.5 Solving the mate-in-n problem: Number of nodes examined by Crafty compared with the number of nodes required by our best evolved individual from over 20 runs. Values shown are averaged over the test problems. As can be seen, for the hardest problems ($n = 5$) our evolved search algorithm obtains a 47% reduction in the number of search nodes.

mate-in	1	2	3	4	5
Crafty	600	7K	50K	138K	1.6M
Evolved	600	2k	28k	55K	850k

compilation, data structures, etc.). The main thrust herein was to demonstrate that far fewer boards need to be examined by evolving a search algorithm, which might ultimately lead to improved chess engines.

6.5 Discussion

Our results show that the number of nodes required to find mates may be significantly reduced by evolving a search algorithm with building blocks that provide a priori knowledge. This is reminiscent of human thinking, since human players survey very few boards (typically 1 or 2 per second) but apply knowledge far more complex than any artificial evaluation function. On the other hand, even strong human players usually do not find mates as fast as machines (especially in complex positions).

The GP trees of our best evolved individuals were quite large and difficult to analyze. However, from examining the results it is clear that the best individuals' search was efficient, and we thus conclude that domain-specific functions and terminals play an important role in guiding search. This implies that much "knowledge" was incorporated into stronger individuals, although it would be difficult to qualify it.

The depths (n values) we dealt with are still relatively small. However, as the notion of evolving the entire search algorithm is new, we expect that better results might be achieved in the future. In the short term we would like to evolve a general mate-in-n module, which could replace a chess engine's current module, thereby increasing its rating—no mean feat where top-of-the-line engines are concerned!

In the longer term we intend to seek ways of combining the algorithms evolved here into an algorithm playing the entire game. The search algorithms we evolved may provide a framework for searching generic chess positions (not only finding mates). Learning how to combine this search with the evaluation functions we previously developed may give rise to stronger (evolved) chess players.

Ultimately, our approach could prove useful in every domain in which knowledge is coupled with search. The GP paradigm still harbors great untapped potential in constructing and applying knowledge.

Chapter 7

Backgammon

Backgammon is a two-player board game that differs in one important aspect from checkers and chess: there is an element of chance involving a throw of dice. The application of machine-learning techniques to obtain strong backgammon players has been done both in academia and industry. Top commercial products include Jellyfish (Dahl [45]) and TD-Gammon (Tesauro [179]).

The majority of learning software for backgammon is based on artificial neural networks, which usually receive as input the board configuration and produce as output the suggested best next move. A prime advantage of GP, which we apply herein, over artificial neural networks, is the automatic development of structure.

7.1 Previous Work

In 1989, Tesauro [177] presented Neurogammon, a neural-network player evolved using supervised learning and several handcrafted input features of the backgammon game. This work eventually led to TD-Gammon, one of the top commercial products (Tesauro [179]). This work is based on the Temporal Difference (TD) method, used to train a neural network through a self-playing model, i.e., learning is accomplished by neural networks playing against themselves and thus improving.[1]

Pollack et al. [142] presented HC-Gammon, a much simpler Hill-Climbing algorithm that also used neural networks. Under their model the current network is declared "Champion" and by adding Gaussian noise to the biases of this champion network a "Challenger" is created. The Champion and the Challenger then engage in a short tournament of backgammon; if the Challenger outperforms the Champion, small changes are made to the Champion biases in the direction of the Challenger

[1]Self-play in temporal difference learning is what we referred to as coevolution in evolutionary algorithms. This is simply a terminological difference between the two fields (Runarsson and Lucas [150]).

biases.

Another interesting work is that of Sanner et al. [156], whose approach was based on cognition (specifically, on the ACT-R theory of cognition (Anderson and Lebiere [6])). Rather than trying to analyze the exact board state, they defined a representational abstraction of the domain, consisting of general backgammon features such as blocking, exposing, and attacking. They maintained a database of feature neighborhoods, recording the statistics of winning and losing for each such neighborhood. All possible moves were encoded as sets of the above features; then, the move with the highest win probability (according to the record obtained so far) was selected.

Darwen [46] studied the coevolution of backgammon players using single- and multi-node neural networks, focusing on whether non-linear functions could be discovered. He concluded that with coevolution there is no advantage in using multi-node networks, and that coevolution is not capable of evolving non-linear solutions.

Finally, Qi and Sun [144] presented a genetic algorithm-based multi-agent reinforcement learning bidding approach (GMARLB). The system comprises several evolving teams, each team composed of a number of agents. The agents learn through reinforcement using the Q-learning algorithm. Each agent has two modules, Q and CQ. At any given moment only one member of the team is in control—and chooses the next action for the whole team. The Q module selects the actions to be performed at each step, while the CQ module determines whether the agent should continue to be in or relinquish control. Once an agent relinquishes control, a new agent is selected through a bidding process, whereby the member who bids highest becomes the new member-in-control.

7.2 Evolving Backgammon-Playing Strategies

As with checkers and chess we used Strongly Typed GP (Chapter 2). Below we delineate our setup (Azaria and Sipper [8, 9]).

7.2.1 Board evaluation

Tesauro [179] noted that due to the presence of stochasticity in the form of dice, backgammon has a high branching factor (about 20 moves on average for each of the 21 dice rolls), therefore rendering deep search strategies impractical. Thus, we opted for the use of a flat evaluator: after rolling the dice, generate all possible next-move boards, evaluate each of them, and finally select the board with the highest score.

This approach has been widely used by neural network-based players and—as shown below—it can be used successfully with GP. In our model, each individual is a LISP program that receives a backgammon board configuration as input and returns a real number that represents the board's score.

An artificial player is attained by combining an (evolved) board evaluator with a program that generates all next-moves given the dice values.

7.2.2 Program architecture

The game of backgammon can be observed to consist of two main stages: the "contact" stage, where the two players can hit each other, and the "race" stage, where there is no contact between the two players. During the contact stage we expect a good strategy to block the opponent's progress and minimize the probably of getting hit. On the other hand, during the race stage, blocks and blots are of no import, rather, one aims to select moves that lead to the removal of a maximum number of pieces off the board.

This observation directed us in designing the genomic structure of individuals in the population. Each individual contained a contact tree and a race tree. When a board was evaluated, the program checked whether there was any contact between the players and then evaluated the tree that was applicable to the current board state. The terminal set of the contact tree was richer and contained various general and specific board query functions. The terminal set of the race tree was much smaller and contained only terminals that examined the checkers' positions. This is because at the race phase the moves of each player are mostly independent of the opponent's status and thus are much simpler.

One can argue that since the strategies of the two stages of the game are independent, it would be better to train contact and race individuals independently. However, the final product of the evolutionary process is a complete individual that needs to win complete games, and not only one of the game stages. For example, to train a race individual would require generating unnatural board race configurations that would not represent the complete wide range of starting race configurations a backgammon game can produce. Therefore, it seemed more natural to train the individuals for both stages together.

7.2.3 Functions and terminals

We used two atomic types: *Float* and *Boolean*. We also used one set type—*Query*—that included both atomic types. We defined three types of terminals:

- `Float-ERC` is a real-valued Ephemeral Random Constant (Chapter 3). When created, the terminal is assigned a constant, real-number value, which becomes the return value of the terminal.

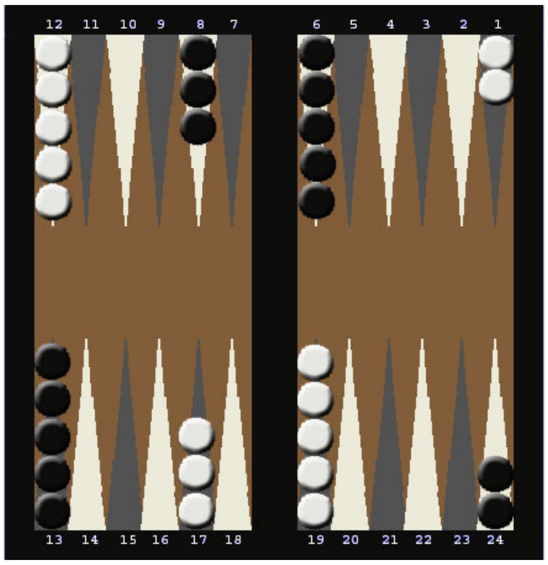


Figure 7.1 Initial backgammon configuration. The White player's home positions are labeled 19-24, and the Black player's home positions are labeled 1-6.

- The board-position query terminals use the `ERC` mechanism to query a specific location on the board. When initialized, an integer value between 0 and 25 is randomly chosen, where 0 specifies the bar location, 1–24 specify the inner board locations, and 25 specifies the off-board location (Figure 7.1). The term Player refers to the contender whose turn it is, while Enemy refers to the opponent. When a board query terminal is evaluated, it refers to the board location that is associated with the terminal, from the Player's point of view.

- In defining the last type of terminal we took advantage of one of GP's most powerful attributes: The ability to easily add non-trivial functionalities that provide useful information about the domain environment. In our case, these terminals provided general information about the board as a whole.

The terminal set for contact trees is given in Table 7.1 and that for race trees in Table 7.2.

The function set contained no domain-specific operators, but only arithmetic and logic ones, so we used the same function set both for contact and race trees. The function set is given in Table 7.3.

7.2.4 Fitness and setup

Our first approach to measuring fitness was based on an external opponent in the role of a "teacher". As external opponent we used *Pubeval*, a free, public-domain board evaluation function written by Tesauro [178].[2] The program—which plays

[2]Commercial products do not have suitable interfaces for benchmarking nor are there published results concerning their performance against other programs.

Table 7.1 Terminal set of the contact tree. B: Boolean, F: Float, Q: Query.

Node name	Type	Return value
`Float-ERC`	F	A random real constant in the range $[0, 5]$
`Player-Exposed`(n)	Q	If Player has exactly one checker at location n, return 1, else return 0
`Player-Blocked`(n)	Q	If Player has two or more checkers at location n, return 1, else return 0
`Player-Tower`(n)	Q	If Player has h or more checkers at location n (where $h \geq 3$), return $h - 2$, else return 0
`Enemy-Exposed`(n)	Q	If Enemy has exactly one checker at location n, return 1, else return 0
`Enemy-Blocked`(n)	Q	If Enemy has two or more checkers at location n, return 1, else return 0
`Player-Pip`	F	Return Player pip-count divided by 167 (pip-count is the number of steps a player needs to move in order to win the game; this value is normalized through division by 167—the pip-count at the beginning of the game)
`Enemy-Pip`	F	Return Enemy pip-count divided by 167
`Total-Hit-Prob`	F	Return sum of hit probabilities over all exposed Player checkers
`Player-Escape`	F	Measure the effectiveness of the Enemy's barrier over its home positions. For each Enemy home position that does not contain an Enemy block, count the number of dice rolls that could potentially lead to the Player's escape. This value is normalized through division by 131—the number of ways the Player can escape when the Enemy has no blocks
`Enemy-Escape`	F	Measure the effectiveness of the Player's barrier over its home positions using the same method as above

Table 7.2 Terminal set of the race tree.

Node name	Type	Return value
`Float-ERC`	F	A random real constant in the range $[0, 5]$
`Player-Position`(n)	Q	Return number of checkers at location n

Table 7.3 Function set of the contact and race trees. B: Boolean, F: Float.

Node name	Type	Return value
`Add`(F_1,F_2)	F	Add two real numbers
`Sub`(F_1,F_2)	F	Subtract two real numbers
`Mul`(F_1,F_2)	F	Multiply two real numbers
`If`(B,F_1,F_2)	F	If B evaluates to a non-zero value, return F_1, else return F_2
`Greater`(F_1,F_2)	B	If F_1 is greater than F_2, return 1, else return 0
`Smaller`(F_1,F_2)	B	If F_1 is smaller than F_2, return 1, else return 0
`And`(B_1,B_2)	B	If both arguments evaluate to a non-zero value, return 1, else return 0
`Or`(B_1,B_2)	B	If at least one of the arguments evaluates to a non-zero value, return 1, else return 0
`Not`(B)	B	If B evaluates to zero, return 1, else return 0

decently—seems to have become the de facto yardstick used by the growing community of backgammon-playing program developers. Several researchers in the field have pitted their own creations against Pubeval.

To evaluate fitness, we let each individual (backgammon strategy) play a 100-game tournament against Pubeval. Fitness was then the individual's score divided by the sum of scores of both players (individual and Pubeval). A game's score was 3 points per backgammon, 2 points per gammon, and 1 point per regular win (attaining 2 or 3 points was very rare).

We used the standard reproduction, crossover, and mutation operators, tournament selection, a population size of 128, and a generation count of 500. Every five generations we pitted the four individuals with the highest fitness in a 1000-game tournament against Pubeval (that is, 10 times the number of games used during fitness evaluation), and the individual with the highest score in these tournaments, over the entire evolutionary run, was declared best-of-run.

7.3 Results: External opponent

We repeated each experiment 20 times. For each performance measure we computed the average, minimum, and maximum values of the best-fitness individual every five generations over the 20 runs. Figure 7.2 shows the average fitness results.

A prima facie observation might lead to the conclusion that these results were remarkable; indeed, scoring over 60% in a backgammon tournament against Pubeval is an exceptional result that was far beyond the highest result ever published. Unfortunately, fitness was computed using tournaments of 100 games, too short for a backgammon player benchmark.

In order to obtain a better indication of performance, we had the best-of-

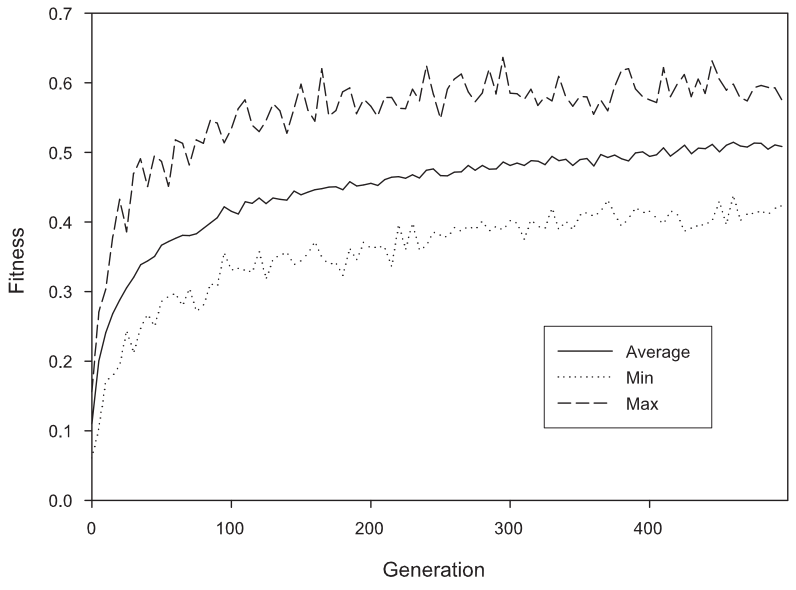


Figure 7.2 Fitness curve when using an external opponent, averaged over 20 runs.

generation individual (according to fitness) play a *1000*-game tournament against Pubeval. Figure 7.3 shows the results of this benchmark, where performance is seen to drop well below the 50% mark.

The results displayed in Figure 7.3, being more indicative of performance, raise the question of whether better players can be had. We answer in the affirmative in the next section.

7.4 Coevolution

One might think that when training against an external opponent, evolved individuals would be able to overpower this opponent, i.e., win above 50% of the games—a thought not borne out by the results. Moreover, the evolved individuals were probably overfitted to the strategy of Pubeval, casting doubt on their generalization capabilities.

This observation led us to the next phase of experimentation: Instead of playing against an external opponent, individuals played against each other, in a coevolutionary manner. The fitness of an individual was relative to its cohorts in the population. To avoid overly lengthy evaluation times, methods such as round robin—where each individual is pitted against all others—were avoided. Through experimentation we concluded that a good evaluation method was the single elimination tournament: Start with a population of n individuals, n being a power of two. Then, divide the individuals into $\frac{n}{2}$ arbitrary pairs, and let each pair engage in a relatively short tournament of 50 games. Finally, set the fitness of the $\frac{n}{2}$ losers to $\frac{1}{n}$. The remaining $\frac{n}{2}$

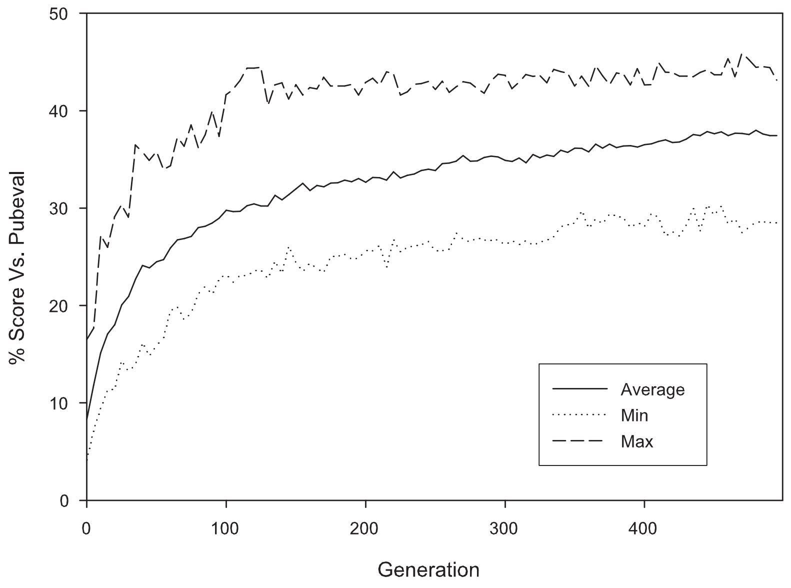


Figure 7.3 Benchmark results when using an external opponent, averaged over 20 runs.

winners are divided into pairs again, engage in tournaments as before, and the losers are assigned fitness values of $\frac{1}{n/2}$. This process continues until one champion individual remains (which is assigned a fitness value of 1). Thus, the more tournaments an individual survives, the higher its fitness.

One of the properties of single elimination tournament is that half of the population is always assigned the same low fitness. Although there is a certain "injustice" in having relatively good individuals receive the same fitness as others with poorer performance, this method proved advantageous. Our preliminary experiments with "fairer" methods, such as round robin, showed that they led to premature convergence because bad individuals were rarely selected, and preserving a small number of low-performance individuals (as does single elimination tournament) affords the discovery of new strategies. On the other hand, an individual must exhibit a consistently good strategy in order to attain high fitness, and thus we are very likely to preserve good strategies.

To allow us to compare the performance of both learning methods as accurately as possible, we kept most GP parameters identical to the external-opponent case. Figure 7.4 shows the benchmark results vs. Pubeval of the individuals evolved through coevolution. Again, the benchmark score of an individual is the score it obtained in a 1000-game tournament against Pubeval, divided by the sum of the scores obtained by both players (the individual and Pubeval). Table 7.4 shows how our top evolved players fared against Pubeval, alongside the performance of the other approaches described in Section 7.1.

Wishing to improve our results yet further we employed a distributed asynchronous island model. In this experiment we used 50 islands, designated Island-0

Table 7.4 Comparison of backgammon players. GP-Gammon-i designates the best GP strategy evolved at run i, which was tested in a tournament of 1000 games against Pubeval. (In comparison, GMARLB-Gammon used 50 games for evaluation, ACT-R-Gammon used 5000 games, Darwen used 10,000 games, and HC-Gammon used 200 games.) "Wins" refers to the percentage of wins against Pubeval.

Rank	Player	Wins	Rank	Player	Wins
1	GP-Gammon-1	56.8[a]	13	GP-Gammon-12	51.4
2	GP-Gammon-2	56.6	14	GMARLB-Gammon	51.2[b]
3	GP-Gammon-3	56.4	15	GP-Gammon-13	51.2
4	GP-Gammon-4	55.7	16	GP-Gammon-14	49.9
5	GP-Gammon-5	54.6	17	GP-Gammon-15	49.9
6	GP-Gammon-6	54.5	18	GP-Gammon-16	49.0
7	GP-Gammon-7	54.2	19	GP-Gammon-17	48.1
8	GP-Gammon-8	54.2	20	GP-Gammon-18	47.8
9	GP-Gammon-9	53.4	21	ACT-R-Gammon	45.94
10	GP-Gammon-10	53.3	22	GP-Gammon-19	45.2
11	GP-Gammon-11	52.9	23	GP-Gammon-20	45.1
12	Darwen	52.7	24	HC-Gammon	40.00

[a] Sanner et al. [156] quoted a paper by Galperin and Viola, who used TD(λ) training to purportedly obtain players with win percentage 59.25 against Pubeval. The reference for Galperin and Viola is of a now-obsolete URL, and all our efforts to obtain the paper by other means came to naught. Moreover, it seems to be but a short project summary and not a bona fide paper with full experimental details. Thus, the article does not meet two necessary criteria of a valid scientific publication: availability and repeatability. We have therefore not included their result herein (which, in any case, we surpassed—see Table 7.5).

[b] This is an average value over a number of runs. The authors cited a best value of 56%, apparently a fitness peak obtained during one evolutionary run, computed over *50 games*. This is too short a tournament and hence we cite their average value. Indeed, we were able to obtain win percentages of over 65% for randomly selected strategies over 50-game tournaments, a result that dwindled to 40–45% when the tournament was extended to 1000 games.

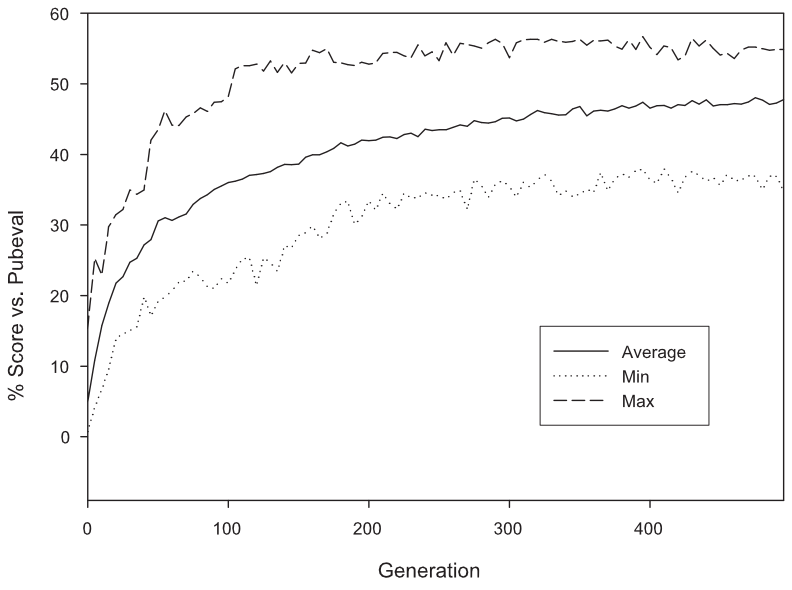


Figure 7.4 Benchmark results when using coevolution.

Table 7.5 Using the island model. I-GP-Gammon-*i* designates the best GP strategy evolved at distributed run *i*, each of which was tested in a tournament of 1000 games against Pubeval.

Rank	Player	Wins
1	I-GP-Gammon-1	62.4
2	I-GP-Gammon-2	62.2
3	I-GP-Gammon-3	62.1
4	I-GP-Gammon-4	62.0
5	I-GP-Gammon-5	61.4
6	I-GP-Gammon-6	61.2
7	I-GP-Gammon-7	59.1

through Island-49. Starting at generation 10, for each generation n, every Island i that satisfied $i \, mod \, 10 = n \, mod \, 10$ migrated 4 individuals to each of the 3 adjacent neighbors (a total of 12). Individuals were selected for migration based on fitness using tournament selection with repeats. The rest of the setup was identical to that of Section 7.4. Table 7.5 shows the improved results.

To get an idea of the human-competitiveness of our evolved players, we obtained statistics from the HC-Gammon homepage (Pollack et al. [141]) of games played by HC-Gammon against human players. Accordingly, HC-Gammon won 58% of the games when counting abounded games as wins, and 38% when disregarding them. Considering that HC-Gammon won 40% of the games vs. Pubeval we expect, by transitivity, that our 62%-vs-Pubeval GP-Gammon is a very strong player in human terms.

One would expect that strategies evolved using an external opponent and tested against the same program would perform much better (with respect to the benchmark program) than strategies that have been evolved without any prior knowledge of the benchmark strategy. Surprisingly, this was not the case here—it is clear that the performance of the coevolutionary approach was much better than the external-opponent approach.

In order to explain these results we should examine the learning model of both approaches. The population size and initialization method were the same for both, as were selection, crossover, and mutation. The only difference lay in the fitness measure. With an external opponent, each individual was measured only by playing against Pubeval, which is known to be a decent player, but still has its own strengths and weaknesses.

Backgammon players that gain experience by playing only with one other player, who does not improve and has only one fixed reply for each game configuration, are likely to form a strategy adapted to this particular environment, i.e., to the external opponent's specific strategy, achieving a moderate score against it. However, in order to gain a significant and consistent advantage over the external opponent, a new strategy needs to be discovered. As it turns out, the individuals were unable to discover such a novel strategy by playing only against Pubeval, and therefore they converged to a moderate level of performance.

On the other hand, with coevolution, individuals exhibiting good performance are likely to play against two or more different opponents at each generation. Moreover, the term "good performance" is relative to the performance of the other individuals in the population and not to those of an external opponent, which performs much better at the beginning of evolution.

A human playing against many different opponents would probably fare better than one who has learned only from a single teacher, due to the fact that the former is exposed to many strategies and thus must develop responses to a wide range of game conditions. In terms of evolution, considering our domain, the fitness of an individual measured by playing backgammon against a variety of other individuals is likely to be more reliable than fitness measured by playing only against Pubeval.

7.5 Playing Backgammon the GP Way

Wearing once again the "molecular biologist" hat, as we did for chess in Chapter 5, we examined many evolved individuals and discovered some interesting behaviors and regularities.

Recall that our terminal set contained two types of board-query functions: those that performed specific board-position queries (e.g., `Player-Exposed` and `Player-Blocked`), and those that performed general board queries (e.g., `Enemy-Escape` and `Total-Hit-Prob`). These latter are more powerful, and, in fact, some of them can be used as stand-alone heuristics (albeit very weak) for playing backgammon.

We observed that general query functions were more common than position-specific functions. Furthermore, GP-evolved strategies seemed to “ignore” some board positions. This should come as no surprise: the general functions provide useful information during most of the game, thus inducing GP to make use of them often. In contrast, information pertaining to a specific board position has less effect on overall performance, and is relevant only at a few specific moves during the game.

We surmised that the general functions form the lion’s share of an evolved backgammon strategy, with specific functions used to balance the strategy by catering for (infrequently encountered) situations. In some sense GP strategies were reminiscent of human game-playing: humans rely on general heuristics (e.g., avoid hits, build effective barriers), whereas local decisions are made only in specific cases. (As noted above, the issue of human cognition in backgammon was central to the paper by Sanner et al. [156].)

7.6 Discussion

We have shown how GP can tackle a board game that has an inherent element of chance. Our model divided the backgammon game into two main stages, thus entailing two types of trees. A natural question arising is that of refining this two-fold division into more sub-stages. The game dynamics may indeed call for such a refined division, with added functions and terminals specific to each game stage.

However, it is unclear how this refining is to be had: Any (human) suggestion beyond the obvious two-stage division is far from being obvious—or correct. One possible avenue of future research is to let GP handle this question altogether and evolve the stages themselves. For example, we might use a main tree to inspect the current board configuration and decide which tree should be used for the current move selection. These “specific” trees would have their own separately evolving function and terminal sets. Automatically defined functions (Koza [109]) and architecture-altering operations (Koza et al. [110]) might well come in handy here.

Having had such a productive experience with board games, it is now time to walk off the board…

Part III

Simulation Games

I wonder how long handcoded algorithms will remain on top.

—Developer's comment at a Robocode discussion group

`http://old.robowiki.net/robowiki?GeneticProgramming`

Chapter 8

Robocode

This part of the book focuses on games that are not played between two players manipulating pieces on a board, but rather between many players in a simulated environment.

Program-based games are a subset of the domain of games in which the human player has no direct influence on the course of the game; rather, the actions during the game are controlled by programs that were written by the (usually human) programmer. The program responds to the current game environment, as captured by its percepts, in order to act within the simulated game world. The winner of such a game is the programmer who has provided the best program; hence, the programming of game strategies is often used to measure the performance of AI algorithms and methodologies. Some famous examples of program-based games are RoboCup,[1] the robotic soccer world championship, and CoreWars,[2] in which assembly-like programs struggle for limited computer resources.

This chapter tackles the game of Robocode,[3] a simulation-based game in which robotic tanks fight to destruction in a closed arena (Shichel et al. [162]). The programmers implement their robots in the Java programming language, and can test their creations either by using a graphical environment in which battles are held, or by submitting them to a central web site where online tournaments regularly take place; this latter enables the assignment of a relative ranking by an absolute yardstick, as is done, e.g., by the Chess Federation. The game has attracted hundreds of human programmers and their submitted strategies show much originality, diversity, and ingenuity. Since the vast majority of Robocode strategies submitted to the league were coded by hand, GP will herein be competing directly with human programmers.

Robocode seems to have attracted little attention from the evolutionary computation community. Eisenstein [53] described the evolution of Robocode players using

[1]`http://www.robocup.org`

[2]`http://corewars.sourceforge.net`

[3]`http://robocode.sourceforge.net`

a fixed-length genome to represent networks of interconnected computational units, which performed simple arithmetic operations. Each element took its input either from the robot's sensors or from another computational unit. Eisenstein was able to evolve Robocode players, each able to defeat a single opponent, but was not able to generalize his method to create players that could beat numerous adversaries and thus hold their own in the international league. This latter failure may be due either to problems with the methodology or to lack of computational resources—no conclusions were provided.

More recently, Harper [74] used Grammatical Evolution to evolve Java programs to control a Robocode robot. He demonstrated how Grammatical Evolution together with spatial coevolution in age layered planes (SCALP) could harness coevolution to evolve relatively complex behavior, including robots capable of beating Robocode's sample robots as well as some more complex human-coded robots.

8.1 The Robocode Simulator

A Robocode (tank) player is written as an event-driven Java program. A main loop controls the tank activities, which can be interrupted on various occasions, called *events*. Whenever an event takes place, a special code segment is activated, according to the given event. For example, when a tank bumps into a wall, the HitWallEvent will be handled, activating a function named onHitWall. Other events include: hitting another tank, spotting another tank, and getting hit by an enemy shell.

There are five actuators controlling the tank: movement actuator (forward and backward), tank-body rotation actuator, gun-rotation actuator, radar-rotation actuator, and fire actuator (which acts both as a trigger and a firepower controller).

At the beginning of a combat round each tank of the several placed in the arena is assigned a fixed value of energy. When the energy meter drops to zero, the tank is disabled, and—if hit—is immediately destroyed. During the course of the match, energy levels may increase or decrease: a tank gains energy by firing shells and hitting other tanks, and loses energy by getting hit by shells, other tanks, or walls. Firing shells costs energy. The energy lost when firing a shell, or gained, in case of a successful hit, is proportional to the firepower of the fired shell.

A round ends when only one tank remains in the battlefield (or no tanks at all), whereupon the participants are assigned scores that reflect their performance during the round. A battle lasts a fixed number of rounds. Figure 8.1 shows the battle scene.

In order to test our evolved Robocode players and compare them to human-written strategies, we submitted them to the international league.[4] The league comprises a number of divisions, classified mainly according to allowed code size. Specif-

[4]`http://robocode.yajags.com`

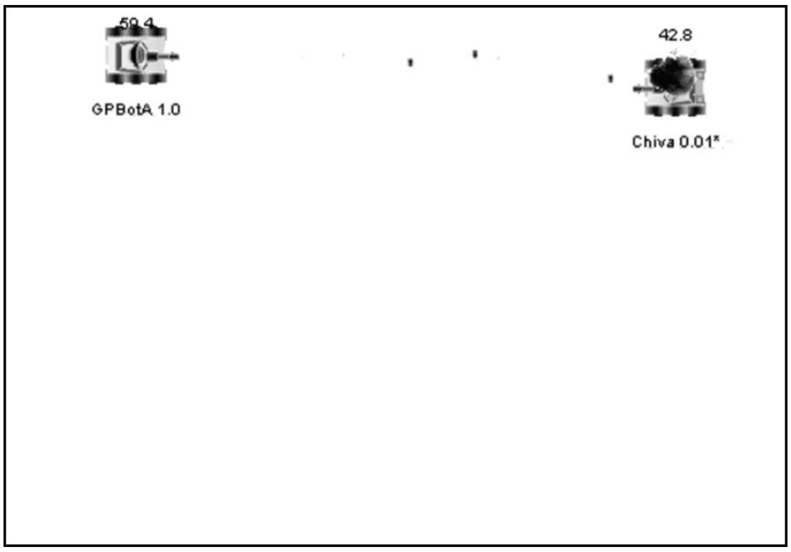


Figure 8.1 GP bot (left) fights an enemy (right) to the death (and lives to tell the tale).

ically, we aimed for the One-on-One HaikuBot challenge, in which the players play duels, and their code is limited to four instances of a semicolon (i.e., four lines of code), with no further restriction on code size. Since GP naturally produces long lines of code, this league seemed most appropriate for our research. Moreover, a code size-limited league places GP at a disadvantage, since, ceteris paribus, GP produces longer programs than humans due to bloat (Poli et al. [140]).

8.2 Evolving Robocode Strategies

We began by examining many different configurations for the various GP characteristics and parameters, including Strongly Typed GP (Chapter 2), which we used in previous chapters, and Automatically Define Functions (ADFs) (Koza [109]), which enable the evolution of subroutines. These techniques and a number of others proved not to be useful for the game of Robocode, and we concentrate below on a description of our winning strategy (Shichel et al. [162]).

8.2.1 Program architecture

We decided to use GP to evolve numerical expressions that would be given as arguments to the player's actuators. As noted above, our players consisted of only four lines of code (each ending with a semicolon). However, there was much variability in the layout of the code: we had to decide which events we wished to implement, and which actuators would be used for these events.

To obtain the strict code-line limit, we made the following adjustments:

- We omitted the radar-rotation command. The radar, mounted on the gun, was instructed to turn using the gun-rotation actuator.

- We implemented the fire actuator as a numerical constant, which could appear at any point within the evolved code sequence (see Table 8.1).

Our main loop contained one line of code that directed the robot to start turning the gun (and the mounted radar) to the right. This insured that within the first gun cycle an enemy tank would be spotted by the radar, triggering a ScannedRobotEvent. Within the code for this event three additional lines of code were added, each controlling a single actuator, and using a single numerical input that was evolved using GP. The first line instructed the tank to move to a distance specified by the first evolved argument. The second line instructed the tank to turn to an azimuth specified by the second evolved argument. The third line instructed the gun (and radar) to turn to an azimuth specified by the third evolved argument (Algorithm 8.1).

Algorithm 8.1 Robocode player's code layout

```
loop
  TurnGunRight // main code loop
end loop
...
OnScannedRobot() do
    MoveTank(<GP#1>)
    TurnTankRight(<GP#2>)
    TurnGunRight(<GP#3>)
end do
```

8.2.2 Functions and terminals

Since terminals can be treated as zero-argument functions, rather than divide the elemental components of the GP setup into functions and terminals, as is usually done, we focused on their functionality (Table 8.1):

1. *Game-status indicators:* A set of terminals that provided real-time information on the game status, such as last enemy azimuth, current tank position, and energy levels.

2. *Numerical constants:* The constant 0 and an Ephemeral Random Constant. This latter terminal was initialized to a random real numerical value in the range $[-1, 1]$, and did not change during evolution.

3. *Arithmetic and logic functions:* A set of zero- to four-argument functions, providing the standard support for basic arithmetic operations and conditional statements.

4. *Fire command:* This special function was used to conserve one line of code by not implementing the fire actuator in a dedicated line.

8.2.3 Fitness

When defining a fitness measure for our Robocode players we had two main considerations in mind: the opponents and the calculation of the fitness value itself.

A good Robocode player should be able to beat many different adversaries. Since the players in the online league differ in behavior, it is generally unwise to assign a fitness value according to a single-adversary match. On the other hand, it is unrealistic to do battle with the entire player set—not only is this a time-consuming task, but new adversaries enter the tournaments regularly. We tested several opponent set sizes, from one to five adversaries. Some of the tested evolutionary configurations involved a random selection of adversaries per individual or per generation, while other configurations consisted of a fixed group of adversaries. The configuration we ultimately chose to use involved a set of three adversaries—fixed throughout the evolutionary run—with unique behavioral patterns, which we downloaded from the top of the HaikuBot league. Since the game is nondeterministic, a total of three rounds was played versus each adversary to reduce the randomness factor of the results.

Given that fitness is crucial in determining the trajectory of the evolutionary process, it was essential to find a way to translate battle results into an appropriate fitness value. Our goal was to excel in the online tournaments; hence, we adopted the scoring algorithms used in these leagues. The basic scoring measure is the fractional score F, which is computed using the score gained by the player, S_P, and the score gained by its adversary, S_A:

$$F = \frac{S_P}{S_P + S_A}.$$

This method reflects the player's skills in relation to its opponent. It encourages the player not only to maximize its own score, but to do so at the expense of its adversary's. We observed that in early stages of evolution, most players attained a fitness of zero, because they could not gain a single point in the course of the battle. To boost population variance at early stages, we then devised a modified fitness measure $\widetilde{F}$:

$$\widetilde{F} = \frac{\epsilon + S_P}{\epsilon + S_P + S_A},$$

where ϵ is a small real-valued constant.

This measure is similar to the fractional-score measure, with one exception: when

Table 8.1 GP Robocode system: Functions and terminals.

Game-status indicators	
`Energy`	Player's remaining energy
`Heading`	Player's current heading
`X`	Player's current horizontal position
`Y`	Player's current vertical position
`MaxX`	Horizontal battlefield dimension
`MaxY`	Vertical battlefield dimension
`EnemyBearing`	Current enemy bearing, relative to current player heading
`EnemyDistance`	Current distance to the enemy
`EnemyVelocity`	Current enemy velocity
`EnemyHeading`	Current enemy heading, relative to current player heading
`EnemyEnergy`	Enemy's remaining energy
Numerical constants	
`Constant`	An ERC in the range $[-1,1]$
`Random`	A random real number in the range $[-1,1]$
`Zero`	The constant 0
Arithmetic and logic functions	
`Add`(x,y)	Adds x and y
`Sub`(x,y)	Subtracts y from x
`Mul`(x,y)	Multiplies x by y
`Div`(x,y)	Divides x by y, if y is nonzero; otherwise returns 0
`Abs`(x)	Absolute value of x
`Neg`(x)	Negative value of x
`Sin`(x)	The function $\sin(x)$
`Cos`(x)	The function $\cos(x)$
`ArcSin`(x)	The function $\arcsin(x)$
`ArcCos`(x)	The function $\arccos(x)$
`IfGreater`(x,y,exp_1,exp_2)	If x is greater than y returns the expression exp_1, otherwise returns the expression exp_2
`IfPositive`(x,exp_1,exp_2)	If x is positive, returns the expression exp_1, otherwise returns the expression exp_2
Fire command	
`Fire`(x)	If x is positive, executes a fire command with x being the firepower, and returns 1; otherwise, does nothing and returns 0

two evolved players obtain no points at all (most common during the first few generations), a higher fitness value will be assigned to the one that avoided its adversary best (i.e., lower S_A). This modified fitness function proved sufficient in enhancing population diversity during the initial phase of evolution. When facing multiple adversaries, we simply used the average value over the battles against each adversary.

8.2.4 Run parameters

- *Population size:* 256 individuals.
- *Generation count:* We did not set a limit for the generation count in our evolutionary runs. Instead, we simply stopped the run manually when the fitness value stopped improving for several generations.
- *Initial population:* We used Koza's ramped-half-and-half method (Koza [108]; Poli et al. [140]).
- *Crossover:* randomly select an internal node (with probability 0.9) or a leaf (with probability 0.1) from each tree, and exchange the subtrees rooted at these nodes. Bloat control was achieved using Langdon's method (Langdon [112]), which ensures that the resulting trees do not exceed the *maxdepth* parameter (set to 10).
- *Mutation:* randomly select one internal node (with probability 0.9) or leaf (with probability 0.1), delete the subtree rooted at that node, and grow a new subtree using the Grow method (Koza [108]). Bloat control was achieved by setting a *maxdepth* parameter (set to 10), and invoking the growth method with this limit.
- *Operator probabilities:* A probability of 0.95 of selecting the crossover operator, and 0.05 of selecting the mutation operator.
- *Selection method:* We used tournament selection, in which a group of individuals of size k (set to 5) is randomly chosen. The individual with the highest fitness value is then selected. In addition, we added elitism: The two highest-fitness individuals were passed on to the next generation with no modifications.

When an evolutionary run ended we needed to determine which of the evolved individuals could be considered the best. Since the game is highly nondeterministic, the fitness measure does not explicitly reflect the quality of the individual: a "lucky" individual might attain a higher fitness value than better overall individuals. In order to obtain a more accurate measure for the players evolved in the last generation, we let each of them do battle for 100 rounds against 12 different adversaries (one at a time). The results were used to extract the top player—to be submitted to the league.

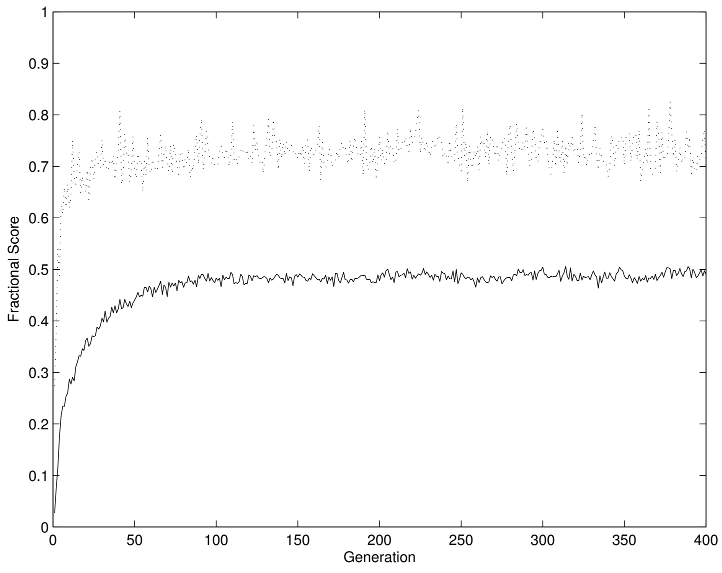


Figure 8.2 Modified fractional score (Section 8.2.3) averaged over three different adversaries, versus time (generations). Top (dotted) curve: best individual. Bottom (solid) curve: population average.

8.3 Results

We performed multiple evolutionary runs against three leading opponents, as described in Section 8.2.3. The progression of the best run is shown in Figure 8.2.

Due to the nondeterministic nature of the Robocode game and the relatively small number of rounds played by each individual, the average fitness is worthy of attention, in addition to the best fitness. The first observation to be made is that the average fractional score converged to a value of approximately 0.5, meaning that the average Robocode player was able to hold its own against its adversaries. When examining the average fitness, one should consider the variance: A player might defeat one opponent with a relatively high score, while losing to the two others.

Though an average fitness of 0.5 might not seem impressive, two comments are in order:

- This value reflects the average fitness of the population; some individuals attained much higher fitness.
- The adversaries used for fitness evaluation were excellent ones, taken from the top of the HaikuBot league. In the "real world" our evolved players faced a greater number of adversaries, most of them inferior to those used in the evolutionary process.

To join the HaikuBot challenge we extracted what we deemed to be the best individual of all runs and submitted it to the online league. At its very first tournament our GP bot came in third, later climbing to first place of 28.[5] All other 27 programs—defeated by our evolved bot—were written by humans (Table 8.2).

[5]`http://robocode.yajags.com/20050625/haiku-1v1.html`

Table 8.2 GP bot takes first place at HaikuBot league on June 25, 2005. The table's columns reflect various aspects of robotic behavior, such as *survival* and *bullet damage* measures. The final rank is determined by the *rating* measure, which reflects the performance of the robot in combat with randomly chosen adversaries.

			Total	Survi-	Last	Bullet		Ram				
Rank	Rating	Robot	Score	val	surv.	dmg.	Bonus	dmg.	Bonus	1sts	2nds	3rds
1	185.29	geep.haiku.GPBotC 1.0	32179	9450	1890	16012	2363	2077	377	197	125	0
2	168.79	kawigi.haiku.HaikuTrogdor 1.1	37822	12650	2530	19080	3292	233	29	255	67	0
3	141.67	cx.haiku.MeleeXaxa 1.0	32593	11000	2200	16022	2181	857	324	223	98	0
4	140.57	pez.femto.HaikuPoet 0.2	26241	9750	1950	12083	1862	569	21	202	121	0
5	126.76	kawigi.femto.FemtoTrogdor 1.0	31527	8800	1760	15337	2022	3138	462	187	133	0
6	120.27	ms.AresHaiku 0.4	29067	9050	1810	15881	2177	143	0	182	138	0
7	118.11	cr.OneOnOneHaiku 1.1	41943	9600	1920	22645	3185	3976	609	193	128	0
8	105.53	mz.HaikuGod 1.01	31604	11800	2360	12986	1835	2270	342	262	83	0
9	67.20	kawigi.haiku.HaikuChicken 1.0	24581	6900	1380	14462	1748	61	24	189	131	0
10	61.81	pez.femto.WallsPoetHaiku 0.1	25739	7950	1590	13039	1660	1323	168	163	161	0
11	60.79	kawigi.haiku.HaikuCircleBot 1.0	32831	10900	2180	16632	2675	344	89	222	98	0
12	36.17	soup.haiku.RammerHK 1.0	41258	7150	1430	21191	2260	8219	999	150	171	0
13	27.97	kawigi.haiku.HaikuSillyBot 1.2	23827	6850	1370	13524	1461	429	182	166	155	0
14	20.13	kawigi.haiku.HaikuLinearAimer 1.0	29473	7850	1570	17367	2349	273	54	164	158	0
15	11.13	cx.haiku.Escape 1.0	26110	10900	2180	10989	1856	177	0	222	101	0
16	7.40	cx.haiku.Xaxa 1.1	34483	11350	2270	17694	2967	180	14	230	93	0
17	-14.08	ahf.HaikuAndrew .1	19705	8250	1650	8650	1051	72	25	181	141	0
18	-24.32	soup.haiku.MirrorHK 1.0	21011	3950	790	14378	1347	513	23	141	182	0
19	-29.64	tango.haiku.HaikuTango 1.0	18769	4700	940	11631	1089	375	24	107	213	0
20	-32.14	soup.haiku.DodgeHK 1.0	22354	7850	1570	11374	941	563	48	158	162	0
21	-47.37	ms.ChaosHaiku 0.1	28704	7850	1570	15469	2141	1440	221	164	158	0
22	-76.73	cx.haiku.Smoku 1.1	22928	5300	1060	14495	1343	609	111	106	214	0
23	-87.01	klo.haikuBounC 1.0	21675	6000	1200	12952	1260	254	0	123	198	0
24	-87.36	soup.haiku.RandomHK 1.0	19712	4500	900	12982	1156	119	43	150	171	0
25	-136.47	soup.haiku.CutoffHK 1.0	26332	4250	850	13519	1247	5798	657	103	220	0
26	-177.37	davidalves.net.PhoenixHaiku 1.0	19896	5450	1090	12196	1018	135	0	127	196	0
27	-208.72	dummy.haiku.Disoriented 1.0	17946	4100	820	12171	775	75	0	86	234	0
28	-478.37	soup.haiku.WallDroidHK 1.0	6271	2050	410	3301	105	329	68	46	274	0

8.4 Discussion

All players in the HaikuBot league except for the GP bot were human-written computer programs. Thus we were able to show that GP could produce computer programs that are highly human competitive. This is even more impressive when one considers the complexity of the problem: the game of Robocode, being nondeterministic, continuous, and highly diverse (due to the unique nature of each contestant), induces a virtually infinite search space, making it an extremely complex (and thus interesting) challenge for the GPer.

When performing an evolutionary run against a single adversary, winning strategies were always evolved. However, these strategies were specialized for the given adversary: When playing against other opponents (even relatively inferior ones), the evolved players were usually beaten. Trying to avoid this obstacle, our evolutionary runs included multiple adversaries, resulting in better generalization, as evidenced by our league results (where our players encountered previously unseen opponents). Nevertheless, there is still room for improvement where generalization is concerned. A simple (yet highly effective, in our experience) enhancement booster would be the increase of computational resources, allowing more adversaries to enter into the fitness function.

One of the evolutionary methods that was evaluated and abandoned was coevolution, wherein evolving bots battled each other rather than external opponents. Coevolution has a prima facie better chance of attaining superior generalization, due to the diversity of opponents encountered during evolution (indeed, it was highly effective with the previous games we encountered). However, we found that the evolved players presented primitive behavior, and were easily defeated by human-written programs. Eisenstein [53] described the same phenomenon, suggesting that the problem lay with the initial generation: The best strategies that appeared early on in evolution involved idleness—i.e., no moving or firing—since these two actions were more likely to cause loss of energy. Breeding such players usually resulted in losing the genes responsible for movement and firing, hence the poor performance of the later generations. We believe that coevolution can be fruitful if carefully planned, using a two-phase evolutionary process. During the first stage, the initial population would be evolved using one or more human-written adversaries as a fitness measure; this phase would last a relatively short period of time, until basic behavioral patterns emerged. The second stage would involve coevolution over the population of individuals that had evolved in the first stage.

Chapter 9

Robot Auto Racing Simulator

As we saw in the previous chapter, with programming games the objective is not to *be* the optimal player, but to *write* the best playing program. Usually, these programs are hand-coded by human programmers; however, in some cases, machine-learning techniques are applied to the creation or optimization of the controllers.

In this chapter we tackle the simulation game of RARS (Robot Auto Racing Simulator), which is an open-source, car-race simulator. This game was chosen mainly because of its extensive human-written driver library, and the substantive amount of published works that describe machine-learning approaches applied to RARS—enabling us to perform significant comparisons between our results and both human- and machine-designed solutions (Shichel and Sipper [161]).

This task is considered a hard problem because race-car control requires a high level of expertise in various game aspects, such as speeding, steering, and race-line planning, and, moreover, the controller should ultimately outperform existing solutions, created both by humans and various AI approaches.

9.1 Previous Work

Controlling a moving vehicle is considered a complex problem, both in simulated and real-world environments. Dealing with physical forces, varying road conditions, unexpected opponent behavior, damage control, and many other factors, renders the car-racing problem a fertile ground for AI research. Below we survey several works on evolving controllers for cars in simulated environments.

Wloch and Bentley [184] used genetic algorithms to optimize setup parameters, such as tire air pressure, gear change rates, and spring tension, in the "Formula One Challenge '99-'02" simulator. Modifying the car setup rather than its controlling software, they were able to show improvements in the overall performance.

Floreano et al. [65] used coevolution of artificial neural networks (ANNs) to develop both image feature selection (filtering an image in order to extract various features) and active vision (selecting parts of the image to focus on) to create a controller for the "CarWorld" open-source simulator.[1] They developed a controller able to complete a lap on several given tracks, relying only on visual inputs, as seen from the driver's seat.

Togelius and Lucas [181] employed various approaches based on GAs and ANNs in order to train a simulated radio-controlled car to drive on simple tracks, in which the controllers possessed complete knowledge of the track structure via a simulated overhead camera. In another, rather unorthodox work, Togelius et al. [182] used a GA to evolve the tracks rather than the controllers, and tried to maximize the "fun factor" for the game players, by suggesting tracks that were challenging yet not overly hard so as to cause frustration. (In a similar unorthodox vein, Sipper [164] evolved environments to fit a robot.)

Chaperot [33] and Chaperot and Fyfe [34] used GAs and ANNs to create motorcycle controllers for the "Motocross—The Force" simulator, which features competitive bike driving across a three-dimensional terrain, including complex rigid-body physics.

Tanev et al. [174] used a GA to optimize the parameters of a real-world, radio-controlled car controller. They demonstrated an increase in performance during the course of evolution, and the emergence of obstacle-avoiding behavior once obstacles were introduced onto the track.

RARS, the Robot Auto Racing Simulator,[2] attracted numerous academic researchers and hobbyists, and was one of the first platforms to enable an objective comparison between the performance of controller algorithms, by holding open, online racing competitions on a regular basis. In addition to many controllers hand-coded by hobbyist programmers, various AI techniques were used to create, train, and optimize RARS controllers.

Several researchers used ANNs within the RARS framework. Coulom [43] applied temporal difference reinforcement learning to train an ANN to drive a car around a track, while Pyeatt and Howe [143] trained multiple ANNs to perform low-level tasks—such as driving and overtaking—and a higher-level mechanism to switch between the low-level behaviors.

Sáez et al. [152] and Eleveld [55] used a GA to find an optimal path around a RARS track, a highly effective method for known tracks without stochastic effects, but one that leads to very poor performance on unknown tracks or in nondeterministic situations.

Stanley et al. [171] presented a combined ANN and GA approach, using Neuro-

[1] `http://carworld.sourceforge.net`
[2] `http://rars.sourceforge.net`

Evolution of Augmenting Topologies (NEAT) to evolve and train a RARS-based collision warning system. This approach combined a conventional ANN training algorithm for the network weights with an evolutionary algorithm that modified their topologies. Although their main focus was on the creation of a collision warning system rather than the development of a fast driver, their evolved controllers were able to complete a lap in good time.

Rule-based solutions, created using reinforcement learning, were suggested by Cleland [40] and led to the creation of rather competitive RARS controllers. Ng et al. [128] trained RARS controllers to imitate the behavior of a "good" human-crafted controller, using the Modular Neuro-Fuzzy (MoNiF) approach—a combination of fuzzy classifying functions that were used to create discrete input values for artificial neural networks.

TORCS (The Open Race Car Simulator[3]), which is based on RARS, has been gaining popularity over the past couple of years, and several TORCS-related papers have been published. Notable works include the application of fuzzy classification functions to the creation of competitive controllers (Onieva et al. [130]; Perez et al. [138]), parameter optimization of a hand-coded controller using an evolutionary strategy (Butz and Lönneker [27]), and the imitation of successful machine- or human-crafted controllers by using either ANNs (Muñoz et al. [124]) or NEAT and k-nearest neighbor classifiers (Cardamone et al. [31]).

Ebner and Tiede [51] showed that GP can be successfully applied to evolving TORCS-playing agents; however, their evolved controllers were not able to compete successfully with manually constructed drivers, and their generalization capabilities were not tested.

Finally, Cardamone et al. [30] used real-time Neuro-Evolution of Augmenting Topologies (rtNEAT) to evolve a TORCS controller from scratch and optimize its performance on unseen tracks during the course of a single game, unlike the usual use of learning techniques, which are applied prior to the race itself.

Some of the above RARS-based works provide the exact lap times of the generated controllers. In Section 9.4, we will inspect these results and compare them with our own.

9.2 The RARS Simulator

RARS is an open-source, car-race simulator, written in C++. It was created by several individual programmers in 1995, and has evolved since then into a complex racing system. This game employs a detailed physical engine, including most of the forces relevant to moving cars, such as acceleration and deceleration, frictional factors, and

[3]`http://torcs.sourceforge.net`

centripetal forces. This game enjoyed a large and lively gamer community, and RARS tournaments were held regularly between 1995 and 2004.

The goal of the game is fairly simple: one or more cars race on a given track. The cars are positioned at the starting line and simultaneously start moving when the race begins. Cars are damaged upon collision or when driving off the track. When a car reaches the starting line, which also acts as the finishing line, a lap counter is incremented. The winner is the driver whose car finished a given number of laps first.

A RARS controller is a C++ class with a single method, which receives the current race *situation* and determines the desired speed and wheel angle of the car. The simulation engine queries the controller approximately 20 times per “game second”, and advances the car according to the returned decisions and physical constraints. The *situation* argument provides the agent (car controller) with detailed information about the current race conditions, such as current speed and direction, road curvature, fuel status, and nearby car positions.

Controlling the car is done by two actuators: speed and steering. The speed actuator specifies the desired speed of the car, while the steering actuator specifies the desired wheel angle. The simulation engine uses both values to calculate the involved physical forces and compute the car’s movement. Extreme values, such as high speed or a steep steering angle, may result in slippage or skidding, and must be taken into consideration when crafting a controller.

RARS controllers should be able to perform well on a variety of tracks and scenarios. The basic RARS package contains several simple tracks of various shapes, such as oval, round-rectangular, and figure 8-shaped. In addition, each of the RARS tournaments contains several tracks of higher complexity, which are not included in the basic package. Some of these are replicas of real-world tracks (such as the Sepang International Circuit[4]), while others are fictional tracks that were designed by the tournament administrator.

RARS tournament rules divide the game-playing controllers into two classes, differing in a single aspect: precomputation. Agents of the first class—planning agents—are allowed to inspect the track prior to the race, and apply a computational process to the track data. This is usually used to produce a precise driving plan—a series of radii and speeds—according to which the car should drive. The second class of agents—reactive agents—are not given the track plan, and their actions rely only on the road conditions observable by the driver in accordance with the car’s physical position at any given time.

Since pure planning agents do not take stochastic factors (such as nearby cars or random friction factors) into consideration, they are rendered useless in many situations; therefore, most of this class’s agents employ some degree of reactive behavior

[4]`http://malaysiangp.com.my`

in addition to the pre-calculated driving plan. By and large, planning agents outperform reactive agents because they are better prepared to handle the road conditions, and their precise knowledge regarding the road curvature for any track segment allows them to be prepared for unexpected road features.

Both problems—reactive driving and optimal path planning—are of interest to the AI community. This chapter focuses on reactive agents.

9.3 Evolving a Race-Car Controller

We chose to focus on the task of creating purely reactive agents for single-car, single-lap races. In this game variant, each race includes one car, attempting to achieve the best lap time (Shichel and Sipper [161]).

Each agent was controlled by two LISP expressions—one for the speed actuator and the other for the steering actuator. Whenever the controller was queried by the RARS simulation engine, both expressions were evaluated and their results were passed back as the desired speed and steering values.

9.3.1 Functions and terminals

The LISP expressions were defined over the set of functions and terminals described in Table 9.1 and Figure 9.1. They were divided into several groups:

- *Basic game status indicators*, which return real-time information regarding the status of the car, as provided by the RARS simulation engine.
- *Complex game status indicators*, which also return real-time information. This indicator set expands the basic set with indicators that are not provided directly by the game engine, but instead are calculated by our software. These indicators, such as the distance to the next obstacle, require complex trigonometric functions and code loops, which are beyond the complexity capabilities of the GP code model we used, and hence are impossible to develop by means of evolution. These building blocks are actually human-made functions, driven by intuition, and can be very powerful when introduced into the evolutionary process.
- *Numerical constants*, which include the constants `0`, `1`, and `ERC`.
- *Mathematical functions*.
- *Conditional statements*.

Table 9.1 Functions and terminals used to evolve race cars.

Basic game status indicators	
`CR`	**C**urrent **R**adius: Radius of current track segment
`NR`	**N**ext **R**adius: Radius of next track segment
`TE`	**T**o **E**nd: Distance to end of current track segment
`NL`	**N**ext **L**ength: Length of next track segment
`V`	**V**elocity: Current velocity of car
`NV`	**N**ormal **V**elocity: Drift speed towards road shoulder
`TL`	**T**o **L**eft: Distance to left road shoulder
`TR`	**T**o **R**ight: Distance to right road shoulder
`TW`	**T**rack **W**idth
Complex game status indicators	
`AH`	**AH**ead: Distance car can move in its current heading without veering off road
`AA`	**A**head **A**ngle: Angle of road shoulder, relative to car's heading, found by AH terminal
Numerical constants	
`ERC`	Ephemeral Random Constant
`0`	Zero constant
`1`	One constant
Mathematical functions	
`+`(x,y)	Adds x and y
`-`(x,y)	Subtracts y from x
`*`(x,y)	Multiplies x by y
`%`(x,y)	Safe-divide x by y: if $y = 0$, returns 0, otherwise returns the division of x by y
`abs`(x)	Absolute value of x
`neg`(x)	Negative value of x
`tan`(x)	Tangent of x
Conditional statements	
`IFG`(x,y,α,β)	If $x > y$, returns α, otherwise returns β
`IFP`(x,α,β)	If x is positive, returns α, otherwise returns β

Notes:

- Game indicators, both basic and complex, were normalized to fit a common scale.
- Distance values `TE`, `NL`, `TL`, `TR`, `TW`, `AH`, `CR`, and `NR` are in feet, divided by 400.
- Velocity values `V` and `NV` are in feet per second, divided by 100.
- The angle indicator `AA` is specified in radians.
- Radii values specify both the radius and the direction of the track segment: positive values indicate a counter-clockwise turn, negative values indicate a clockwise turn, and a value of zero represents a straight track segment.

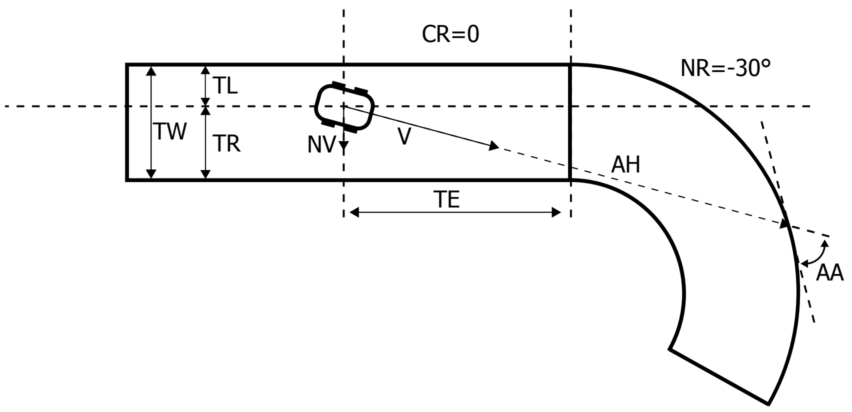


Figure 9.1 RARS game indicators (basic and complex).

9.3.2 Fitness function

Two aspects were taken into consideration when defining fitness: track selection and fitness-value calculation.

The track on which the individuals are evaluated should be as diverse as possible. A homogeneous track (an oval one, for example) might yield specialized agents, which perform well on the given track but show poor performance on other tracks. A heterogeneous track, which contains many distinct features, is likely to yield more generalized drivers, able to drive well on any given track.

We inspected the RARS track library and chose the sepang track, which exhibits many common track features, such as sharp and moderate curves, U-turns, and straight segments of varying lengths (Figure 9.4(f)).

Two related fitness functions were used in order to measure the quality of a driver: *Race Distance* and *Modified Race Time*:

- *Race Distance* is the distance, in (simulated) feet, that the car traverses during a 250-game-second period. When this function was used during evolution, the goal was to maximize the fitness value of the individuals.

- *Modified Race Time* is the time, in game seconds, required by the car to complete the race. Because some agents fail to complete a single lap (due to extremely slow driving or a fatal crash—a phenomenon not uncommon in early generations), we amended this simple measure. The modified measure was a comparison-based fitness measure, which did not produce a quantitative fitness value, but instead compared two (or more) individuals and determined the fitter of the lot.

  Such a measure can be used only with comparison-based selection methods, such as tournament selection. Specifically, when comparing two controllers that finished a single lap, the one with the shortest lap time was considered to be fitter. If one of the controllers was not able to complete the lap, it was considered less fit than the one that did finish. If both controllers were not able to finish the lap, the one that traveled the farthest was considered to be the fittest.

Using Modified Race Time we were able to directly address the challenge at hand—evolving controllers with the shortest lap time—while maintaining a diverse population in early generations, wherein no controller was able to complete a single lap.

9.3.3 Run parameters

The evolutionary parameters were carefully chosen through a long calibration process. In this process, various evolutionary runs were executed in an attempt to measure the influence of each evolutionary parameter. The final set of parameters was as follows:

- *Population size*: 250 individuals. Using larger populations did not yield significantly better results, but smaller populations were not able to produce good results.

- *Generation limit*: A value of 255 generations was used. Usually, the population reached an observed peak performance between generations 150 and 200, so best-of-run individuals often emerged before the limit was reached.

- *Selection method*: Tournament of 3 individuals. In this method, the selection of a single individual is done by randomly choosing three individuals, and returning the fittest among them.

  Different tournament group sizes were tested during the calibration process: groups larger than 4 individuals yielded faster convergence to non-optimal solutions, while groups of 2 individuals resulted in slow convergence to non-optimal solutions.

- *Breeding operators*:

  – Reproduction (40%): Selects one individual, using the selection method described above, and passes it on to the next generation as is. Other reproduction probabilities, including no reproduction at all, were tested during the calibration phase. We found that a lower reproduction rate resulted in faster convergence, but not necessarily to optimal solutions. We surmise that a high reproduction rate allowed enough good individuals to move unmodified into the next generation, thus affording the preservation of their properties without incurring the risk of damaging them by mutation or crossover.

- Crossover (50%): Selects two individuals, using the selection method described above, and creates two new individuals by substituting random subtrees between them. Bloat (Langdon [112]) was controlled by setting a tree depth limit of 8 and choosing subtrees such that the resulting trees did not exceed this limit.
- Structural mutation (5%): Randomly selects one individual and creates a new one by choosing a random tree node, discarding its rooted subtree, and growing a new subtree instead. Bloat control was achieved through the same mechanism that was used in crossover. Structural mutation was used in order to introduce variants of existing individuals; however, due to its destructive potential, it was used in small doses.
- `ERC` mutation (5%): Randomly selects one individual and modifies its `ERC`s. This is done by randomly choosing an `ERC` node within the individual and modifying its numerical value. This operator was used to fine-tune the constant values that were used as evolutionary building blocks.

• *Creation of initial population* was done according to Koza's ramped-half-and-half method (Koza [108]).

9.4 Results and Analysis of an Evolved Driver

We executed ten evolutionary runs with the Race Distance fitness function, and ten runs with the Modified Race Time fitness function. An individual's fitness was calculated on the sepang track (Figure 9.4(f)). The progress of the two best evolutionary runs is shown in Figures 9.2 and 9.3. We extracted one individual from each of these runs: GP-Single-1 (evolved using Race Distance fitness) and GP-Single-2 (evolved using Modified Race Time), both found by performing ten independent races per each individual in the last generation, and choosing the individual with the best average lap time.

Figure 9.4 shows the performance of the GP-Single-2 driver on several tracks from the RARS library, clearly exhibiting advanced driving features. The car slows down before curves in proportion to their sharpness, to eliminate the risk of losing control; moreover, the controller attempts to increase the path radius by entering and exiting the curve from the outer shoulders and touching the inner shoulder at mid-curve, thus enabling the car to travel at higher speeds without the risk of skidding.

A comparison with human-crafted reactive drivers on the sepang track is shown in Table 9.2. Note that lap times vary from race to race due to random factors in the friction coefficient formula aimed at simulating real-world conditions, such as dirt and debris on the race track. Therefore, each race was performed 100 times per driver, and the results were used to calculate the average timings, as well as standard

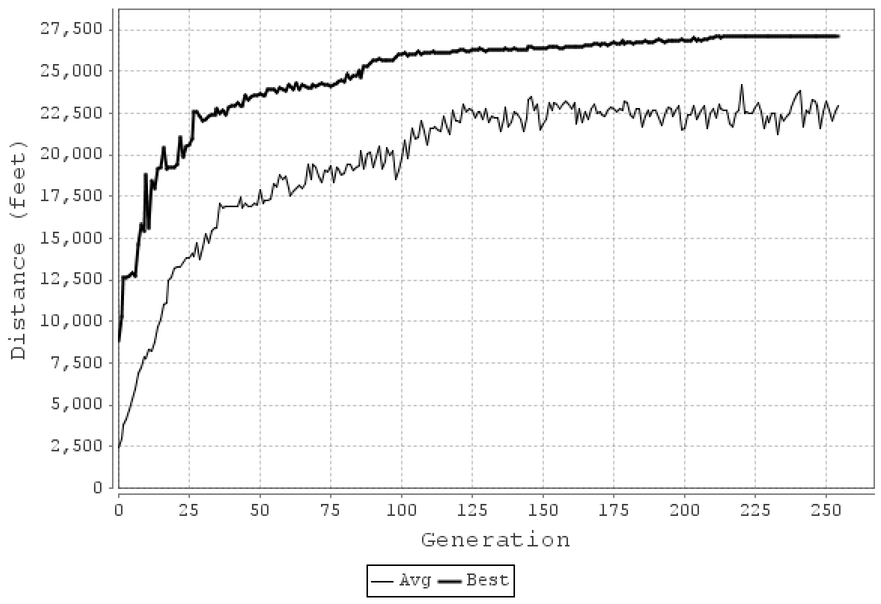


Figure 9.2 Fitness vs. Time plot of the best evolutionary run using the Race Distance fitness measure. The thick line denotes the best fitness of each generation, while the thin line denotes the average fitness of the population in each generation.

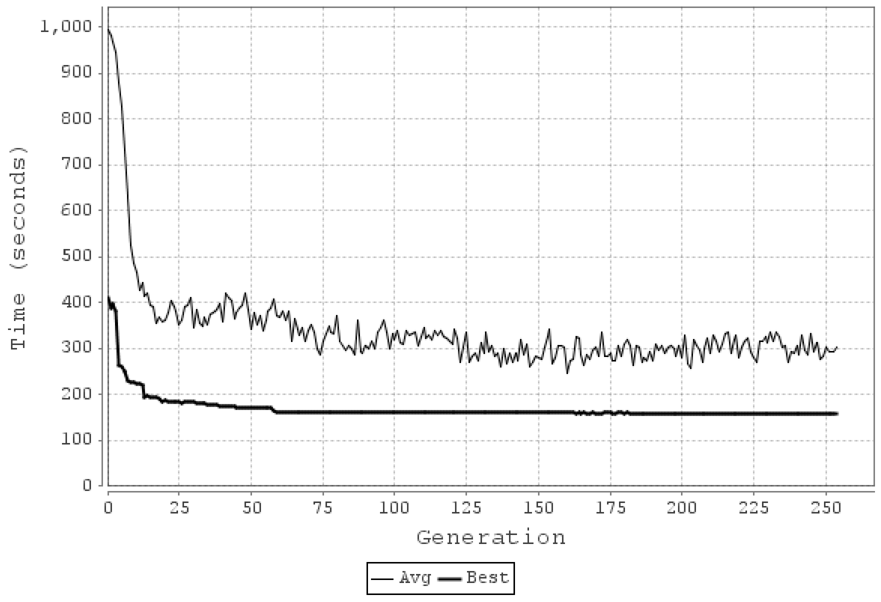
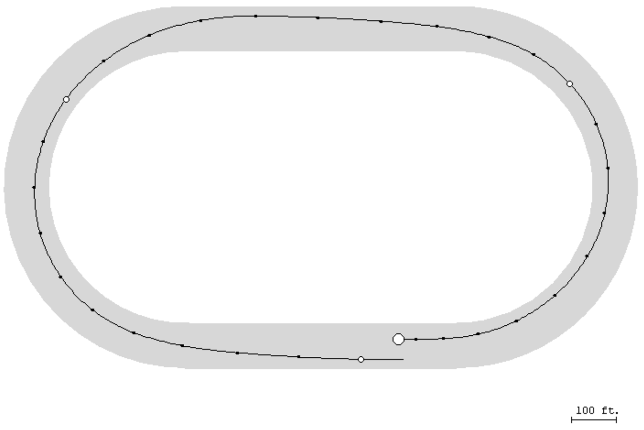
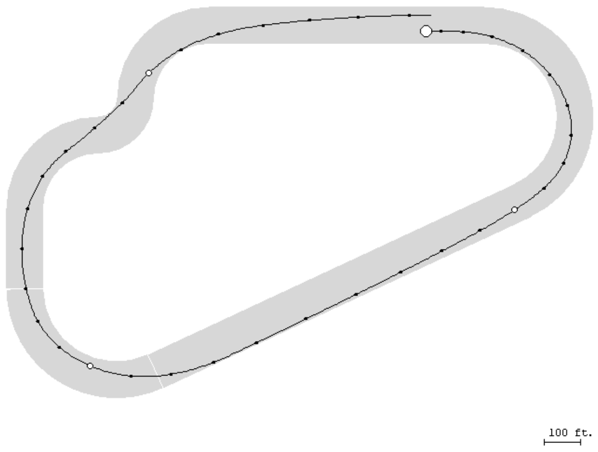


Figure 9.3 Fitness vs. Time plot of the best evolutionary run using the Modified Race Time fitness measure. Since this fitness measure does not produce a numerical value (but uses a comparative model instead), it cannot be plotted straightforwardly. Hence, to properly plot this run we used the following method: drivers that were able to complete a single lap were plotted using their lap times, while drivers that were not able to complete a single lap were assigned an arbitrary lap-time value of 1000 seconds.

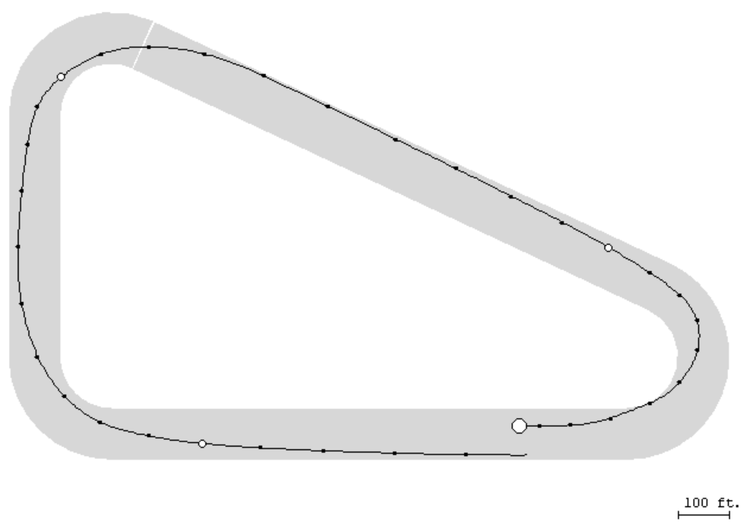


(a) oval (30.8 seconds)

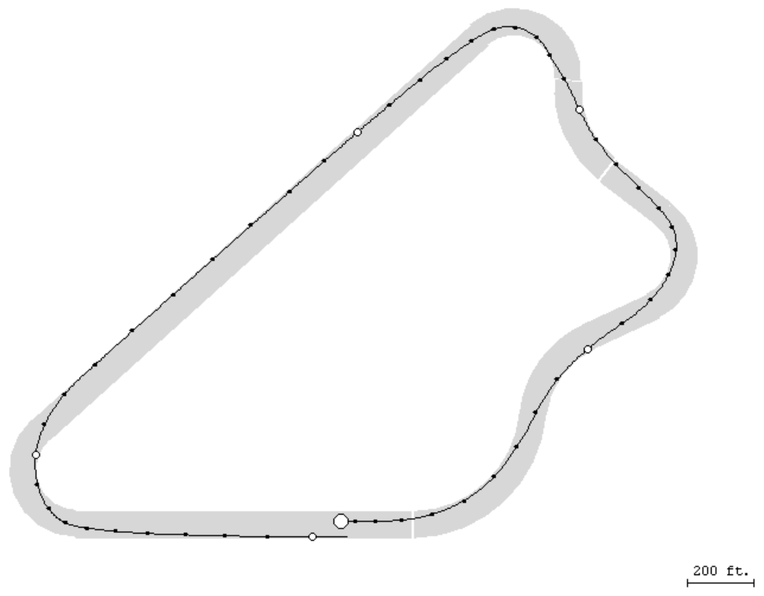


(b) clkwis (36.5 seconds)

Figure 9.4 Performance of GP-Single-2 on six tracks. Black dots represent one-second intervals, while white dots represent ten-second intervals. The large white dot is the starting point, from which the car starts moving. Some advanced driving techniques can be observed from these figures by examining the path line and the density of the time marker dots—which implicitly indicate the car's speed at any given time. The car slows down when approaching sharp curves, thus reducing the risk of skidding (tracks (c), (d), and (f)). Tracks (b) and (d) exhibit a special slalom behavior, where the controller doesn't follow the curvature of the road, but drives straight through the slalom instead. Finally, tracks (b), (c), (d), and (f) depict the controller's attempt to maximize the path radius by touching the inner shoulder at mid-curve, thus allowing the car to travel faster within the curve.

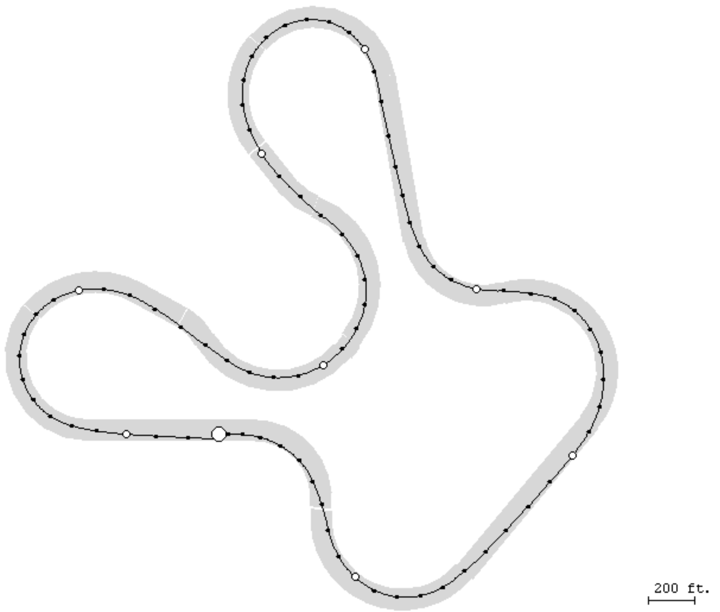


(c) v01 (35.0 seconds)

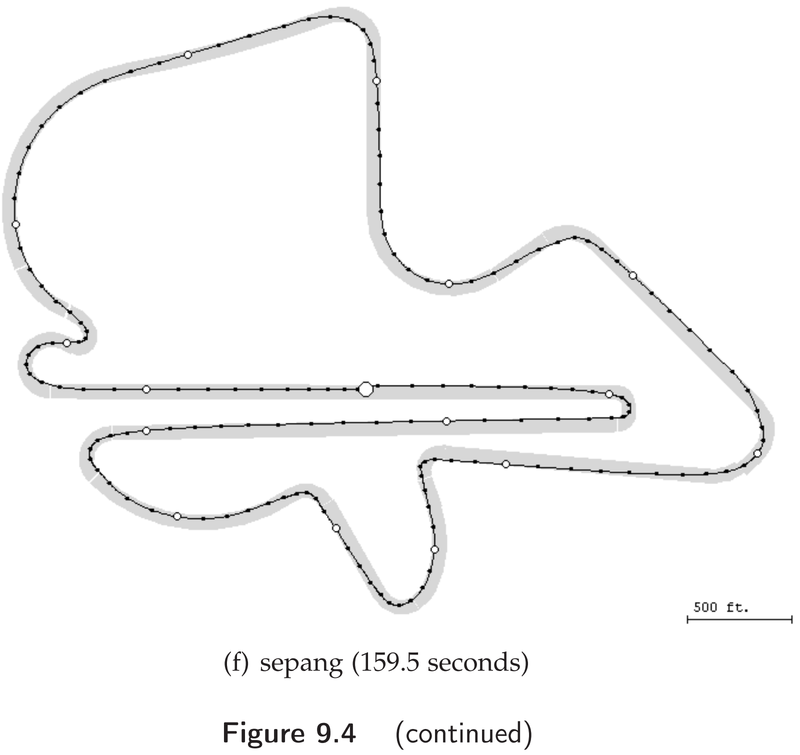


(d) aug04/race13 (50.8 seconds)

Figure 9.4 (continued)



(e) aug04/race5 (83.1 seconds)



(f) sepang (159.5 seconds)

Figure 9.4 (continued)

Table 9.2 Comparison of evolved drivers on the sepang track. The entire set of reactive human-crafted controllers from the latest RARS distribution was tested. Results vary slightly from race to race due to the simulation's stochastic nature; therefore, each race was performed 100 times per driver, and the results were used to calculate the average timings, as well as standard deviation values and standard error values (in parentheses).

Rank	Driver	Lap Time (seconds)
1	GP-Single-2	159.8 ± 0.6 (std. error: 0.06)
2	Vector	160.9 ± 0.1 (0.01)
-	GP-Single-1	160.9 ± 0.3 (0.03)
4	WappuCar	161.7 ± 0.1 (0.01)
5	Apex8	162.5 ± 0.2 (0.02)
6	Djoefe	163.7 ± 0.1 (0.01)
7	Ali2	163.9 ± 0.1 (0.01)
8	Mafanja	164.3 ± 0.2 (0.02)
9	SBv1r4	165.6 ± 0.1 (0.01)
10	Burns	167.8 ± 5.6 (0.56)
11	Eagle	169.3 ± 0.6 (0.06)
12	Bulle	169.4 ± 0.3 (0.03)
13	Magic	173.9 ± 0.1 (0.01)
14	JR001	178.3 ± 0.2 (0.02)

deviation and standard error values. Our top evolved drivers were able to rank first and second out of 14 contestants.

Both evolved drivers exhibit shorter lap times than any human-crafted driver in their class (excluding Vector, which shares the second-best result with GP-Single-1). However, since many machine-learning techniques tend to prefer specialization over generalization, the performance of our evolved drivers should be checked on tracks other than sepang—which was used for fitness calculation in the evolutionary process. In order to perform such a comparison we evaluated each human-crafted driver along with our own evolved drivers on 16 tracks, taken from the August 2004 RARS tournament. This tournament is the most recent one for which the source code of human-crafted drivers is available online, thus allowing us to compare the results between our drivers and the human-crafted ones. The results are shown in Table 9.3.

Out of 14 drivers (all but ours designed by humans), the evolved drivers ranked second and third. These results show that the evolved solutions exhibit a high degree of generalization, and are able to successfully solve instances of the problem that were not included in their original training set.

To further inspect the evolved drivers and their generalization capabilities, we tested their performance on the August 2004 season with two scenarios that were not targeted in the training phase: multiple-lap races and multiple-car races. These scenarios require different behaviors than single-lap races, as well as several indicators that were not available to our evolved drivers, such as damage levels and information

Table 9.3 Comparison of evolved drivers with human-crafted drivers on 16 tracks from the August 2004 season, based on 10 races per controller and using the IndyCar points system, wherein the twelve fastest drivers receive 20 points (best driver), 16, 14, 12, 10, 8, 6, 5, 4, 3, 2, and 1 point (worst driver), respectively; in addition, the driver that leads the most laps receives an additional bonus point, and the winner of the qualifications round—if one is held—receives an additional point. We held no qualification round and the race consisted of a single lap, hence the fastest driver received 21 points. The total score of each driver is simply the sum of its single race scores. Each driver's rank per race is listed along with its total seasonal score (rightmost "Total" column).

Rank	Driver	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	Total
1	Vector	1	1	6	5	13	1	8	1	1	1	3	9	3	2	2	2	229
2	GP-Single-2	3	4	10	7	1	3	3	3	2	2	1	5	1	5	11	1	215
3	GP-Single-1	4	3	12	1	5	2	1	9	4	7	4	1	13	4	3	5	186
4	Mafanja	2	5	8	3	2	7	4	7	5	4	6	4	4	3	4	4	177
5	SBv1r4	9	6	11	6	6	4	5	2	3	5	5	2	2	8	9	6	151
6	Eagle	10	2	1	13	11	13	6	8	7	3	2	8	12	1	1	8	144
7	WappuCar	8	7	9	4	8	5	2	5	9	6	7	3	7	11	6	10	119
8	Djoefe	6	10	3	9	4	9	9	4	10	9	9	10	5	6	5	3	117
9	Burns	5	8	7	8	3	8	7	6	6	8	10	6	6	7	7	7	109
10	Magic	11	9	2	2	10	6	10	12	8	11	11	7	10	10	8	11	81
11	Ali2	7	11	4	10	7	11	11	11	11	10	8	11	8	9	10	9	63
12	Apex8	12	12	5	11	9	10	13	10	12	12	12	12	9	12	12	12	35
13	JR001	13	13	13	12	12	12	12	13	13	13	13	13	11	13	13	13	6
14	Bulle	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	0

about nearby cars.

The performance on multiple-lap scenarios (30-50 laps) was rather poor. Our controllers were unable to complete a single race because they reached a critical damage level and broke down after a few laps. It appears that the evolved drivers gain a small amount of damage per lap—a harmless phenomenon in single-lap races—but after several laps the accumulated damage level reaches a critical level and prevents the drivers from finishing the race. This problem will probably be alleviated by evolving multiple-lap drivers.

Multiple-car scenarios, however, proved surprisingly good, as seen in Table 9.4. Our controllers reached the first and fourth places, scoring better than Vector—the winner of the single-lap challenge, and Mafanja—the winner of the original August 2004 season. Considering the fact that the evolved controllers do not have information regarding their surrounding cars, we conclude that multiple-car behaviors—such as overtaking and collision avoidance—are of less importance in this game, compared to the task of driving as fast as possible. If damage control is not a consideration, a brutal drive-through strategy is apparently sufficient for our controllers to gain the leading position, and, once gained, the expected behavior is similar to the single-car scenario behavior. It appears that Vector is less successful in multi-car scenarios (as seen in Table 9.4), and Mafanja is less successful in the fast-driving challenge (as seen in Table 9.3), hence GP-Single-2 was able to rank first.

Table 9.4 Comparison of evolved drivers with human-crafted drivers on 16 tracks from the Aug. 2004 season, based on 10 races per controller and using the IndyCar points system, on a 3-lap, multiple-car scenario. Each driver's average score per race and total seasonal score is listed. The rightmost "Orig." column shows the original score of the Aug. 2004 season.

Rank	Driver	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	Total	Orig.
1	GP-Single-2	11	15.1	2.5	13.4	20.3	17.8	19.5	16.3	16.6	16.7	20.9	17.3	16.9	11.1	6.6	20.4	242.4	-
2	Mafanja	14.9	11.5	13	13	13.2	10.8	12	11.9	10.6	15.6	8.6	13.1	11.5	16.9	12.9	10.3	199.8	236
3	Vector	14.9	16.3	8.3	2.9	0	8.1	9.5	14.6	17.3	14.7	14.2	2.6	13.1	13.2	14.8	6.4	170.9	190
4	GP-Single-1	16.8	12	2	15.6	4.1	14.3	17.1	7.8	10.4	11.2	11	11.1	1.6	11.7	12.4	9.1	168.2	-
5	Djoefe	10.4	5.4	20.1	7.4	13	6.1	6.7	14.6	7.9	6.4	7.5	5.9	13.3	8.5	14.4	15.2	162.8	139
6	Burns	8.5	8.3	8.7	10	12.4	7.9	6.6	7.6	8.5	7.6	6.4	7.6	10.7	8.6	9.8	11.4	140.6	144
7	WappuCar	7.3	4.7	5.9	9	7.6	9.6	10.8	6.7	5.4	6.4	9	11	8.2	5.3	6.6	4.5	118	160
8	Bulle	3.4	9.7	4.8	11.6	3.1	4.7	9.1	4.4	6	8.9	3.2	14.6	5.8	4.4	2.6	3.8	100.1	137
9	Ali2	7.5	3.3	12.6	3.7	8.4	4.9	3	4.9	4.3	3.1	8.8	2.8	7.3	5.4	8.9	6.7	95.6	100
10	Magic	3.4	5.5	7.2	10.4	4.2	11.4	3	1.3	4.1	2.7	3.1	5.7	4.9	7.7	5.3	6.6	86.5	60
11	SBv1r4-1	1.6	4.1	1.1	2.5	3.5	2.7	2.9	7.7	1.8	3.9	4	7.1	3.8	1.8	2.2	4.3	55	85
12	Eagle	0.2	5	6.2	0.2	9.1	0	0.4	1.1	7.5	4.4	4.8	0.3	0.7	5.8	1.5	1.7	48.9	51
13	Apex8	2.3	0.8	9.6	1.3	3	3.8	0.9	2.9	1.4	0.4	0.4	2.2	3.8	1.1	3.9	1.5	39.3	67
14	JR001	0	0.3	0	1	0.1	0.3	0.5	0.4	0.2	0	0.1	0.9	0.4	0.5	0.1	0.1	4.9	28

Table 9.5 Comparison of evolved drivers with machine-generated drivers (best results in boldface).

		Lap Time (seconds)		
Author	Track	Reported	GP-Single-1	GP-Single-2
Ng et al.	v03	59.4	55.5 ± 1.4 (std. error: 0.14)	**49.3** ± 0.1 (0.01)
	oval	33.0	31.0 ± 0.1 (0.01)	**30.7** ± 0.1 (0.01)
	complex	209.0	**199.4** ± 5.9 (0.59)	204.4 ± 1.3 (0.13)
Coulom	clkwis	38.0	37.7 ± 0.1 (0.01)	**36.5** ± 0.1 (0.01)
Cleland	v01	37.4	37.9 ± 1.6 (0.16)	**35.0** ± 0.1 (0.01)

Comparison with machine-generated solutions discussed in Section 9.1 was done by recording the performance of our evolved drivers on each track for which machine-generated results were reported. Table 9.5 lists our findings. Again, due to the stochastic nature of the simulation, each race was performed 100 times per driver and average results were noted along with standard deviation and standard error values. However, since we had only the reported results for the machine-generated drivers—rather than an executable version—no statistical information was available for them.

The evolved drivers perform better than any machine-generated reactive drivers. Furthermore, the tracks used for these comparisons were *not* included in the training set of the evolved drivers, but *were* used to train most of the machine-generated solutions.

The human-crafted controllers described in Tables 9.2 and 9.3 were built for multiple-lap, multiple-car scenarios, in which additional behavioral patterns—such as overtaking slow opponents, damage control, and fuel consumption monitoring—might be required. However, we assume that most human-crafted controllers would attempt to drive as fast as possible when no opponents are nearby, which is the case in the single-car scenario. Each of the machine-generated controllers was designed

Figure 9.5 GP-Single-2: Expression for wheel angle, α.

```
(% (% (% (% (IFG 0.702 AH AA (* NV -0.985)) (- AH (neg AH))) (-
(% 1.0 (% V AH)) (neg AH))) (- (- (* NV (neg NV)) (neg AH)) (neg
AH))) (- (% 1.0 (% V AH)) (neg (% (% 1.0 (% V AH)) (% V AH)))))
```

for the single-car scenario, differing only in the race-length parameter: Ng et al.'s controllers were trained on 60-lap races, Coulom's controllers were trained on a single-lap scenario, and Cleland's incorporated very long training phases during a single race, usually featuring hundreds of laps. All three controllers, however, aimed at reducing the average time of a single lap.

As over-specialization is a common phenomenon in many machine-learning approaches, the emergence of generalized solutions is nontrivial. We surmised that our choice of a complex track for fitness evaluation, combined with a sound yet simple set of genetic building blocks, contributed greatly to the generalization capabilities.

To further explore this hypothesis, we executed several evolutionary runs using track v01, which is a fairly simple one (compare Figure 9.4(c), depicting v01, with Figure 9.4(f), depicting sepang—which we used during evolution). The individuals evolved in these runs were highly competitive when driving on their training track (v01): the best-of-run was able to complete a lap in 34.1 ($\pm$ 0.6) seconds (averaged over 100 runs), a result that is 3.8 seconds better than GP-Single-1 and 0.9 seconds better than GP-Single-2 on average. However, on unseen tracks this controller's performance was rather poor: on v03 it completed a lap in 90.0 ($\pm$ 19.2) seconds on average, on clkwis it completed a lap in 71.7 ($\pm$ 17.9) seconds on average, and on sepang it failed altogether, having reached a critical damage level before completing a single lap.

The large error margins also suggest that the controller's behavior was inconsistent on unseen tracks; it was probably intolerant to subtle random factors on such tracks, since its behavior was specialized to the simple track on which it was trained. Hence, we concluded that our choice of a *complex* track contributed greatly to the generalized nature of the evolved controllers.

In an attempt to further understand the evolved controllers we wore the "molecular biologist" hat once again and analyzed their code. As explained in Section 9.3, each driver comprised two LISP expressions: one provided the desired wheel angle α, while the other provided the desired speed v. Both expressions for GP-Single-2 are shown, respectively, in Figures 9.5 and 9.6. Although seemingly complex, these expressions can be simplified manually:

$$\alpha \;=\; \Psi \cdot \left(\frac{1}{2AH} \cdot \frac{1}{\frac{AH}{V}+AH} \cdot \frac{1}{2AH-NV^2} \cdot \frac{1}{\frac{AH}{V}-(\frac{AH}{V})^2}\right), \tag{9.1}$$

Figure 9.6 GP-Single-2: Expression for speed, v.

```
(IFP (abs (% V AH)) (- (% 1.0 (% V AH)) (neg (- (* NV (* NV
-0.868)) (neg AH)))) (% (neg (- (- (* NV (neg TR)) (neg AH))
(neg AH))) ( - (% 1.0 (% V AH)) (neg (% (* NV (neg NV)) (% V
AH))))))
```

where:

$$\Psi = \begin{cases} AA & AH < 0.7 \\ -0.98 \cdot NV & AH \geq 0.7, \end{cases}$$

and

$$v = |AH \cdot (\frac{1}{V} - 1) + 0.87 \cdot NV^2|. \qquad (9.2)$$

These equations intimate at the logic behind the evolved controller's decisions. The Ψ element in Equation 9.1 shows that the steering behavior depends on the distance, AH, to the upcoming curve: when the next turn is far enough, the controller slightly adjusts the wheel angle to prevent drifting off track; when approaching a curve, however, the controller steers according to the relative curve angle—steep curves will result in extreme wheel angle values.

The AH indicator is used in Equation 9.2 too, and we observe that the desired speed is also determined by the distance to the next curve, among other factors.

The expression AH/V is used quite frequently: one instance is seen in the speed equation and three instances in the steering equation. Given that AH is the distance to the upcoming road shoulder, and V is the current velocity, this expression is simply the *time to crash* indicator: when the car will veer off-road if it keeps its current speed and heading. As this indicator is undoubtedly important for a race-car controller, and *wasn't provided as a genetic building block*, evolution found a way of expressing it—and used it extensively.

9.5 Discussion

We used GP to evolve RARS controllers, finding that the evolutionary approach yields high-performance controllers, able to compete successfully both with human-crafted and machine-generated controllers.

The evolved drivers demonstrated a high degree of generalization, enabling them to perform well on most tracks—including ones that were not used during the evolutionary process. We noted that using a complex track for fitness evaluation, coupled with a comprehensive yet simple set of genetic building blocks, contributed greatly to our controllers' generalization capabilities. We also observed the emergence of useful code patterns, such as the *time to crash* indicator. Such patterns were repeatedly used in the evolved individuals' code, acting as evolution-made genetic building blocks.

Having focused on single-car, single-lap races, we could expand our research to more complex scenarios, including multiple cars, multiple laps, damage control, and pit stops for repairs and refueling. This can be done during evolution with the guidance of an appropriate fitness function, and not just post-evolutionarily, as we did.

We could further extend our work by using a genetic algorithm to precompute an optimal path, to be combined with a GP-evolved controller in charge of following the path, for either single-car or multiple-car scenarios. Using GAs for path optimization has been done before (e.g., *DougE1* by Eleveld [55]) but not in combination with a machine-learning approach to the path-following behavior.

In addition, the RARS engine may be replaced with its successor, TORCS. This latter has, among others, the option of racing against human-*controlled* (as opposed to human-*crafted*) drivers, which is another interesting challenge.

Part IV

Puzzles

A good puzzle, it's a fair thing. Nobody is lying. It's very clear, and the problem depends just on you.

—Erno Rubik

Chapter 10

Rush Hour

Single-player games in the form of puzzles have received much attention from the AI community for some time (e.g., Hearn [84]; Robertson and Munro [147]). However, quite a few NP-complete puzzles have remained relatively neglected by researchers (see Kendall et al. [102] for a review).

Among these difficult games we find the Rush Hour puzzle,[1] which was proven to be PSPACE-complete (i.e., more difficult than NP-complete problems, if $NP \subset PSPACE$) for the general $n \times n$ case (Flake and Baum [64]). The commercial version of this popular single-player game is played on a 6x6 grid, simulating a parking lot replete with several cars (comprising two tiles) and trucks (comprising three tiles). The goal is to find a sequence of legal vehicular moves that ultimately clears the way for the red target car, allowing it to exit the lot through a tile that marks the exit (see Figure 10.1). Vehicles are restricted to moving either vertically or horizontally (but not both), they cannot vault over other vehicles, and no two vehicles may occupy the same tile at the same time. The generalized version of the game is defined on an arbitrary grid size, though the 6x6 board is sufficiently challenging for humans (we are not aware of humans playing, let alone solving, complex boards larger than 6x6).

A major problem-solving approach within the field of AI is that of heuristic search. One of the most important heuristic search algorithms is iterative deepening A* (IDA*) (Hart et al. [75]; Korf [105]), which has several well-known enhancements, including move ordering (Reinefeld and Marsland [145]) and pattern databases (Felner et al. [61]) (we will expand upon iterative deepening in Section 10.2). This method is widely used to solve single-player games (e.g., Junghanns and Schaeffer [96]; Korf [106]). IDA* and similar algorithms are strongly based on the notion of approximating the distance of a given configuration (or *state*) to the problem's solution (or *goal*). Such approximations are found by means of a computationally efficient function, known as the *heuristic function*.

[1]The name "Rush Hour" is a trademark of Binary Arts, Inc. The game was originally invented by Nobuyuki Yoshigahara in the late 1970s.

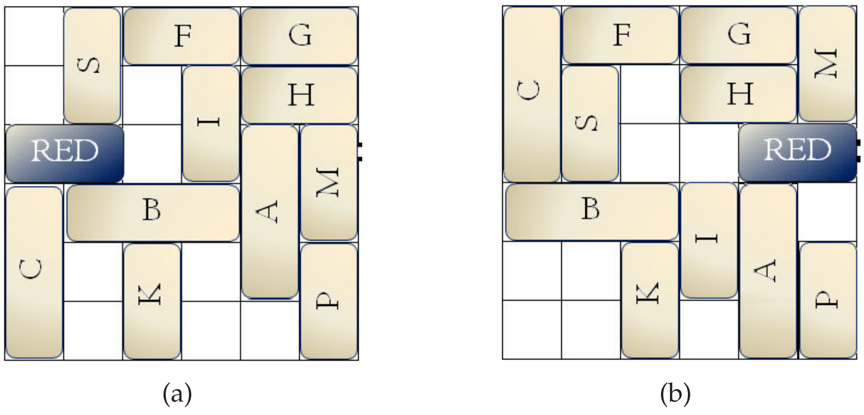


Figure 10.1 (a) A sample Rush Hour configuration. This is problem no. 9 of the problem set shipped with the standard version of the game by Binary Arts, Inc. We refer to this problem as *JAM09*. (b) A possible goal state: the red car has reached the exit tile on the right-hand side of the grid.

By applying the heuristic function to states reachable from the current ones considered, it becomes possible to select more-promising alternatives earlier in the search process, possibly reducing the amount of search effort (typically measured in number of search-tree nodes expanded) required to solve a given problem. The putative reduction is strongly tied to the quality of the heuristic function used: employing a perfect function means simply “strolling” onto the solution (i.e., no search de facto), while using a bad function could render the search less efficient than totally uninformed search, such as breadth-first search (BFS) or depth-first search (DFS).

Until our work (Hauptman et al. [82, 83]) no efficient heuristics had been reported for the Rush Hour puzzle. We believe that the main reason for this is the lack of domain knowledge for the problem, which stems directly from the lack of research into this domain. Moreover, due to the specific structure of the Rush Hour puzzle, standard methods for deriving heuristics—such as solving either subproblems (possibly with pattern databases (Felner et al. [61])) or relaxed problems (e.g., using the Manhattan distance heuristic augmented with linear conflicts (Hansson et al. [73]))—which are typically easy to apply to other well-known domains, are not applicable here (see Section 10.2.3). For these reasons, using IDA* search, or similar algorithms, had probably not been tried.

We used GP to *evolve* heuristic functions for the Rush Hour puzzle (Hauptman et al. [82, 83]). We first constructed a “brute-force,” iterative deepening search algorithm, along with several search enhancements—some culled from the literature, some of our own devise—but with no heuristic functions. As expected, this method worked well on relatively simple boards, and even solved most moderately difficult ones within reasonable bounds of space and time. However, when dealing with complex problems, this method yielded inadequate performance.

We moved on to handcrafting several novel heuristics for Rush Hour, which we then tested empirically. The effect of these heuristics on search efficiency was inconsistent, alternating between decreasing the number of search-tree nodes traversed by 70% (for certain initial configurations) and increasing this number by as much as 170% (for other configurations). It was clear at this point that using our heuristics correctly was a difficult task.

To accomplish this task, we used GP. Our main set of experiments focused on evolving combinations of the basic heuristics devised. We used these basic heuristics as building blocks in a GP setting, where individuals were embodied as ordered sets of search-guiding rules (or *policies*), the components of which were GP trees.

The effect on performance was profound: evolution proved immensely efficacious, managing to combine heuristics of such highly variable utility into composites that were nearly always beneficial, and far better than each separate component.

We thus made a number of important, novel contributions:

- This was the first reported successful attempt to solve the Rush Hour puzzle using intelligent search.
- Along the way we devised several novel heuristics for this domain, some of which could be applied to other domains.
- We demonstrated how policies could be evolved to solve more-difficult problems than ones previously attempted with this method.
- We showed how difficult *solvable* puzzles could be generated, a task that is also considered hard (due to the fact that the decidability question, i.e., whether a given board is solvable or not, is also PSPACE-complete).

10.1 Previous Work

Little work has been done on the Rush Hour puzzle within the computer science community—work which we review herein, along with several related topics.

10.1.1 Rush Hour

Flake and Baum [64] examined a generalized version of Rush Hour, with arbitrary grid size and exit placements, proving that the question of whether an initial configuration is solvable is NP-Hard (the proof uses a reduction from the Satisfiability problem). They then showed the general problem's PSPACE-completeness, by emulating arbitrary recurrent circuits within generalized Rush Hour configurations. Hearn and Demaine [85] proved PSPACE-completeness of sliding block puzzles in the more general case, by demonstrating a reduction from the Quantified Boolean Formula Satisfiability problem.

These formal results imply that there is no polynomial-time algorithm able to find a solution for a given Rush Hour instance (unless $P = PSPACE$), and that the length of the shortest solution of a hard initial configuration grows exponentially with board size n (provided that $P \neq NP$ and $NP \neq PSPACE$). However, Fernau et al. [62] considered the parameterized complexity of the generalized version ($n \times n$) of the game, and showed that solutions can be found in polynomial time when either the total number of vehicles or the total number of moves is bounded by a constant.

Colette et al. [42] focused on finding hard initial configurations for 6x6 Rush Hour by modeling the game in propositional logic, and applying symbolic model-checking techniques to studying the graph of configurations underlying the game. They were able to classify all 3.6×10^{10} possible 6x6 configurations according to the lengths of their shortest solution within approximately 20 hours of computation time. More than 500 of the hardest configurations they found are available online.[2] On the downside, they proved a general theorem regarding the limitations of applying their method to board games, which stems from the fact that the underlying data structure grows exponentially with problem size.

Baum and Durdanovic [15] tackled Rush Hour with an artificial economy of agents. Their best reported solver was able to solve 15 of the 40 standard problems (we solved all, as we shall see). Interestingly, they also tried a GP approach, noting, "We have tried several approaches to getting a Genetic Program to solve these problems, varying the instance presentation scheme and other parameters... it has never learned to solve any of the original problem set." It would seem that with the right GP approach, Rush Hour can be solved.

None of the works above describe an efficient way to solve a given Rush Hour problem. The configurations database constructed by Colette et al. [42] may be used for this purpose (although this is not what the authors intended (Servais [160])). However, we would need to query the database for the distance to solution of each board we encounter during the search. For difficult instances, this would be highly inefficient.

10.1.2 Search heuristics for single-player games

Ample examples are found in the literature of handcrafting heuristics to guide IDA* search in single-player games (as opposed to using an automatic method, such as evolution, which is described in the next section).

Korf [106] described the use of pattern databases, which are precomputed tables of the exact cost of solving various subproblems of an existing problem. This method

[2] `http://cs.ulb.ac.be/~fservais/rushhour`

was used to guide IDA* to solve Rubik's cube. Korf and Felner [107] dealt with disjoint pattern databases for the sliding-tile puzzle problem. In a later work, Felner et al. [61] generalized this notion to include dynamic partitioning of the problem into disjoint subproblems for each state. They applied their method to three different problem domains: the sliding-tile puzzle, the 4-peg Towers of Hanoi problem, and finding an optimal vertex-cover of a graph.

Junghanns and Schaeffer [97, 98] and later Botea et al. [23] dealt with one of the most complex single-player domains: the game of Sokoban. This is a transport puzzle in which the player pushes boxes around a maze and tries to put them in designated locations. The game is challenging due to several reasons, including: large and variable branching factors (potentially over 100); long solutions (some problems require over 500 moves); subgoals are interrelated and thus cannot be solved independently; heuristic estimators are complex; and, deadlocks exist—some moves render the problem unsolvable. Their IDA*-based program, Rolling Stone, equipped with several enhancements—including transposition tables, move ordering, deadlock tables, various macros, and pattern search—was able to solve 52 of the 90-problem standard test suite for Sokoban.

10.1.3 Evolving heuristics for AI planning

Some of the research on evolving heuristics for search is related to the area of AI planning, where heuristics are used to guide search in a way highly similar to single-agent IDA* search, as we employ here.

Aler et al. [4] (see also Aler et al. [2, 3]) proposed a multi-strategy approach for learning heuristics, embodied as ordered sets of control rules (called *policies*) for search problems in AI planning. Policies were evolved using a GP-based system called EvoCK (Aler et al. [3]), whose initial population was generated by a specialized learning algorithm, called Hamlet (Borrajo and Veloso [22]). Their hybrid system (Hamlet-EvoCK) outperformed each of its sub-systems on two benchmark problems often used in planning: Blocks World and Logistics (solving 85% and 87% of the problems in these domains, respectively). Note that both these domains are far simpler than Rush Hour, mainly because they are less constrained.

Levine and Humphreys [115] also evolved policies and used them as heuristic measures to guide search for the Blocks World and Logistic domains. Their system, L2Plan, included rule-level genetic operators (for dealing with entire rules), as well as simple local search to augment GP crossover and mutation. They demonstrated some success in these two domains, although hand-coded policies sometimes outperformed the evolved ones.

10.2 Method

Our work on the Rush Hour puzzle developed through four main phases:

1. Construction of an iterative deepening (uninformed) search engine, enhanced with macro steps. Heuristics were not used during this phase.
2. Design of several novel heuristics, which were tested in conjunction with our engine.
3. Evolution of combinations of heuristics, along with conditions for applying them, using GP.
4. Evolution of difficult 8x8 boards, using our engine, augmented by heuristics, to test board fitness.

First we briefly describe our test suite of problems.

10.2.1 Test suite

The Rush Hour game (standard edition) is shipped along with 40 problems, grouped into four difficulty levels (Beginner, Intermediate, Advanced, and Expert). We designate these problems as *JAM01*, . . . , *JAM40*. The minimal solution length (i.e., minimal number of moves) is an oft-used rough estimate of problem difficulty. Their optimal-solution lengths vary from 8 to 52 moves.

To add harder problems to the test suite we expanded it with the 200 most difficult configurations published online by Colette et al. [42], whose optimal-solution lengths vary from 53 to 93 moves. We denote these *SER1*, . . . , *SER200*. We are now in possession of an ample test suite, with problems of increasing difficulty—from simplest (*JAM1*) to hardest (*SER200*). The problem *SER200* is the hardest 6x6 Rush Hour configuration, as reported by Colette et al. [42].

In order to work with more-challenging problems, we evolved 15 difficult solvable 8x8 boards using the method described in Section 10.2.5. This resulted in much more difficult boards, which are denoted $E1, \ldots, E15$. Solution lengths for these problems vary from 90 to 120 moves.

10.2.2 Enhanced iterative deepening search

We initially implemented standard iterative deepening search (Korf [105]) as the heart of our game engine. This algorithm may be viewed as a combination of DFS and BFS: starting from a given configuration (e.g., the initial state), with a minimal depth bound, we perform a DFS search for the goal state through the graph of game states (in which vertices represent game configurations, and edges—legal moves). Thus, the algorithm requires only $\theta(n)$ memory, where n is the depth of the search tree. If we succeed, the path is returned. If not, we increase the depth bound by a fixed amount, and restart the search. Note that since the search is incremental, when we find a solution we are guaranteed that it is optimal since a shorter solution would have been found in a previous iteration (more precisely, the solution is near-optimal, given that the depth increase is usually larger than one). However, for difficult problems, such as Rush Hour and Sokoban, finding *a* solution is sufficient, and there is typically no requirement of finding the optimal solution.

Our game engine received as input a Rush Hour board, as well as some run parameters, and output a solution (i.e., a list of moves) or a message indicating that the given instance could not be solved within the time or space constraints given. The idea of limiting the search derives from the work of Junghanns and Schaeffer [96] for the domain of Sokoban, where a limit of $20,000,000$ nodes was set for each problem. Since the Sokoban standard test suite they used contains problems that typically require more resources than 6x6 Rush Hour problems, we used a stricter limit of $1,500,000$ nodes (since both depth and branching factor are lower for our problem, and hence search trees are smaller, this bound was reasonable).

The basic version of the game engine also included several simple search macros (sets of moves grouped together as a single move (Korf [104])), such as moving a vehicle several tiles (if applicable) as a single move, and always moving the red car toward the exit as a single move when possible.

Using search alone, along with macro moves, we were able to solve all boards of the first problem set, *JAM01*, . . . , *JAM40*, expanding fewer than 500,000 nodes. However, 20 problems from the group *SER150*, . . . , *SER200* still took over $1,500,000$ nodes to solve, which violated our space bound.

We concluded that uninformed search, even when augmented by several enhancements, is not powerful enough to solve difficult instances of this problem. Thus, it was clear that heuristics were needed.

10.2.3 Heuristics and advisors for Rush Hour

In this section we describe the heuristics we devised, which were used to estimate the distance to the goal from a given board. In addition, we define *advisors* (or auxiliary functions), incorporating domain features, i.e., functions that do not provide an estimate of the distance to the goal but which are nonetheless beneficial in a GP setting

We encountered difficulties when attempting to implement standard methods for devising heuristics, mainly in the form of problem relaxation (Pearl [136]) (e.g., with pattern databases (Culberson and Schaeffer [44]; Felner et al. [61]), and more specifically, the Manhattan-Distance heuristic (Hansson et al. [73])). This methodology is difficult to apply to the Rush Hour puzzle due to the structure of the domain—every vehicle can potentially have a substantial effect over the problem as a whole. Alleviating the constraints imposed by even a single vehicle (e.g., by removing it or allowing it to move freely) in order to obtain a heuristic value for the above-mentioned methods may render a difficult problem easy (for example, if we remove vehicles M and I in problem *JAM09* of Figure 10.1, the problem can be solved in a mere two steps). These ideas can, however, be refined into useful heuristics, which we describe below.

Additionally, when we use heuristics to guide search, we move from simple iterative deepening, to iterative deepening A* (IDA*) (Korf [105]). This algorithm operates similarly to iterative deepening, except for using the heuristic value to guide the search at each node (this method is known as move ordering (Reinefeld and Marsland [145])).

We now turn to describing our heuristics.

Blockers estimation The first obvious estimate to the closeness of a board configuration to the goal is the number of vehicles blocking the red car's path to the exit, because when this number reaches zero, the problem is solved. However, simply counting the number of such vehicles is not very informative (e.g., for several difficult problems only one vehicle blocks the path in the initial configuration, yet still the distance to the solution is large).

Computing a lower-bound estimate of the number of moves required to move each vehicle out of the red car's path provides a better measure. This entails estimating the number of moves needed to move each vehicle blocking these vehicles, and so on, recursively. The numbers are then summed, with some redundancy checks to avoid counting the same vehicle more than once. When we have to choose between two possible directions of moving a vehicle out of the way, we compute both and retain the minimal value.

This heuristic, which we dubbed `BlockersLowerBound`, reduced the number of nodes for several difficult problems by 70% when tested empirically, although for some problems it actually *increased* the node count by more than 170%, compared to iterative deepening with no heuristics. This latter increase was probably because some parts of the solutions required moves that increased the blockers' estimate, and this heuristic guided the search away from them. What was missing was a measure of *when* to apply the heuristic. Moreover, errors are expected due to the fact that no estimator for a difficult problem can be perfect.

The ambivalent nature of this heuristic—often helpful, at times detrimental—is also true of the other heuristics introduced below.

Goal distance The following heuristic, dubbed `GoalDistance`, is a possible way to implement the Manhattan-Distance heuristic, as used for the sliding-tile puzzle (e.g., Korf and Felner [107]). To devise such a measure we need to count each vehicle's distance from its designated place in the goal board. However, compared to the sliding tiles or Rubik's cube, the final position for each vehicle is not known in advance.

In order to get around this problem we constructed, for each initial configuration, a *deduced* goal: a board containing a clear path to the goal, where all interfering vehicles (and vehicles blocking them) have been "forcibly" positioned (i.e., ignoring move rules while still forbidding two vehicles from occupying the same tile) in possible locations in which they are no longer blocking the red car. If necessary, we also move the cars blocking their paths in the same manner. Devising a good heuristic function for deducing goal boards was not easy, as it required some complex reasoning for several cases. Moreover, there is no guarantee, especially for difficult problems, that the deduced goal board will actually be the correct goal board. However, this heuristic proved to be a useful building block for high-fitness individuals.

Hybrid blockers distance Here we combine the essence of the previous two heuristics. Instead of merely summing each vehicle's distance to its location in the deduced goal, we also counted the number of vehicles in its path, and added it to the sum. This heuristic was dubbed `Hybrid`. Note that we did not perform a full blocker's estimation for each vehicle (only the number of blockers was summed) because computing a more detailed measure would have been time consuming and would have sometimes produced larger estimates than required (since the same vehicle may block several other vehicles, and it would be counted as a blocker for each of them).

Advisors Additional auxiliary functions, incorporating domain features, were used to assign scores to boards, including:

- `IsReleasingMove`: Checks if the last move made increases the number of vehicles free to move.
- `IsMoveToSecluded`: Did the last move place a car in a position to which no other car can move?
- `ProblemDifficulty`: The given difficulty level of the problem at hand (this information is also available to humans when solving the problems shipped with the game).

As noted above, these auxiliary functions are not heuristics as they do not provide an estimate of the distance to the goal but they are nonetheless beneficial in a GP setting.

For a complete list of heuristics and advisors, see Table 10.1.

Table 10.1 Terminal set of an individual program in the population. B: Boolean, R: Real or Integer.

Node name	Type	Return value
Used in *Condition* and *Value* trees		
`BlockersLowerBound`	R	A lower bound on the number of moves required to remove blocking vehicles out of the red car's path
`GoalDistance`	R	Sum of all vehicles' distances to their locations in the deduced-goal board
`Hybrid`	R	Same as `GoalDistance`, but also includes the number of vehicles between each car and its designated location
Used in *Condition* trees only		
`IsMoveToSecluded`	B	Did the last move taken position the vehicle at a location that no other vehicle can occupy?
`IsReleasingMove`	B	Did the last move made add new possible moves?
g	R	Distance from the initial board
`PhaseByDistance`	R	$g \div (g + \texttt{GoalDistance})$
`PhaseByBlockers`	R	$g \div (g + \texttt{BlockersLowerBound})$
`NumberOfSiblings`	R	The number of nodes expanded from the parent of the current node
`DifficultyLevel`	R	The difficulty level of the given problem, relative to other problems in the current problem set
`ERC`	R	Ephemeral Random Constant in the range $[0,1)$
Used in *Value* trees only		
$\{0,\ldots,9\}$	R	Numeric terminals

10.2.4 Evolving heuristics

Using the heuristics we devised to make search more efficient is a difficult task, as it involves solving two major subproblems:

1. Finding exact conditions regarding *when* to apply each heuristic (in order to avoid the strong inconsistent effect on performance mentioned above).
2. Combining several estimates to get a more accurate one. We hypothesized that different areas of the search space might benefit from the application of different heuristics.

Solving the above subproblems means traversing an extremely large search space of possible conditions and combinations. This is precisely where we turned to evolution.

Genome As we wanted to embody both application conditions and combinations of estimates, we decided to evolve ordered sets of control rules, or *policies*. As stated above, policies were evolved successfully with GP to solve search problems—albeit simpler ones (for example, see Aler et al. [4] and Borrajo and Veloso [22], mentioned above).

Policies typically have the following structure:[3]

$RULE_1$: IF $Condition_1$ THEN $Value_1$
.
.
.
$RULE_N$: IF $Condition_N$ THEN $Value_N$
$DEFAULT$: $Value_{N+1}$

where $Condition_i$ and $Value_i$ represent conditions and estimates, respectively.

Policies are used by the search algorithm in the following manner: The rules are ordered such that we apply the first rule that "fires" (meaning its condition is true for a given board), returning its *Value* part. If no rule fires, the value is taken from the last (default) rule: $Value_{N+1}$. Thus, individuals in the evolving GP population, while in the form of policies, are still board evaluators (or heuristics)—the value returned by the activated rule is an arithmetic combination of heuristic values, and is thus a heuristic value itself. This suits our requirements: rule ordering and conditions control when we apply a heuristic combination, and values provide the combinations themselves.

Thus, with N being the number of rules used, each individual in the evolving population contained N *Condition* GP-trees and $N + 1$ *Value* GP-trees. After experimenting with several sizes of policies, we settled on $N = 5$, providing us with enough rules per individual, while avoiding "heavy" individuals with too many rules. The depth limit used both for the *Condition* and *Value* trees was empirically set to 5.

The function set included the functions $\{AND,OR,\leq,\geq\}$ for condition trees and the functions $\{\times,+\}$ for the value trees. The heuristics of Table 10.1 were used as terminals. To get a more uniform calculation we normalized the values returned by terminals of *Condition* trees to lie within the range $[0, 1]$, by maintaining a maximal possible value for each terminal, and dividing the returned value by it. For example, `BlockersLowerBound` might return an estimate of 20 moves, with the maximal value for this terminal determined empirically to be 40, thus setting the return value to 0.5.

Genetic operators We used the standard crossover and mutation operators; however, before selecting the crossover or mutation point, we first randomly selected rules whose conditions (or values) were to be substituted. Crossover was only performed between nodes of the same type (using Strongly Typed GP—see Chapter 2).

We also added rule-crossover and rule-mutation operators, whose purpose was to swap entire randomly selected rules between individuals and within the same

[3]Actually, policies are commonly defined as rules where the result is an *action*, not a value. However, actions lead to the selection of a child node, and are thus similar to heuristic values.

individual, respectively. One of the major advantages of policies is that they facilitate the use of diverse genetic operators such as these.

Test and training sets Individuals were evolved with fixed groups of problems (one group per run): The suite of all 6x6 problems (*JAM01* through *SER200*) was divided into five equally sized groups (48 problems per group). Additionally, we used a sixth group containing 15 difficult 8x8 problems, discovered through evolution (see Section 10.2.5).

For each group, 10 problems (taken from the 20 most difficult ones) were tagged as *test* problems, and the remaining ones were used as *training* problems. Training problems were used for fitness purposes, while test problems were used to test the best individual in order to assess the overall progress of the run.

Fitness Fitness scores were obtained by performing full IDA* search, with the given individual used as the heuristic function. For each solved board we assigned to the individual a score equal to the percentage of nodes reduced, compared with searching with no heuristics. For unsolved boards the score was 0. Scores were averaged over 10 randomly selected boards from the training set.

GP parameters We experimented with several configurations, finally settling upon: population size—between 50 and 100, generation count—between 100 and 400, reproduction probability—0.5, crossover probability—0.4 , and mutation probability—0.1. For both the crossover and mutation operators, we used a uniform distribution for selecting trees inside individuals.

10.2.5 Evolving difficult solvable 8x8 boards

Since our enhanced IDA* search solved over 90% of the 6x6 problems (including 30% of the 50 most difficult problems reported by Colette et al. [42]), well within the space bounds (in fact, with far fewer requirements), and, moreover, we wanted to demonstrate our method's scalability to larger boards, we needed to design more challenging problems. This we did through evolution.

We generated the initial population of boards by taking solvable 6x6 boards and expanding each one to size 8x8 by "wrapping" it with a perimeter of empty cells (i.e., each 6x6 board was embedded in the center of an empty 8x8 board). Then, using simple mutation operators, which randomly either added, swapped, or deleted vehicles, we assigned to each board a fitness score equal to the number of boards required to solve it using our enhanced IDA* search. A board that could not be solved within 15 minutes (on a Linux-based PC, with processor speed 3GHz, and 2GB of main memory) received a fitness score of 0.

We repeated this process until evolution showed no further improvement. While this mutation-based process might generate genotypically similar boards, they are phenotypically different due to the domain structure, described above. The most difficult 8x8 board found required 26,000,000 nodes to solve with no-heuristic, iterative deepening (the None column in Table 10.2).

10.3 Results

We assessed the performance of the heuristics with the same scoring method used for fitness computation, except we averaged over the entire test set instead of over boards taken from the training set.

We compared several heuristics: the three handcrafted heuristics described in Section 10.2.3; a handcrafted policy that we designed ourselves by combining the basic (handcrafted) heuristics; and the top full-fledged policy developed via GP, which we took from the best run.

Results are summarized in Table 10.2. As can be seen, the average performance of our handcrafted heuristics did not show significant improvement over iterative deepening with no heuristic (although `BlockersLowerBound` proved better than the other two). While our handcrafted policy fared somewhat better, the evolved policy yielded the best results by a wide margin, especially given the increasing difficulty of node reduction as search gets better. *Overall, evolved policies managed to cut the required amount of search to 40% for 6x6 boards and to 10% for 8x8 boards, compared to iterative deepening.*

It should also be noted that performance over 8x8 boards was better relative to 6x6 boards. This may be ascribed to the fact that while the entire space of difficult 6x6 boards is covered by our test and training sets, this is not the case for our 8x8 boards. Still, considering that the evolved 8x8 boards we used proved immensely difficult for no-heuristic iterative deepening (requiring over 20,000,000 nodes to solve in some cases), *results show that our method is scalable, which is non-trivial for a PSPACE-complete problem.*

Next, we turned to comparing the performance of evolution to that of humans. Since we had no exact measure for the number of boards examined by humans for this problem, we turned to another measure: solution time. All comparisons performed so far treated only the number of nodes expanded, due to the fact that the amount of time required to solve a problem is linearly related to the number of nodes (i.e., fewer nodes implies less time). This is obvious since the engine's speed (or nodes per second) is constant. The time data was collected along with the number of nodes for all our runs.

Table 10.2 Average percentage of nodes required to solve *test* problems, with respect to the number of nodes scanned by iterative deepening (shown as 100% in the second column). $H1$: the heuristic function `BlockersLowerBound`; $H2$: `GoalDistance`; $H3$: `Hybrid`. Hc is our handcrafted policy, and GP is the best evolved policy, selected according to performance on the training set. 6x6 represents the test cases taken from the set $\{JAM01\ldots,SER200\}$. 8x8 represents the 15 most difficult 8x8 problems we evolved. Values were rounded to nearest integer.

Heuristic: Problem	None	$H1$	$H2$	$H3$	Hc	GP
6x6	100%	72%	94%	102%	70%	40%
8x8	100%	69%	75%	70%	50%	10%

Table 10.3 Time (in seconds) required to solve problems *JAM01,...,JAM40* by: ID—iterative deepening, *Hi*—average of our three handcrafted heuristics, *Hc*—our handcrafted policy, *GP*—our best evolved policy, and human players (average of top 5). Problems are divided into five groups, with the average presented below.

Problems	ID	*Hi*	*Hc*	*GP*	Humans
JAM01...*JAM08*	0.2	0.65	0.06	0.03	2.6
JAM09...*JAM16*	1.7	0.35	1.74	0.6	8.15
JAM17...*JAM24*	2.4	1.8	1.08	0.83	10.32
JAM25...*JAM32*	6.3	1.6	3.94	1.17	14.1
JAM33...*JAM40*	7.65	2.8	7.71	2.56	20.00
Average	3.65	1.44	2.69	1.04	11.03

Data regarding human performance was available online [4] in the form of High Scores (sorted by time to solution) for each of the problems *JAM01* to *JAM40*. The site contains thousands of entries for each problem, so the data is reliable, although it doesn't necessarily reflect the *best* human performance. We compared the time required to solve the 40 standard problems by humans to the runtime of several algorithms: iterative deepening, *Hi* (representing the average time of our three handcrafted heuristics), our handcrafted policy, and our best evolved policy. Results are presented in Table 10.3. Clearly, all algorithms tested are much faster than human players, and evolved policies are the fastest. This emphasizes the fact that evolved policies save *both* search time *and* space.

10.4 Discussion

We designed an IDA*-based solver for the Rush Hour puzzle, a problem to which intelligent search has not been applied to date. With no heuristics we managed to

[4] `http://www.trafficjamgame.com` (the site seems to have gone defunct)

solve most 6x6 problems within reasonable time and space limits, but only a few of our newly evolved, difficult 8x8 problems. After designing several novel heuristics for Rush Hour we discovered that their effect on search was limited and somewhat inconsistent, at times reducing node count but in several cases actually *increasing* it. Solving the problem of correctly applying our heuristics was done by evolving policies with GP (which outperformed a less successful attempt to devise policies by hand). To push the limit yet further we evolved difficult 8x8 boards, which aided in the training of board-solving individuals by augmenting the fitness function.

Our results show that the improvement attained with heuristics increased substantially when evolution entered into play: search with evolved policies required less than 50% of the nodes required by search with non-evolved heuristics. As a result, 85% of the problems, which were unsolvable before, became solvable within the 1,500,000 node limit, including several difficult 8x8 instances. The rest of the problems were solved using more than 1.5M nodes.

There are several conceivable extensions to our work. First, we are confident that better heuristics for Rush Hour remain to be discovered. For example, it is possible to take the ideas underlying the `GoalDistance` heuristic and apply them to deducing more configurations along the path to the goal (calculating distances to them, as we did with `GoalDistance`). While this calculation requires more preprocessing we are certain that it will yield a more efficient algorithm, since we would be providing search with a more detailed map to the goal.

Handcrafted heuristics may themselves be improved by evolution. This could be done by breaking them into their elemental pieces, and evolving their combinations thereof. For example, the values we add when computing `BlockerLowerBound` might be real numbers, not integers, whose values evolve subject to more domain knowledge. It is possible both to evolve a given heuristic as the only one used in IDA* search or to evolve it as part of a larger structure of heuristics, itself subject to (piecewise) evolution. Totally new heuristics may also be evolved using parts comprising several known heuristics (just like the `Hybrid` heuristic was conceptualized as a combination of `BlockersLowerBound` and `GoalDistance`).

As our search keeps improving and we use it to find more-difficult solvable configurations, which, in turn, aid in evolving search, we feel that the limits of our method (i.e., solving the most difficult boards possible within the given bounds) have not yet been reached. As we are dealing with a PSPACE-complete problem it is certain that if we take large-enough boards, solving them would become infeasible. However, for the time being we plan to continue discovering the most challenging configurations attainable.

Many single-agent search problems fall within the framework of AI-planning problems, and Rush Hour is no exception. Algorithms for generating and maintaining agendas, policies, interfering subgoals, relaxed problems, and other methodologies mentioned above are readily available, provided we encode Rush Hour as a planning domain (e.g., with ADL (Pednault [137])). However, using evolution in conjunction with these techniques is not trivial.

Chapter 11

FreeCell

A well-known, highly popular example within the domain of discrete puzzles is the card game of FreeCell. Starting with all cards randomly divided into *k* piles (called *cascades*), the objective of the game is to move all cards onto four different piles (called *foundations*)—one per suit—arranged upwards from the ace to the king. Additionally, there are initially empty cells (called *FreeCells*), whose purpose is to aid with moving the cards. Only exposed cards can be moved, either from FreeCells or cascades. Legal move destinations include: a home (foundation) cell, if all previous (i.e., lower) cards are already there; empty FreeCells; and, on top of a next-highest card of opposite color in a cascade (Figure 11.1). FreeCell was proven by Helmert [87] to be NP-complete. Computational complexity aside, many (oft-frustrated) human players (including the author) will readily attest to the game's hardness. The attainment of a competent machine player would undoubtedly be considered a human-competitive result.

FreeCell remained relatively obscure until it was included in the Windows 95 operating system (and in all subsequent versions), along with 32,000 problems—known as *Microsoft 32K*—all solvable but one (this latter, game #11982, was proven to be unsolvable). Due to Microsoft's move, FreeCell has been claimed to be one of the world's most popular games (Bacchus [10]). The Microsoft version of the game comprises a standard deck of 52 cards, 8 cascades, 4 foundations, and 4 FreeCells. Though limited in size, this FreeCell version still requires an enormous amount of search, due both to long solutions and to large branching factors. Thus it remains out of reach for optimal heuristic search algorithms, such as A* and iterative deepening A*, which we encountered in the previous chapter. FreeCell remains unsolvable even when powerful enhancement techniques are employed, such as transposition tables (Frey [68]; Taylor and Korf [175]) and macro moves (Korf [104]).

Despite there being numerous FreeCell solvers available via the Web, few have been written up in the scientific literature. The best published solver to date is that of Heineman [86], able to solve 96% of Microsoft 32K using a hybrid A* / hill-

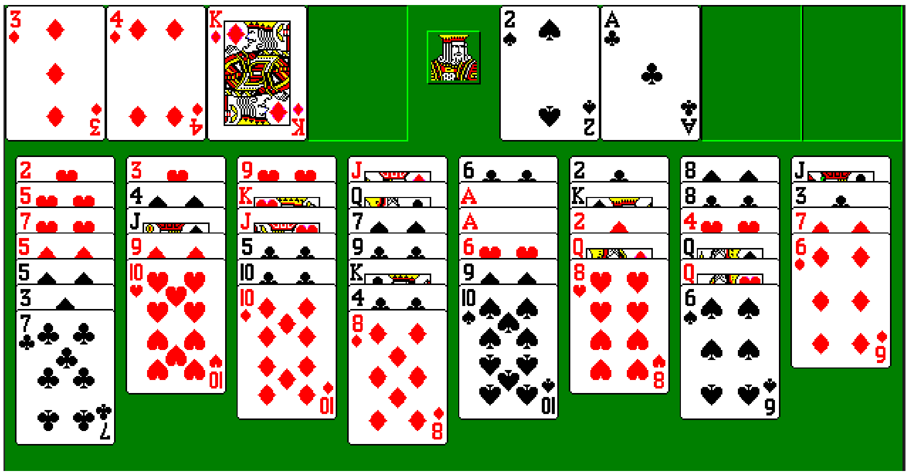


Figure 11.1 A FreeCell game configuration. Cascades: Bottom 8 piles. Foundations: 4 upper-right piles. FreeCells: 4 upper-left piles. Note that cascades are not arranged according to suits, but foundations are. Legal moves for the current configuration: 1) moving 7♣ from the leftmost cascade to either the pile fourth from the left (on top of the 8♢), or to the pile third from the right (on top of the 8♡); 2) moving the 6♢ from the right cascade to the left one (on top of the 7♣); and 3) moving any single card on top of a cascade onto the empty FreeCell.

climbing search algorithm called *staged deepening* (henceforth referred to as the *HSD* algorithm). The HSD algorithm, along with a heuristic function, forms Heineman's FreeCell solver (we shall distinguish between the HSD algorithm, the HSD heuristic, and the HSD solver—which includes both). Heineman's system exploits several important characteristics of the game, elaborated below.

As noted in the previous chapter, search algorithms for puzzles are strongly based on the notion of approximating the distance of a given configuration to the problem's solution by means of a heuristic function. By applying this function to states reachable from the current ones considered, it becomes possible to select more-promising alternatives earlier on in the search process, possibly reducing the amount of search effort required to solve a given problem.

This chapter presents our work on FreeCell (Elyasaf et al. [56]). Our main set of experiments focused on evolving combinations of handcrafted heuristics we devised specifically for FreeCell. We used these basic heuristics as building blocks in a genetic algorithm (GA) setting, where individuals represented the heuristics' weights. We used Hillis-style competitive coevolution (Hillis [89]) to simultaneously coevolve good solvers and various deals (initial card configurations) of varying difficulty levels.

We will show that not only do we solve over 98% of the Microsoft 32K problem set, a result far better than the best solver on record, but we also do so significantly more efficiently in terms of time to solve, space (number of search-tree nodes expanded), and solution length (number of nodes along the path to the correct solution found).

We thus made a number of important, novel contributions (Elyasaf et al. [56]):

- Using a genetic algorithm we developed the strongest known heuristic-based solver for the game of FreeCell.
- Along the way we devised several novel heuristics for FreeCell, many of which could be applied to other domains and games.
- We pushed the limit of what has been done with evolution further, FreeCell being one of the most difficult single-player domains (if not the most difficult) to which evolutionary algorithms have been applied to date.

At the end of this chapter we will present some preliminary results showing that policy-based GP can improve the results even further.

11.1 Previous Work

We survey below the work done on FreeCell along with some related topics. Note that research into planning systems, discussed in the previous chapter, is also relevant herein.

Most reported work on FreeCell has been done in the context of automated planning, a field of research in which generalized problem solvers (known as *planning systems* or *planners*) are constructed and tested across various benchmark puzzle domains. FreeCell was used as such a domain both in several International Planning Competitions (IPCs) (e.g., Long and Fox [116]), and in numerous attempts to construct state-of-the-art planners reported in the literature (e.g., Coles and Smith [41]; Yoon et al. [186]). The version of the game we solve herein, played with a full deck of 52 cards, is considered to be one of the most difficult domains for classical planning (Bacchus [10]), evidenced by the poor performance of general-purpose planners.

There are numerous solvers that were developed specifically for FreeCell available via the web, the best of which is that of Heineman [86]. Although it fails to solve 4% of Microsoft 32K, Heineman's solver significantly outperforms all other solvers in terms of both space and time. We elaborate on this solver in Section 11.2.1.

Terashima-Marín et al. [176] compared two models to produce hyper-heuristics[1] that solved two-dimensional regular and irregular bin-packing problems, an NP-Hard problem domain. The learning process in both of the models produced a rule-based mechanism to determine which heuristic to apply at each state. Both models

[1]In the hyper-heuristic framework, within which our work on puzzles also falls, the system is provided with a set of predefined or preexisting heuristics for solving a certain problem, and it tries to discover the best manner in which to apply these heuristics at different stages of the search process. The aim is to find new, higher-level heuristics, or hyper-heuristics (Bader-El-Den et al. [11]).

outperformed the continual use of a single heuristic. We note that their rules classified a state and then applied a (single) heuristic, whereas we apply a *combination* of heuristics at each state, which we believed would perform better.

Note that *none* of the deals in the Microsoft 32K problem set could be solved with blind search, nor with IDA* equipped with handcrafted heuristics, evidencing that FreeCell is far more difficult than Rush Hour.

11.2 Methods

Our work on the game of FreeCell progressed in five phases:

1. Construction of an iterative deepening (uninformed) search engine, endowed with several enhancements. Heuristics were not used during this phase.

2. Guiding an IDA* search algorithm with the HSD heuristic function (HSDH).

3. Implementation of the HSD algorithm (including the heuristic function).

4. Design of several novel heuristics for FreeCell.

5. Learning weights for these novel heuristics using Hillis-style coevolution.

11.2.1 Search algorithms

Iterative deepening We initially tested the enhanced iterative deepening search algorithm, described in the previous chapter. An iterative deepening-based game engine received as input a FreeCell initial configuration (a deal), as well as some run parameters, and output a solution (i.e., a list of moves) or an indication that the deal could not be solved.

We observed that even when we permitted the search algorithm to use all the available memory (2GB in our case, as opposed to the Rush Hour case where the node count was limited) virtually all Microsoft 32K problems could not be solved. Hence, we deduced that heuristics were essential for solving FreeCell instances—uninformed search alone was insufficient.

Iterative deepening A* Given that the HSD algorithm outperforms all other solvers, we implemented the heuristic function used by HSD (described in Section 11.2.2) along with iterative deepening A* (IDA*). As explained in Chapter 10, IDA* operates similarly to iterative deepening, except that in the DFS phase heuristic values are used to determine the order by which children of a given node are visited. This move ordering is the only phase wherein the heuristic function is used—the open list structure is still sorted according to depth alone.

IDA* underperformed where FreeCell was concerned, unable to solve many instances (deals). Even using a strong heuristic function, IDA*—despite its success in other difficult domains—yielded inadequate performance: less than 1% of the deals we tackled were solved, with all other instances resulting in memory overflow.

At this point we opted for employing the HSD algorithm in its entirety, rather than merely the HSD heuristic function.

Staged deepening Staged deepening—used by the HSD algorithm—is based on the observation that there is no need to store the entire search space seen so far in memory. This is so because of a number of significant characteristics of FreeCell:

- For most states there is more than one distinct permutation of moves creating valid solutions. Hence, very little backtracking is needed.
- There is a relatively high percentage of irreversible moves: according to the game's rules a card placed in a home cell cannot be moved again, and a card moved from an unsorted pile cannot be returned to it.
- If we start from game state s and reach state t after performing k moves, and k is large enough, then there is no longer any need to store the intermediate states between s and t. The reason is that there is a solution from t (first characteristic) and a high percentage of the moves along the path are irreversible anyway (second characteristic).

Thus, the HSD algorithm may be viewed as two-layered IDA* with periodic memory cleanup. The two layers operate in an interleaved fashion: 1) At each iteration, a local DFS is performed from the head of the open list up to depth k, with no heuristic evaluations, using a transposition table—storing visited nodes—to avoid loops; 2) Only nodes at *precisely* depth k are stored in the open list,[2] which is sorted according to the nodes' heuristic values. In addition to these two interleaved layers, whenever the transposition table reaches a predetermined size, it is emptied entirely, and only the open list remains in memory. Algorithm 11.1 presents the pseudocode of the HSD algorithm. S was empirically set by Heineman to 200,000.

Compared with IDA*, HSD uses fewer heuristic evaluations (which are performed only on nodes entering the open list), and does periodic memory cleanup, resulting in significant reduction in time. Reduction is achieved through the second layer of the search, which stores enough information to perform backtracking (as stated above, this does not occur often), and the size of T is controlled by overwriting nodes.

[2]Note that since we are using DFS and not BFS we do not find all such states.

Algorithm 11.1 HSD (Heineman's staged deepening)

```
    // Parameter: $S$, size of transposition table
 1: $T \leftarrow$ initial state
 2: **while** $T$ not empty **do**
 3:    $s \leftarrow$ remove best state in $T$ according to heuristic value
 4:    $U \leftarrow$ all states exactly $k$ moves away from $s$, discovered by DFS
 5:    $T \leftarrow$ merge($T$, $U$)
       // merge maintains $T$ sorted by descending heuristic value
       // merge overwrites nodes in $T$ with newer nodes from $U$
       // of equal heuristic value
 6:    **if** size of transposition table $\geq S$ **then**
 7:       clear transposition table
 8:    **end if**
 9:    **if** $goal \in T$ **then**
10:       return path to goal
11:    **end if**
12: **end while**
```

Although the staged deepening algorithm does not guarantee an optimal solution, we noted in the previous chapter that for difficult problems finding *a* solution is sufficient.

When we ran the HSD algorithm it solved 96% of Microsoft 32K, as reported by Heineman.

At this point we were at the limit of the current state-of-the-art for FreeCell, and we turned to evolution to attain better results. However we first needed to develop additional heuristics for this domain.

11.2.2 FreeCell heuristics

In this section we describe the heuristics we used, all of which estimate the distance to the goal from a given game configuration:

`Heineman's Staged Deepening Heuristic (HSDH)`: This is the heuristic used by the HSD solver. For each foundation pile (recall that foundation piles are constructed in ascending order), locate within the cascade piles the next card that should be placed there, and count the cards found on top of it. The returned value is the sum of this count for all foundations. This number is multiplied by 2 if there are no free FreeCells or empty cascade piles (reflecting the fact that freeing the next card is harder in this case).

`NumberWellPlaced`: Count the number of *well-placed* cards in cascade piles. A pile of cards is well placed if *all* its cards are in descending order and alternating colors.

`NumCardsNotAtFoundations`: Count the number of cards that are not at the foun-

Table 11.1 List of heuristics used by the genetic algorithm. R: Real or Integer.

Node name	Type	Return value
`HSDH`	R	Heineman's staged deepening heuristic
`NumberWellPlaced`	R	Number of well-placed cards in cascade piles
`NumCardsNotAtFoundations`	R	Number of cards not at foundation piles
`FreeCells`	R	Number of free FreeCells and cascades
`DifferenceFromTop`	R	Average value of top cards in cascades minus average value of top cards in foundation piles
`LowestHomeCard`	R	Highest possible card value minus lowest card value in foundation piles
`HighestHomeCard`	R	Highest card value in foundation piles
`DifferenceHome`	R	Highest card value in foundation piles minus lowest one
`SumOfBottomCards`	R	Highest possible card value multiplied by number of suites, minus sum of cascades' bottom card

dation piles.

`FreeCells`: Count the number of free FreeCells and cascades.

`DifferenceFromTop`: The average value of the top cards in cascades, minus the average value of the top cards in foundation piles.

`LowestHomeCard`: The highest possible card value (typically the king) minus the lowest card value in foundation piles.

`HighestHomeCard`: The highest card value in foundation piles.

`DifferenceHome`: The highest card value in the foundation piles minus the lowest one.

`SumOfBottomCards`: Take the highest possible sum of cards in the bottom of cascades (e.g., for 8 cascades, this is $4 * 13 + 4 * 12 = 100$), and subtract the sum of values of cards actually located there. For example, in Figure 11.1, `SumOfBottomCards` is $100 - (2 + 3 + 9 + 11 + 6 + 2 + 8 + 11) = 48$.

Table 11.1 provides a summary of all heuristics.

Experiments with these heuristics demonstrated that each one separately (except for HSDH) was not good enough to guide search for this difficult problem. Thus we turned to evolution.

11.2.3 Evolving heuristics for FreeCell

As we saw previously, combining several heuristics to get a more accurate one is considered one of the most difficult problems in contemporary heuristics research (Burke et al. [26]; Samadi et al. [153]). Herein we tackle a subproblem, that of combining heuristics by *arithmetic* means, e.g., by summing their values or taking the maximal value.

The problem of combining heuristics is difficult primarily because it entails traversing an extremely large search space of possible numeric combinations and game configurations. To tackle this problem we used a genetic algorithm. Below we describe the elements of our setup in detail.

Genome Each individual comprised 9 real values in the range $[0, 1)$, representing a linear combination of all 9 heuristics described in Table 11.1. Specifically, the heuristic value, H, designated by an evolving individual was defined as $H = \sum_{i=1}^{9} w_i h_i$, where w_i is the ith weight specified by the genome, and h_i is the ith heuristic shown in Table 11.1. To obtain a more uniform calculation we normalized all heuristic values to within the range $[0, 1]$ by maintaining a maximal possible value for each heuristic, and dividing by it. For example, `DifferenceHome` returns values in the range $[0, 13]$ (13 being the difference between the king's value and the ace's value), and the normalized values were attained by dividing by 13.

GA operators and parameters We applied GP-style evolution in the sense that first an operator (reproduction, crossover, or mutation) was selected with a given probability, and then one or two individuals were selected in accordance with the operator chosen (Chapter 2). We used standard fitness-proportionate selection and single-point crossover. Mutation was performed in a manner analogous to bitwise mutation by replacing with independent probability 0.1 a (real-valued) weight by a new random value in the range $[0, 1)$.

We experimented with several parameter settings, finally settling on: population size—between 40 and 60, generation count—between 300 and 400, reproduction probability—0.2, crossover probability—0.7, mutation probability—0.1, and elitism set size—1.

Training and test sets The Microsoft 32K suite contains a random assortment of deals of varying difficulty levels. In each of our experiments 1,000 of these deals were randomly selected for the training set and the remaining 31K were used as the test set.

Fitness An individual's fitness score was obtained by running the HSD algorithm on deals taken from the training set, with the individual used as the heuristic function. Fitness equaled the average search-node reduction ratio. This ratio was obtained by comparing the reduction in number of search nodes—averaged over solved deals—with the average number of nodes when searching with the original HSD heuristic (HSDH). For example, if the average reduction in search was 70% compared with HSDH (i.e., 70% fewer nodes expanded on average), the fitness score was set to 0.7.

If a given deal was not solved within 2 minutes (a time limit we set empirically), we assigned a fitness score of 0 to that deal.

To distinguish between individuals that did not solve a given deal and individuals that solved it but without reducing the amount of search (the latter case reflecting better performance than the former), we assigned to the latter a partial score of $(1 - \mathit{FractionExcessNodes})/C$, where *FractionExcessNodes* was the fraction of excessive nodes (values greater than 1 were truncated to 1), and C was a constant used to decrease the score relative to search reduction (set empirically to 1000). For example, an excess of 30% would yield a partial score of $(1 - 0.3)/C$; an excess of over 200% would yield 0.

Because of the puzzle's difficulty, some deals were solved by an evolving individual or by HSDH—but not by both, thus rendering comparison (and fitness computation) problematic. To overcome this we imposed a penalty for unsuccessful search: Problems not solved within 2 minutes were counted as requiring 1000M search nodes. For example, if HSDH did not solve within 2 minutes a deal that an evolving individual did solve using 500M nodes, the percent of nodes reduced was computed as 50%. The 1000M value was derived by taking the hardest problem solved by HSDH and multiplying by two the number of nodes required to solve it.

An evolving solver's fitness per single deal, f_i, thus equaled:

$$f_i = \begin{cases} \textit{search-node reduction ratio} & \\ \qquad\qquad \text{if solution found with node reduction} & \\ \max\{\textit{(1-FractionExcessNodes)/1000, 0}\} & \\ \qquad\qquad \text{if solution found without node reduction} & \\ \textit{0} \qquad\quad \text{if no solution found} & \end{cases}$$

and the total fitness, f_s, was defined as the average, $f_s = 1/N \sum_{i=1}^{N} f_i$. Initially we computed fitness by using a constant number, N, of deals (set to 10 to allow diversity while avoiding prolonged evaluations), which were chosen randomly from the training set. However, as the test set was large, fitness scores fluctuated wildly and improvement proved difficult. To overcome this problem we turned to coevolution.

11.2.4 Hillis-style coevolution

We used Hillis-style coevolution wherein a population of solutions coevolves alongside a population of problems (Hillis [89]). The basic idea is that neither population is allowed to stagnate: As solvers become more adept at solving certain problems these latter do not remain in the problem set (as with a simple GA), but are rather removed from the population of problems—which itself evolves. In this form of competitive coevolution the fitness of one population is inversely related to the fitness of the other population.

In our coevolutionary scenario the first population comprised the solvers, as described above. In the second population an individual represented a *set* of FreeCell deals. Thus a "hard"-to-solve individual in this latter, problem population would contain various deals of varying difficulty levels. This multi-deal individual made life harder for the evolving solvers: They had to maintain a consistent level of play over several deals. With single-deal individuals, which we initially experimented with, either the solvers did not improve if the deal population evolved every generation (i.e., too fast), or the solvers became adept at solving certain deals and failed on others if the deal population evolved more slowly (i.e., every k generations, for a given $k > 1$).

The genome and genetic operators of the solver population were identical to those defined above.

The genome of an individual in the deal population contained 6 FreeCell deals, represented as integer-valued indexes from the training set $\{v_1, v_2, \ldots, v_{1000}\}$, where v_i is a random index in the range $[1, 32000]$. We applied GP-style evolution in the sense that first an operator (reproduction, crossover, or mutation) was selected with a given probability, and then one or two individuals were selected in accordance with the operator chosen. We used standard fitness-proportionate selection and single-point crossover. Mutation was performed in a manner analogous to bitwise mutation by replacing with independent probability 0.1 an (integer-valued) index with a randomly chosen deal (index) from the training set, i.e., $\{v_1, v_2, \ldots, v_{1000}\}$ (Figure 11.2). Since the solvers needed more time to adapt to deals, we evolved the deal population every 5 solver generations (this slower evolutionary rate was set empirically).

We experimented with several parameter settings, finally settling on: population size—between 40 and 60, generation count—between 60 and 80, reproduction probability—0.2, crossover probability—0.7, mutation probability—0.1, and elitism set size—1.

Fitness was assigned to a solver by picking 2 individuals in the deal population and attempting to solve all 12 deals they represented. The fitness value was an average of all 12 deals, as described in Section 11.2.3.

Whenever a solver "ran" a deal individual's 6 deals its performance was maintained in order to derive the fitness of the deal population. A deal individual's fitness was defined as the average number of nodes needed to solve the 6 deals, averaged over the solvers that "ran" this individual, and divided by the average number of nodes when searching with the original HSD heuristic. If a particular deal was not solved by any of the solvers—a value of 1000M nodes was assigned to it.

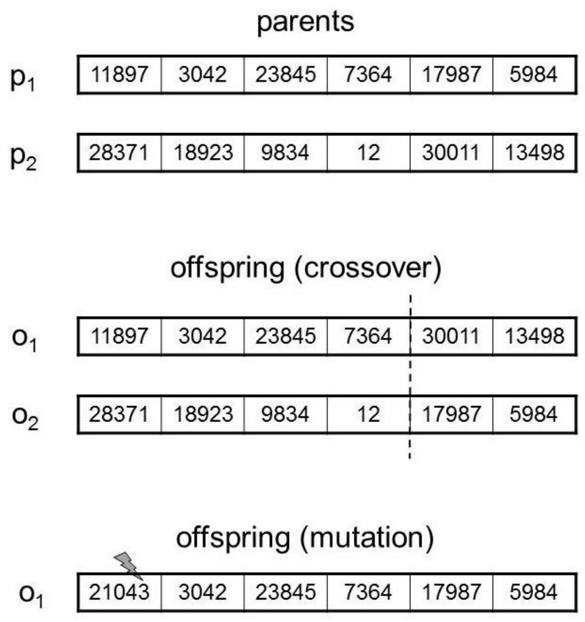


Figure 11.2 Crossover and mutation of individuals in the population of problems (deals).

11.3 Results

We evaluated the performance of evolved heuristics with the same scoring method used for fitness computation, except we averaged over all Microsoft 32K deals instead of over the training set. We also measured average improvement in time, solution length (number of nodes along the path to the correct solution found), and number of solved instances of Microsoft 32K, all compared to the HSD heuristic, HSDH.

The results for the test set (Microsoft 32K minus 1K training set) and for the entire Microsoft 32K set were very similar, and therefore we report only the latter. The runs proved quite similar in their results, with the number of generations being 150 on average. The first few generations took more than 8 hours since most of the solvers did not solve most of the deals within the 2-minute time limit. As evolution progressed a generation came to take less than an hour.

We compared the following heuristics: `HSDH` (Section 11.2.2), `HighestHomeCard` and `DifferenceHome` (Section 11.2.2)—both of which proliferated in evolved individuals, and GA-FreeCell—the top evolved individual.

Table 11.2 shows our results. The `HighestHomeCard` and `DifferenceHome` heuristics proved worse than HSD's heuristic function in all of the measures and therefore were not included in the tables. For comparing unsolved deals we applied the 1000M penalty scheme described in Section 11.2.3 to the node reduction measure. Since we also compared time to solve and solution length, we applied the penalties of 9,000 seconds and 60,000 moves to these measures, respectively.

GA-FreeCell reduced the amount of search by 87%, solution time by 93%, and solution length by 41%, compared to HSDH. In addition, GA-FreeCell solved 98%

Table 11.2 Average number of nodes, time (in seconds), and solution length required to solve all Microsoft 32K problems, along with the number of problems solved. Two sets of measures are given: 1) unsolved problems are assigned a penalty, and 2) unsolved problems are excluded from the count. HSDH is the heuristic function used by HSD. GA-FreeCell is our top evolved solver.

Heuristic	Nodes	Time	Length	Solved
unsolved problems penalized				
HSDH	75,713,179	709	4,680	30,859
GA-FreeCell	**16,626,567**	**150**	**1,132**	**31,475**
unsolved problems excluded				
HSDH	1,780,216	44.45	255	30,859
GA-FreeCell	**230,345**	**2.95**	**151**	**31,475**

of Microsoft 32K, thus outperforming HSDH, the (now) previous top solver, which solved only 96% of Microsoft 32K. Note that although GA-FreeCell solves “only” 2% more instances, these 2% are far harder to solve due to the long tail of the learning curve.

One of our best solvers is the following:

```
(+ (* NumCardsNotAtFoundations 0.09) (* HSDH 0.01) (* FreeCells 0.0) (*
DifferenceFromTop 0.77) (* LowestHomeCard 0.01) (* HighestHomeCard 0.08)
(* NumberWellPlaced 0.01) (* DifferenceHome 0.01) (* SumOfBottomCards
0.02)).
```

(In other good solvers DifferenceFromTop was less weighty.)

How does our evolution-produced player fare against humans? A major FreeCell website[3] provides a ranking of human FreeCell players, listing solution times and win rates (alas, no data on number of deals examined by humans, nor on solution lengths). This site contains thousands of entries and has been active since 1996, so the data is reliable. It should be noted that the game engine used by this site generates random deals in a somewhat different manner than the one used to generate Microsoft 32K. Yet, since the deals are randomly generated, it is reasonable to assume that the deals are not biased in any way. Since statistics regarding players who played sparsely are not reliable, we focused on humans who played over 30K games—a figure commensurate with our own.

The site statistics, which we downloaded on April 12, 2011, included results for 76 humans who met the minimal-game requirement—all but two of whom exhibited a win rate greater than 91%. Sorted according to number of games played, the no. 1 player played 147,219 games, achieving a win rate of 97.61%. This human is therefore pushed to the second position, with our top player (98.36% win rate) taking the first place (Table 11.3). If the statistics are sorted according to win rate then our player assumes the no. 9 position. Either way, it is clear that when compared with strong, persistent, and consistent humans GA-FreeCell emerges as a highly competi-

[3]`http://www.freecell.net`

Table 11.3 The top three human players (when sorted according to number of games played), compared with HSDH and GA-FreeCell. Shown are number of deals played, average time (in seconds) to solve, and percent of solved deals from Microsoft 32K.

Name	Deals played	Time	Solved
sugar357	147,219	241	97.61%
volwin	146,380	190	96.00%
caralina	146,224	68	66.40%
HSDH	32,000	44	96.43%
GA-FreeCell	32,000	**3**	**98.36%**

tive player.

Having won a Gold HUMIE for the results presented above, we asked ourselves whether they could be improved yet further by chucking the simple GA in favor of a more sophisticated evolutionary algorithm, specifically, the policy-based GP of the previous chapter. Once again, we added advisors (or auxiliary functions), incorporating domain features, to the heuristics of Table 11.1, i.e., functions that do not provide an estimate of the distance to the goal but which are nonetheless beneficial in a GP setting (Elyasaf et al. [57]):

`PhaseByX`: This is a set of functions that includes a "mirror" function for each of the heuristics in Table 11.1. Each function's name (and purpose) is derived by replacing `X` in `PhaseByX` with the original heuristic's name, e.g., `LowestHomeCard` produces `PhaseByLowestHomeCard`. `PhaseByX` incorporates the notion of applying different strategies (embodied as heuristics) at different *phases* of the game, with a phase defined by $g/(g+h)$, where g is the number of moves made so far, and h is the value of the original heuristic.

For example, suppose 10 moves have been made ($g = 10$), and the value returned by `LowestHomeCard` is 5. The `PhaseByLowestHomeCard` heuristic will return $10/(10+5)$ or $2/3$ in this case, a value that represents the belief that by using this heuristic the configuration being examined is at approximately $2/3$ of the way from the initial state to the goal.

`DifficultyLevel`: This function returns the location of the current problem being solved in the ordered problem set, and thus yields an estimate of how difficult it is.

`IsMoveToCascade` is a Boolean function that examines the destination of the last move and returns true if it was a cascade.

Table 11.4 provides a list of the auxiliary functions, including the above functions and a number of additional ones.

As in the previous chapter we used 5 rules plus the default rule. The *Condition* part of each rule was represented by a GP tree, whose function set included $\{AND, OR, \leq, \geq\}$, and whose terminal set included all heuristics and auxiliary functions in Tables 11.1 and 11.4. The *Value* part of each rule was represented by a linear

Table 11.4 List of auxiliary functions. B: Boolean, R: Real or Integer.

Node name	Type	Return value
`IsMoveToFreecell`	B	True if last move was to a FreeCell, false otherwise
`IsMoveToCascade`	B	True if last move was to a cascade, false otherwise
`IsMoveToFoundation`	B	True if last move was to a foundation pile, false otherwise
`IsMoveToSortedPile`	B	True if last move was to a sorted pile, false otherwise
`LastCardMoved`	R	Value of last card moved
`NumberOfSiblings`	R	Number of reachable states (in one move) from last state
`NumberOfChildren`	R	Number of reachable states (in one move) from current state
`DifficultyLevel`	R	Index of the current problem in the problem set (sorted by difficulty)
`PhaseByX`	R	"Mirror" function for each heuristic
`g`	R	Number of moves made from initial configuration to current

vector of 9 weights—one per heuristic—in the range $[0, 1]$ (as opposed to the previous chapter where *Value*s were represented as trees). The final heuristic value (and the normalization of each single heuristic value) was computed as with the simple GA (Section 11.2.3).

Using policy-based GP we were able to make a substantive improvement in the number of problems solved, going from 98.36% to 99.65%. A fuller account of GP-FreeCell is currently being written up (Elyasaf et al. [57]). We have also obtained initial promising results on the sliding-tile puzzle (Elyasaf et al. [58]).

11.4 Discussion

We evolved a solver for the FreeCell puzzle, one of the most difficult single-player domains (if not the most difficult) to which evolutionary algorithms have been applied to date. GA-FreeCell beats the previous top published solver by a wide margin on several measures (and GP-FreeCell goes even further).

There are a number of possible extensions to this work. The HSD algorithm, enhanced with evolved heuristics, is more efficient than the original version. This is evidenced both by the amount of search reduction and the increased number of solved deals. It remains to be determined whether the algorithm, when aided by evolution, can outperform other widely used algorithms (such as IDA*) in different domains. The fact that the algorithm is based on several properties of search problems, such as the high percentage of irreversible moves along with the small number of deadlocks, already points the way towards several domains. A good candidate may be the Satellite game, previously studied by Haslum et al. [76] and Helmert [88].

Again, as noted in the previous chapter, handcrafted heuristics may themselves be improved by evolution. This could be done by breaking them into their elemental components and evolving their combinations thereof.

Part V

Parting Words of…

"Why do you not solve it yourself, Mycroft? You can see as far as I."

"Possibly, Sherlock… No, you are the one man who can clear the matter up. If you have a fancy to see your name in the next honours list –"

My friend smiled and shook his head.

"I play the game for the game's own sake," said he.

—Arthur Conan Doyle
"The Adventure of the Bruce-Partington Plans"

Chapter 12

. . . (Putative) Wisdom

And so we come to the final chapter of this book. After witnessing the many successful applications of evolutionary algorithms—mostly genetic programming—to several games of diverse nature, it's time to draw some general conclusions and also offer a tip or eight for the evolutionary computation practitioner wishing to tackle games.

12.1 Attribute 17

In their book, Koza et al. [110] delineated 16 attributes a system for automatically creating computer programs might beneficially possess:

1. Starts with problem requirements.
2. Produces tractable and viable solution to problem.
3. Produces an executable computer program.
4. Automatic determination of program size.
5. Code reuse.
6. Parameterized reuse.
7. Internal storage.
8. Iterations, loops, and recursions.
9. The ability to organize chunks of code into hierarchies.
10. Automatic determination of program architecture.
11. Ability to implement a wide range of programming constructs.

12. Operates in a well-defined manner.
13. Problem-independent, i.e., possesses some degree of generalization capabilities.
14. Applicable to a wide variety of problems from different domains.
15. Able to scale well to larger instances of a given problem.
16. Competitive with human-produced results.

Our own work with GP has prompted us to suggest an additional attribute to this list (Sipper et al. [169]):

17. Cooperative with humans.

We believe that a major reason for our success in evolving winning game strategies is GP's ability to readily accommodate human expertise in the *language of design.* My colleagues and I defined this latter term within the framework of our proposed *emergence test* (Ronald et al. [148]). The test involves two separate languages—one used to *design* a system, the other used to describe *observations* of its (putative) emergent behavior. The effect of surprise arising from the gap between design and observation is at the heart of the emergence test. Our languages of design for the games in this book possess several functions and terminals that attest to the presence of a (self-proclaimed) intelligent designer. These design languages, which give rise to powerful languages of observation in the form of successful players, were designed not instantaneously—like Athena springing from Zeus's head fully grown—but rather through an incremental, interactive process, whereby man (represented by the humble authors of these works) and machine (represented by man's computers) worked hand-in-keyboard. To wit, we would begin our experimentation with small sets of functions and terminals, which would then be revised and added upon through our examination of evolved players and their performance. Figure 12.1 describes three major steps in our hand-in-keyboard development of the evolutionary chess setup of Chapter 4.

We believe that GP represents a viable means to automatic programming, and perhaps more generally to machine intelligence, in no small part due to attribute 17: more than many other adaptive-search techniques (such as artificial neural networks and ant algorithms), GP's representational affluence and openness lend it to the ready imbuing of the designer's own intelligence within the language of design. While AI purists may wrinkle their noses at this, taking the AI-should-emerge-from-scratch stance, we argue that a more practical path to AI involves man-machine cooperation. GP, as evidenced herein, is a forerunning candidate for the "machine" part.

This brings up a related issue, derived from Koza et al.'s affirmation that, "Genetic programming now routinely delivers high-return human-competitive machine intelligence" (Koza et al. [111]). They define high-return as a high "artificial-to-intelligence ratio" (*A/I*), namely, the ratio of that which is delivered by the automated operation

1. Initial runs: No distinction between "good" and "bad" terminals (i.e., no negative terminals), e.g., `IsMyKingInCheck` and `IsOppKingInCheck` (later the former will become `NotMyKingInCheck`).
Terminals:

- `Is[My/Opp]PieceAttacked`
- `MaterialCount`
- `NumMoves[My/Opp]King`
- `Is[My/Opp]PieceAttacked`
- `Is[My/Opp]PieceProtected`
- `Is[My/Opp]QueenAttacked`
- `IsMate`
- `ERCs` in range [-1000,+1000]

Functions:

- Arithmetic: `*`, `+`, `-`
- Logic: `And2`, `And3`, `And4`, `Or2`, `Or3`, `Or4`, `Not`
- Others: `If`, <, =, >

2. Later runs: We consulted a chess Master.
Terminals:

- Modified to distinguish between positive and negative, e.g., `NotMyKingInCheck` and `MyKingDistEdges`
- Added `IsMaterialIncrease`
- Added `Not[My/Opp]KingMovesDecrease`
- Added `Num[My/Opp]PiecesNotAttacked`
- Added `IsMyKingProtectingPiece`
- Added `IsMyPieceAttackedUnprotected`
- Added `IsOppKingBehingPiece`
- Added `IsStalemate`

Functions:

- Removed arithmetic functions (see why in Chapter 4) except for Negate
- Removed ">" to simplify computation
- Used `IfAdvantageThen[Left Subtree]Else[Right Subtree]` to create separate calculations

3. Final runs: We further consulted a Master, adding complex and simple terminals.
Terminals:

- Added `MateInOne`
- Added `IsOppKingStuck`
- Added `OppPieceCanBeCaptured`
- `IsMaterialIncrease` changed to 100*`IsMaterialIncrease`
- Added `ValueOf[My/Opp][Attacking/Protecting]Pieces`
- Added `Is[My/Opp][Not]Fork`
- Added `[My/Opp]King[Dist/Prox]Rook`
- Added `[My/Opp]Pieces[Not]SameLine`
- `Num[My/Opp]Pieces[Not]Attacked`
- `ERCs`: Now only six values allowed, $\pm$ {0.25, 0.5, 1}*1000

Functions:

- Removed Negate
- Changed program topology to three trees

Figure 12.1 Three major steps in developing the evolutionary chess setup (Sipper et al. [169]).

of the artificial method to the amount of intelligence that is supplied by the human applying the method to a particular system.

Our discussion regarding attribute 17 stands in contrast to the property of high-return, which we believe to be of little import in the domain of human-competitive machines, and indeed, in the attainment of machine intelligence in general. Rather than aiming to maximize *A/I* we believe the "correct" equation is:

$$A - I \geq M_\epsilon,$$

where M_ϵ stands for "meaningful epsilon". When wishing to attain machine competence in some real-life, hard-to-learn domain, then—by all means—imbue the machine with as much *I*(ntelligence) as possible! After all, if imbuing the *I* reduces the problem's complexity to triviality, then it was probably not hard to begin with. Conversely, if the problem is truly hard, then have man and machine work in concert to push the frontiers of *A* as far as possible. Thus, it is not $\max(A/I)$ that is of interest but the added value of the machine's output: Granting the designer "permission" to imbue the machine with as much *I* as he can, will it then produce a $\Delta A = A - I$, namely, added intelligence, that is sufficiently meaningful? Even if this meaningful epsilon (M_ϵ) is small in (some) absolute terms, its relative value can be huge (e.g., a chip that can pack 1-2% more transistors, or a game player that is slightly better—and thus world champion).

One problem with the $\max(A/I)$ view is its ignoring the important distinction between two phases of intelligence (or knowledge) development: 1) from scratch to a mediocre level, and 2) from mediocre to expert level. Traditional AI is often better at handling the first phase. GP allows the AIer to focus his attention on the second phase, namely, the attainment of true expertise. When aiming to develop a winning strategy, be it in games or any other domain, the GP practitioner will set his sights at the mediocre-to-expert phase of development, with the scratch-to-mediocre handled automatically during the initial generations of the evolutionary process. Although the designer is "imposing" his own views on the machine, this affords the "pushing" of the *A* frontier further out. Note that, at the limit, if *I* tends to zero, you may get an extremely high A/I ratio, but with very little truly meaningful *A*. Focusing on $A - I$ underscores the need, or wish, for a high level of intelligence, where even a small M_ϵ becomes important.

Cognitive psychology recognizes the importance of *schemata*, a fundamental notion first defined by Bartlett in his influential book from 1932 (Bartlett [14]). Schemata are mental patterns or models that give rise to certain cognitive abilities—complex unconscious knowledge structures such as symmetry in vision, plans in a story, and rules. Much of our knowledge is encoded as schemata, to be neurally activated when their components are triggered in a certain way (only a certain configuration of face parts will activate the "face" schema). GP is able to go beyond low-level "bits-and-pieces" knowledge and handle what may well be schemata analogs. In our treatment of games we were able to encode meaningful patterns (schemata) as terminals, then combined through the use of functions. This adds a whole new dimension to the representations one can design.

As an example, a chess master's knowledge seems to comprise some 100,000 schemata (Simon and Gilmartin [163]), and his advantage over the machine lies in his ability to combine these schemata intelligently in response to a given situation. It is not impossible to imagine programming 100,000 chess features when striving to grant your machine as much *I* as possible; but finding and applying the correct combinations is exponential in nature. Here GP steps in, constantly trying new combinations and combinations of combinations, beyond that which is possible to accomplish by (traditional) AI (artificial neural networks, for example, also traverse the search space but they lack the ability to integrate deep knowledge in a natural manner).

GP is able to combine search with pattern recognition, as is true of humans, with the terminals acting as pattern recognizers (e.g., safety of king, mate in one). Chess players, for example, seem to make extensive use of patterns, or templates (Gobet and Simon [71]). Patterns are a powerful addition to the toolbox of the machine's *A*, enabling it to make use of an important element of *I*.

An early, well-known AI program—Newell's and Simon's General Problem Solver (GPS) (Newell and Simon [127])—ultimately proved to be far from general and quite limited in scope. As opposed to their GPS we do not advocate complete generality and—more importantly—neither do we promote total machine autonomy. We believe our approach represents a more practical means of attaining machine expertise—at least at the current state of AI—and suggest replacement of the original GPS with a more humble one: Genetic Programming Solver.

In short, $GP + I \Rightarrow HC$, i.e., Genetic Programming + (Human) Intelligence yields Human-Competitiveness.

12.2 Tips for the Game Evolver

It is customary to tip before departing and far be it from me to forgo this time-honored tradition. I have attempted to distill our group's experience in game research into a number of tips for the evolutionary computation practitioner wishing to tackle games.

- **Know thy game.** As I've discussed above, a main advantage of GP is its cooperation with the human GPer. My experience has been that insights into the intricacies of the game at hand can—and should—be embodied as domain knowledge supplied to the evolutionary system.

This does not mean you should be a top-notch player of the game—but you *should* be a player, or at least be familiar with the game. Where hard problems are concerned, the more you imbue the system with domain knowledge the better. Given the right ingredients—GP will soar by evolutionarily forming winning composites.

- **Something borrowed, something new.** There will be cases where you don't know your game well enough because the game is hard, new, has not received much attention, or any combination of these reasons. Just as any other AI researcher, the game researcher too may sometimes find himself facing a challenge for which the tools are insufficient. As an example, for lose checkers (Chapter 3) the little previous work that existed taught us little about board-state evaluation for this game.

  All is not lost. Find the principles that are important for similar games—mobility, for instance—and let evolution try them out. They may work for you. If they don't, find others.

- **Fitness, fitness, fitness.** Wonderful ingredients alone will not suffice if the fitness function is ill-conceived. Should you use an external opponent? Several external opponents? Coevolution? A database of hard game configurations? Create random game configurations?

  In Chapter 7 we noted how evolved backgammon players performed better when trained against their own kind (coevolution), compared with being trained by an external opponent. For Robocode (Chapter 8) just the opposite was true—top players were evolved by pitting them against external opponents (note the plural here: we did not use a single opponent for fitness evaluation but rather several, to promote diversity). The game of FreeCell (Chapter 11) proved a hard nut to crack, and we went through several attempts at fitness definitions before we hit upon the successful blend of coevolution.

- **Beware the demon of overfitting.** The simplest and perhaps most straightforward approach to examining an individual's quality is that of defining a static fitness evaluation function (i.e., every individual in every generation has its fitness evaluated in precisely the same manner). In adversarial games this kind of thinking most naturally leads to fitness evaluation via playing against a single external opponent. In our experience this can often lead one astray. In many of the cases studied herein, especially board games, fitness evaluation based on games against other members of the population (coevolution) proved superior to (external) guide play.

  The culprit here is a well-known one: overfitting. When an individual's fitness is computed through competition against a single, constant external opponent you run the risk of learning the opponent rather than the game. This seems to hold true especially for turn-based, full-knowledge, zero-sum combinatorial board games. Evolving a good player for these games requires noise, which can be had through the use of a dynamic, coevolutionary fitness function.

- **Competition is good.** The training set used by an evolutionary algorithm (or, for that matter, any learning process) should include a good dose of diversity. This helps the learners face a wide variety of situations, which increases their ability to generalize to unseen situations once the learning process has finished. Ideally, at each generation, each individual should be trained using the entire training set at our disposal. Unfortunately, this evaluation process is often very time-consuming and it is only possible to use a small number of training samples. For example, in Chapters 10 and 11 we used only a dozen or so problems to compute the fitness of a single individual. No matter how you select this small subset the results over the entire set will be poor. Moreover, if you inject randomness into the training set, replacing some elements every so often, your individuals may well end up solving only problems from the last few generations (a phenomenon we observed with FreeCell).

  We overcame this obstacle in Chapter 11 by using Hillis-style coevolution, wherein a population of solvers coevolves alongside a population of problems, not affording the two populations an opportunity to stagnate. Having concluded that there really is no sense in speaking of *a* hard problem in this case, but only of a set of hard problems, we used multi-problem individuals in the problems population. This proved highly efficacious, constantly "forcing" the evolving solvers to solve several types of problems, thus inducing them toward generality.

- **Don't be embarrassed about parallelism.** Evolutionary algorithms have been referred to as "embarrassingly parallel", given their inherent parallel nature, due, essentially, to the existence of a *population* of struggling individuals. Don't be shy about throwing the cluster at your system. We have often used several computers plodding through the lone nights. In Chapter 7, for example, we noted explicitly how the move from a single computer to a cluster dramatically improved our backgammon players.

- **The bane of graphics.** Simulation games often possess—ipso facto—complex graphic engines. This is wonderful from the user's perspective, but can be quite bothersome when your aim is evolution. While bells and whistles enhance the human experience, they slow down evolution—often to a grind. When running an evolutionary algorithm your aim is to use the simulator to evaluate many individuals, over many generations, with fitness values ascribed on the basis of statistics gathered during the simulation (energy used, tanks destroyed, distance race car has covered). The one thing you most emphatically do *not* need during fitness evaluation is nice graphics. Beware that decoupling the graphics component from a simulator is often a very unpleasant task, at times even a Herculean effort. Most simulators were simply not written with evolutionary computation in mind.

We actually abandoned the RARS work (Chapter 9) at some point, after a huge amount of effort invested in trying to get those darn cars to race toward the finish line. Several months went by and we decided to return to that conundrum, an act which Leopold von Sacher-Masoch[1] might have appreciated. After much additional work we discovered a highly intricate bug in our graphics-decoupling code, stemming from some weird coding in the simulator. Having made that discovery, the rest of the (research) journey became merely uphill rather than up Mount Everest.

Which, come to think of it, naturally segues into my last tip.

- **If at first you don't succeed, try, try, try again.** A trite cliché if there ever was one—but tried and true, nonetheless. None of the results presented here leaped out singing from our very first evolutionary setup. Nor the second or third, for that matter. . .

[1]Who "donated" his name to form the term *masochism*.

Appendix: A Potpourri of Games

Over the years I have supervised dozens of undergraduate projects in the area of games. The works described in Chapters 8 (Robocode) and 10 (Rush Hour) began as small undergraduate projects, only blossoming into full-blown research agendas as the respective students entered the graduate world. I believe it might be of interest to list some of the games tackled (see also Kendall et al. [102] for a good survey of hard puzzles). In general, as we saw throughout this book, one can apply evolution in a number of ways: evolve solutions to particular game configurations, evolve solvers for any game configuration, evolve game-playing agents, and more.

Most of the games herein have been little explored by evolutionary computation researchers, though some of the more popular ones have been investigated to a certain extent (e.g., Mastermind—Berghman et al. [19], Sudoku—Mantere and Koljonen [120], and Rubik's Cube—El-Sourani et al. [54]).[2]

- **Battleship** is a guessing game played by two people. It is known throughout the world as a pencil and paper game and predates World War I in this form. The game is played on four grids, two for each player. On one grid the player arranges ships and records the shots by the opponent. On the other grid the player records his own shots. The objective is to sink the opponent's ships by calling out their positions.

- **Bloxorz** is a game whose aim is to get a block to fall into the square hole at the end of each stage. To move the block one uses the left, right, up and down arrow keys and must be careful not to fall off the edges.[3]

- **Connect Four** is a two-player game in which the players first choose a color and then take turns dropping their colored discs from the top into a seven-column, six-row, vertically suspended grid. The pieces fall straight down, occupying the next available space within the column. The object of the game is to connect four of one's own discs of the same color next to each other vertically, horizontally, or diagonally before one's opponent can do so.

[2]Incidentally, the word *potpourri*, meaning a miscellaneous collection, derives its meaning from the French *pot pourri*, literally, rotten pot. Hopefully, the potpourri of games presented herein will *not* be regarded as rotten...

[3]`http://www.coolmath-games.com/0-bloxorz/index.html`

- **Corral Puzzles** consist of a grid of squares, some of which contain numbers. The goal is to find a closed loop containing some of the grid squares so that all the numbers are inside the loop and each number is equal to the number of grid squares visible from it.[4]

- **enDice** consists of several dice arranged in a two-dimensional space. The objective is to get all of the dice into the dotted areas with no moves remaining. Each die has a number representing the number of spaces it can move. A die can also push other dice without using their moves.[5]

- **Eternity II** is a 256-piece combinatorial puzzle, which has not been solved to date.[6]

- **FreeNet** is played on a rectangular grid, the objective being to turn blocks with wires and devices so as to connect all devices to electricity.[7]

- **Instant Insanity** is a puzzle consisting of four cubes with faces colored with four colors. The object of the puzzle is to stack these cubes in a column so that each side (front, back, left, and right) of the stack shows each of the four colors. The distribution of colors on each cube is unique. The generalized version of the game includes n cubes with n colors.[8]

- **Kakuro** in its canonical form is played on a grid of filled and barred cells, "black" and "white", respectively. The objective of the puzzle is to insert a digit from 1 to 9 inclusive into each white cell such that the sum of the numbers in each row and column equals a given clue.

- **KPlumber**, or **Linkz**, is a puzzle whose objective is to connect all the open ends of the pipes, electronic circuits, roads, or whatever tiles are used, together so that they form continuous shapes.[9]

- **Lights Out** consists of a rectangular grid of lights. When the game starts, a random number or a stored pattern of these lights is switched on. Pressing any of the lights will toggle it and the four adjacent lights. The goal of the puzzle is to switch all the lights off, preferably in as few button presses as possible.[10]

- **Light Up** is played on a rectangular grid made up of black and white cells, the objective being to place light bulbs on the grid so that every white square is lit.

[4] `http://www2.stetson.edu/~efriedma/papers/corral/corral.html`
[5] `http://armorgames.com/play/2759/endice`
[6] `http://www.eternityii.com`
[7] `http://www.jurjans.lv/stuff/net/FreeNet.htm`
[8] `http://en.wikipedia.org/wiki/Instant_Insanity`
[9] `http://www.vanderlee.com/linkz/index.html`
[10] `http://www.genuine-lights-out.com`

A cell is illuminated by a light bulb if both are in the same row or column, and if there are no black cells between them. Also, no light bulb may illuminate another light bulb.[11]

- **Mahjong solitaire**, also known as **Shanghai solitaire**, is a solitaire matching game that uses a set of Mahjong tiles rather than cards. The tiles come from the four-player game popular in East Asia known as Mahjong.
- **Mancala** is a family of board games played around the world, sharing a common general game play. Players begin by placing a certain number of seeds in each of the pits on the game board. A turn consists of removing all seeds from a pit, sowing the seeds (placing one in each of the following pits in sequence), and capturing based on the state of the board.
- **Mastermind** is a classic, two-player game where the objective is for one player (the codebreaker) to break the other player's (the codemaker's) code. The code is a sequence of colors or digits. The modern game with pegs was invented in 1970 by Mordecai Meirowitz, an Israeli postmaster and telecommunications expert.
- **Monopoly** is a classic board game where one buys property and attempts not to go bankrupt.
- **Nonogram** is a picture logic puzzle in which cells in a grid have to be colored or left blank according to numbers given at the side of the grid to reveal a hidden picture.
- **Nurikabe** is played on a typically rectangular grid of cells, some of which contain numbers. Cells are initially of unknown color, but can only be black or white. The challenge is to paint each cell black or white, subject to a number of rules and constraints.[12]
- **Pac-Man** is a classic game where the player controls Pac-Man through a maze, eating pac-dots. When all dots have been eaten, Pac-Man is taken to the next stage.
- **Peg solitaire** is a board game for one player, involving movement of pegs on a board with holes. The standard game fills the entire board with pegs except for the central hole. The objective is, making valid moves, to empty the entire board except for a solitary peg in the central hole.
- **Poker** is a highly challenging and complex game, which has been tackled by AI practitioners in recent years.[13]

[11]`http://www.puzzle-light-up.com`
[12]`http://www.puzzle-nurikabe.com`
[13]See, for example, the University of Alberta's poker page: `http://poker.cs.ualberta.ca`.

- **River Crossing** presents the challenge of crossing a crocodile-infested river by walking over wooden planks supported by tree stumps. You can pick up planks and put them down between stumps, as long as they are exactly the right distance apart.[14]

- **Rummikub** is a tile-based game for two to four players invented in Israel. There are 104 number tiles and two or more jokers. Players continually meld their tiles into legal combinations until one player has used all of the tiles in his rack.[15]

- **Rubik's Cube** is a cube, each of whose six faces is subdivided into nine cells, each colored by one of six colors. A pivot mechanism enables each face to turn independently, thus mixing up the colors. For the puzzle to be solved, each face must be a solid color.

- **Go** might well be the most challenging board game of all, and machine players are still far below the level of human masters. This game has been receiving increased attention from AIers.

- **Sudoku**'s objective is to fill a 9 by 9 grid with digits so that each column, each row, and each of the nine 3 by 3 sub-grids that compose the grid contains all of the digits from 1 to 9 (other square grid sizes are also used, but 9 by 9 is by far the most popular size). A puzzle instance comprises a partially completed grid, which typically has a unique solution. Several variants of this classic version of the game are now in existence.[16]

- **WordIt**'s objective is to arrange all the letters on a board to form valid words. The player can reuse letters and make words overlap to increase the score.[17]

[14]`http://www.clickmazes.com/planks/ixplanks.htm`
[15]`http://www.rummikub.com`
[16]`http://en.wikipedia.org/wiki/Sudoku`
[17]`http://www.freeonlinegames.com/game/wordit.html`

Bibliography

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.

[2] R. Aler, D. Borrajo, and P. Isasi. Evolving heuristics for planning. *Lecture Notes in Computer Science*, 1447:745–754, 1998.

[3] R. Aler, D. Borrajo, and P. Isasi. Learning to solve planning problems efficiently by means of genetic programming. *Evolutionary Computation*, 9(4):387–420, Winter 2001.

[4] R. Aler, D. Borrajo, and P. Isasi. Using genetic programming to learn and improve knowledge. *Artificial Intelligence*, 141(1–2):29–56, 2002.

[5] L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-number search. *Artificial Intelligence*, 66:91–124, 1994.

[6] J. R. Anderson and C. Lebiere. *The Atomic Components of Thought*. Lawrence Erlbaum Associates, Mahwah, NJ, 1998.

[7] P. J. Angeline and J. B. Pollack. Competitive environments evolve better solutions for complex tasks. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 264–270, 1993.

[8] Y. Azaria and M. Sipper. GP-Gammon: Genetically programming backgammon players. *Genetic Programming and Evolvable Machines*, 6(3):283–300, September 2005.

[9] Y. Azaria and M. Sipper. GP-Gammon: Using genetic programming to evolve backgammon players. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, editors, *Proceedings of 8th European Conference on Genetic Programming (EuroGP2005)*, volume 3447 of *Lecture Notes in Computer Science*, pages 132–142. Springer-Verlag, Heidelberg, 2005.

[10] F. Bacchus. AIPS'00 planning competition. *AI Magazine*, 22(1):47–56, 2001.

[11] M. Bader-El-Den, R. Poli, and S. Fatima. Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework. *Memetic Computing*, 1(3):205–219, November 2009.

[12] M. Bain. *Learning Logical Exceptions in Chess*. PhD thesis, University of Strathclyde, Glasgow, Scotland, 1994.

[13] Y. Bar-Yam. *Dynamics Of Complex Systems (Studies in Nonlinearity)*. Westview Press, July 2003.

[14] F. C. Bartlett. *Remembering: An Experimental and Social Study*. Cambridge University Press, Cambridge, UK, 1932.

[15] E. B. Baum and I. B. Durdanovic. Evolution of cooperative problem solving in an artificial economy. *Neural Computation*, 12:2743–2775, December 2000.

[16] D. F. Beal and M. C. Smith. Multiple probes of transposition tables. *ICCA Journal*, 19(4):227–233, December 1996.

[17] A. Benbassat and M. Sipper. Evolving lose-checkers players using genetic programming. In *IEEE Conference on Computational Intelligence and Games*, pages 30–37. IEEE Press, 2010.

[18] A. Benbassat and M. Sipper. Evolving board-game players with genetic programming. In N. Krasnogor et al., editors, *GECCO '11: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (Workshops)*, pages 739–742, New York, NY, USA, 2011. ACM.

[19] L. Berghman, D. Goossens, and R. Leus. Efficient solutions for Mastermind using genetic algorithms. *Computers & Operations Research*, 36:1880–1885, June 2009.

[20] A. Bernstein and M. de V. Roberts. Computer versus chess-player. *Scientific American*, 198(6):96–105, 1958.

[21] G. Bonanno. The logic of rational play in games of perfect information. Papers 347, California Davis—Institute of Governmental Affairs, 1989.

[22] D. Borrajo and M. M. Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *Artificial Intelligence Review*, 11(1-5):371–405, 1997.

[23] A. Botea, M. Muller, and J. Schaeffer. Using abstraction for planning in Sokoban. In *Proceedings of the 3rd International Conference on Computers and Games (CG'2002)*, pages 360–375. Springer, 2002.

[24] M. S. Bourzutschky, J. A. Tamplin, and G. M. Haworth. Chess endgames: 6-man data and strategy. *Theoretical Computer Science*, 349:140–157, December 2005.

[25] S. Brave. Evolving recursive programs for tree search. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10, pages 203–220. MIT Press, Cambridge, MA, USA, 1996.

[26] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. R. Woodward. A classification of hyper-heuristic approaches. In M. Gendreau and J. Potvin, editors, *Handbook of Meta-Heuristics*, pages 449–468. Springer, 2nd edition, 2010.

[27] M. V. Butz and T. D. Lönneker. Optimized sensory-motor couplings plus strategy extensions for the TORCS car racing challenge. In *CIG'09: Proceedings of the 5th International Conference on Computational Intelligence and Games*, pages 317–324, Piscataway, NJ, USA, 2009. IEEE Press.

[28] M. Campbell, A. J. Hoane, Jr., and F.-H. Hsu. Deep blue. *Artificial Intelligence*, 134(1–2):57–83, 2002.

[29] M. S. Campbell and T. A. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20:347–367, 1983.

[30] L. Cardamone, D. Loiacono, and P. L. Lanzi. On-line neuroevolution applied to the open racing car simulator. In *CEC'09: Proceedings of the Eleventh Congress on Evolutionary Computation*, pages 2622–2629, Piscataway, NJ, USA, 2009. IEEE Press.

[31] L. Cardamone, D. Loiacono, and P. L. Lanzi. Learning drivers for TORCS through imitation using supervised methods. In *CIG'09: Proceedings of the 5th International Conference on Computational Intelligence and Games*, pages 148–155, Piscataway, NJ, USA, 2009. IEEE Press.

[32] C. F. Chabris and E. S. Hearst. Visualization, pattern recognition, and forward search: Effects of playing speed and sight of the position on grandmaster chess errors. *Cognitive Science*, 27:637–648, February 2003.

[33] B. Chaperot. Motocross and artificial neural networks. In *Game Design and Technology Workshop*, 2005.

[34] B. Chaperot and C. Fyfe. Improving artificial intelligence in a motocross game. In *IEEE Symposium on Computational Intelligence and Games (CIG'06)*, pages 181–186, 2006.

[35] W. Chase and H. Simon. Perception in chess. *Cognitive Psychology*, 4:55–81, 1973.

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

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

[38] N. Chomsky. *Language and Thought*. Moyer Bell, Wakefield, RI, 1993.

[39] S. Y. Chong, D. C. Ku, H. S. Lim, M. K. Tan, and J. D. White. Evolved neural networks learning Othello strategies. In *2003 Congress on Evolutionary Computation (CEC '03)*, volume 3, pages 2222–2229. IEEE Press, December 2003.

[40] B. Cleland. Reinforcement learning for racecar control. Master's thesis, The University of Waikato, 2006.

[41] A. Coles and K. A. Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28:119–156, 2007.

[42] S. Colette, J.-F. Raskin, and F. Servais. On the symbolic computation of the hardest configurations of the Rush Hour game. In *Proceedings of the Fifth International Conference on Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 220–233. Springer-Verlag, 2006.

[43] R. Coulom. *Reinforcement Learning using Neural Networks, with Applications to Motor Control*. PhD thesis, Institut National Polytechnique de Grenoble, 2002.

[44] J. C. Culberson and J. Schaeffer. Searching with pattern databases. In *Canadian Conference on AI*, volume 1081 of *Lecture Notes in Computer Science*, pages 402–416. Springer Verlag, 1996.

[45] F. Dahl. *JellyFish Backgammon*, 1998–2004. URL `http://www.jellyfish-backgammon.com`.

[46] P. Darwen. Why co-evolution beats temporal-difference learning at backgammon for a linear architecture, but not a non-linear architecture. In *Proceedings of the 2001 Congress on Evolutionary Computation (CEC-01)*, pages 1003–1010, 2001.

[47] O. David-Tabibi, M. Koppel, and N. S. Netanyahu. Expert-driven genetic algorithms for simulating evaluation functions. *Genetic Programming and Evolvable Machines*, 12(1):5–22, 2011.

[48] R. Dawkins. *The Selfish Gene*. Oxford University Press, Oxford, UK, 1976.

[49] D. DeCoste. The significance of Kasparov vs. Deep Blue and the future of computer chess. *ICCA Journal*, 21(1):33–43, January 1998.

[50] Deep Blue. Deep Blue Versus Kasparov: The Significance for Artificial Intelligence, Collected Papers from the 1997 AAAI Workshop, 1997.

[51] M. Ebner and T. Tiede. Evolving driving controllers using genetic programming. In *CIG'09: Proceedings of the 5th International Conference on Computational Intelligence and Games*, pages 279–286, Piscataway, NJ, USA, 2009. IEEE Press.

[52] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer Verlag, 2nd edition, 2007.

[53] J. Eisenstein. Evolving Robocode tank fighters. Technical Report AIM-2003-023, AI Lab, Massachusetts Institute Of Technology, 2003.

[54] N. El-Sourani, S. Hauke, and M. Borschbach. An evolutionary approach for solving the Rubik's Cube incorporating exact methods. In C. D. Chio et al., editors, *Applications of Evolutionary Computation*, volume 6024 of *Lecture Notes in Computer Science*, pages 80–89. Springer Berlin / Heidelberg, 2010.

[55] D. Eleveld. DougE1, 2003. URL `http://rars.sourceforge.net/selection/douge1.txt`.

[56] A. Elyasaf, A. Hauptman, and M. Sipper. GA-FreeCell: Evolving solvers for the game of FreeCell. In N. Krasnogor et al., editors, *GECCO '11: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, pages 1931–1938, New York, NY, USA, 2011. ACM.

[57] A. Elyasaf, A. Hauptman, and M. Sipper. GP-FreeCell: Evolving solvers for the game of FreeCell using genetic programming, 2011. (in preparation).

[58] A. Elyasaf, Y. Zaritsky, A. Hauptman, and M. Sipper. Evolving solvers for FreeCell and the sliding-tile puzzle. In *SoCS '11: Proceedings of the Fourth Annual Symposium on Combinatorial Search*, 2011.

[59] S. L. Epstein. Game playing: The next moves. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 987–993. AAAI Press, Menlo Park, California USA, 1999.

[60] E. Eskin and E. Siegel. Genetic programming applied to Othello: Introducing students to machine learning research. *SIGCSE Bulletin*, 31:242–246, March 1999.

[61] A. Felner, R. E. Korf, and S. Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[62] H. Fernau, T. Hagerup, N. Nishimura, P. Ragde, and K. Reinhardt. On the parameterized complexity of the generalized Rush Hour puzzle. In *Proceedings of 15th Canadian Conference on Compututational Geometry*, pages 6–9, 2003.

[63] G. J. Ferrer and W. N. Martin. Using genetic programming to evolve board evaluation functions for a board game. In *1995 IEEE Conference on Evolutionary Computation*, volume 2, pages 747–752, Perth, Australia, 1995. IEEE Press.

[64] G. W. Flake and E. B. Baum. Rush Hour is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270 (1-2):895–911, 2002.

[65] D. Floreano, T. Kato, D. Marocco, and E. Sauser. Co-evolution of active vision and feature selection. *Biological Cybernetics*, 90(3):218–228, 2004.

[66] D. Fogel, T. J. Hays, S. Hahn, and J. Quon. A self-learning evolutionary chess program. *Proceedings of the IEEE*, 92(12):1947–1954, December 2004.

[67] A. Fraenkel. Selected bibliography on combinatorial games and some related material. *The Electronic Journal of Combinatorics*, 1:1–33, 1994.

[68] P. W. Frey. *Chess Skill in Man and Machine*. Springer-Verlag, Secaucus, NJ, USA, 1979.

[69] J. Fürnkranz. Machine learning in computer chess: The next generation. *International Computer Chess Association Journal*, 19(3):147–161, September 1996.

[70] A. Glazer and M. Sipper. Evolving an automatic defect classification tool. In M. Giacobini et al., editors, *Applications of Evolutionary Computing: Proceedings of EvoWorkshops 2008*, volume 4974 of *Lecture Notes in Computer Science*, pages 194–203. Springer-Verlag, Heidelberg, 2008.

[71] F. Gobet and H. A. Simon. Templates in chess memory: A mechanism for recalling several boards. *Cognitive Psychology*, 31:1–40, 1996.

[72] R. Gross, K. Albrecht, W. Kantschik, and W. Banzhaf. Evolving chess playing programs. In W. B. Langdon et al., editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 740–747, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[73] O. Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3):207–227, September 1992.

[74] R. Harper. Co-evolving Robocode tanks. In N. Krasnogor et al., editors, *GECCO '11: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, pages 1443–1450, New York, NY, USA, 2011. ACM.

[75] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.

[76] P. Haslum, B. Bonet, and H. Geffner. New admissible heuristics for domain-independent planning. In M. M. Veloso and S. Kambhampati, editors, *AAAI '05: Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, pages 1163–1168. AAAI Press / MIT Press, 2005.

[77] A. Hauptman. *Evolving Search Heuristics for Combinatorial Games with Genetic Programming*. PhD thesis, Ben-Gurion Unversity, Beer-Sheva, Israel, 2009.

[78] A. Hauptman and M. Sipper. Analyzing the intelligence of a genetically programmed chess player. In *Late Breaking Papers at the 2005 Genetic and Evolutionary Computation Conference*, Washington DC, June 2005.

[79] A. Hauptman and M. Sipper. GP-EndChess: Using genetic programming to evolve chess endgame players. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, editors, *Proceedings of 8th European Conference on Genetic Programming (EuroGP2005)*, volume 3447 of *Lecture Notes in Computer Science*, pages 120–131. Springer-Verlag, Heidelberg, 2005.

[80] A. Hauptman and M. Sipper. Emergence of complex strategies in the evolution of chess endgame players. *Advances in Complex Systems*, 10(suppl. no. 1):35–59, 2007.

[81] A. Hauptman and M. Sipper. Evolution of an efficient search algorithm for the mate-in-*n* problem in chess. In M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, and A. I. Esparcia-Alcázar, editors, *Proceedings of 10th European Conference on Genetic Programming (EuroGP2007)*, volume 4445 of *Lecture Notes in Computer Science*, pages 78–89. Springer-Verlag, Heidelberg, 2007.

[82] A. Hauptman, A. Elyasaf, M. Sipper, and A. Karmon. GP-Rush: Using genetic programming to evolve solvers for the Rush Hour puzzle. In G. Raidl et al., editors, *GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pages 955–962, New York, NY, USA, 2009. ACM.

[83] A. Hauptman, A. Elyasaf, and M. Sipper. Evolving hyper heuristic-based solvers for Rush Hour and FreeCell. In *SoCS '10: Proceedings of the Third Annual Symposium on Combinatorial Search*, pages 149–150, 2010.

[84] R. A. Hearn. *Games, Puzzles, and Computation*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2006.

[85] R. A. Hearn and E. D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1-2):72–96, October 2005.

[86] G. T. Heineman. Algorithm to solve FreeCell solitaire games, January 2009. URL `http://broadcast.oreilly.com/2009/01/january-column-graph-algorithm.html`. Blog column associated with the book "Algorithms in a Nutshell book", by G. T. Heineman, G. Pollice, and S. Selkow, O'Reilly Media, 2008.

[87] M. Helmert. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2):219–262, 2003.

[88] M. Helmert. *Understanding Planning Tasks: Domain Complexity and Heuristic Decomposition*, volume 4929 of *Lecture Notes in Computer Science*. Springer, 2008.

[89] D. W. Hillis. Co-evolving parasites improve simulated evolution in an optimization procedure. *Physica D*, 42:228–234, 1990.

[90] M. Hlynka and J. Schaeffer. Automatic generation of search engines. In *Advances in Computer Games*, pages 23–38, 2006.

[91] T.-P. Hong, K.-Y. Huang, and W.-Y. Lin. Adversarial search by evolutionary computation. *Evolutionary Computation*, 9(3):371–385, 2001.

[92] T.-P. Hong, K.-Y. Huang, and W.-Y. Lin. Applying genetic algorithms to game search trees. *Soft Computing*, 6(3-4):277–283, 2002.

[93] R. Hyatt and M. Newborn. Crafty goes deep. *ICCA Journal*, 20(2):79–86, June 1997.

[94] A. X. Jiang and M. Buro. First experimental results of ProbCut applied to chess. In *Proceedings of 10th Advances in Computer Games Conference*, pages 19–32. Kluwer Academic Publishers, Norwell, MA, 2003.

[95] A. Junghanns. *Pushing the Limits: New Developments in Single-Agent Search*. PhD thesis, University of Alberta, Edmonton, Canada, 1999.

[96] A. Junghanns and J. Schaeffer. Sokoban: A challenging single-agent search problem. In *IJCAI Workshop on Using Games as an Experimental Testbed for AI Research*, pages 27–36, 1997.

[97] A. Junghanns and J. Schaeffer. Sokoban: Improving the search with relevance cuts. *Journal of Theoretical Computer Science*, 252:1–2, 1999.

[98] A. Junghanns and J. Schaeffer. Domain-dependent single-agent search enhancements. In T. Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 99)*, pages 570–577. Morgan Kaufmann, 1999.

[99] H. Kaindl. Quiescence search in computer chess. *ACM SIGART Bulletin*, (80): 124–131, 1982. Reprinted in *Computer Game-Playing: Theory and Practice*, Ellis Horwood, Chichester, England, 1983.

[100] H. Kaindl. Minimaxing: Theory and practice. *AI Magazine*, 9(3):69–76, 1988.

[101] G. Kendall and G. Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Proceedings of the 2001 Congress on Evolutionary Computation (CEC 2001)*, pages 995–1002. IEEE Press, 27-30 2001.

[102] G. Kendall, A. Parkes, and K. Spoerer. A survey of NP-complete puzzles. *International Computer Games Association (ICGA) Journal*, 31(1):13–34, March 2008.

[103] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[104] R. E. Korf. Macro-operators: a weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[105] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[106] R. E. Korf. Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 700–705, 1997. Reprinted in Games in AI Research, J. ven den Herik and H. Iida, Universiteit Maastricht, 1999, pp. 129-141.

[107] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1-2):9–22, 2002.

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

[109] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, 1994.

[110] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, San Francisco, California, 1999.

[111] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, Norwell, MA, 2003.

[112] W. B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119, 2000.

[113] K.-F. Lee and S. Mahajan. The development of a world class Othello program. *Artificial Intelligence*, 43(1):21–36, 1990.

[114] J. R. Levenick. Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 123–127. Morgan Kaufmann, 1991.

[115] J. Levine and D. Humphreys. Learning action strategies for planning domains using genetic programming. In G. R. Raidl et al., editors, *EvoWorkshops*, volume 2611 of *Lecture Notes in Computer Science*, pages 684–695. Springer, 2003.

[116] D. Long and M. Fox. The 3rd International Planning Competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[117] S. Luke. Code growth is not caused by introns. In D. Whitley, editor, *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pages 228–235, Las Vegas, Nevada, USA, July 2000.

[118] S. Luke. *Essentials of Metaheuristics*. Lulu, 2009. URL `http://cs.gmu.edu/~sean/book/metaheuristics`.

[119] S. Luke. *ECJ: A Java-based Evolutionary Computation and Genetic Programming Research System*, 2011. URL `http://cs.gmu.edu/~eclab/projects/ecj`.

[120] T. Mantere and J. Koljonen. Solving, rating and generating Sudoku puzzles with GA. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2007)*, pages 1382–1389. IEEE, 2007.

[121] T. A. Marsland and M. S. Campbell. A survey of enhancements to the alpha-beta algorithm. In *Proceedings of the ACM '81 conference*, pages 109–114, November 1981.

[122] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[123] D. E. Moriarty and R. Miikkulainen. Discovering complex Othello strategies through evolutionary neural networks. *Connection Science*, 7(3):195–210, 1995.

[124] J. Muñoz, G. Gutierrez, and A. Sanchis. Controller for TORCS created by imitation. In *CIG'09: Proceedings of the 5th International Conference on Computational Intelligence and Games*, pages 271–278, Piscataway, NJ, USA, 2009. IEEE Press.

[125] H. L. Nelson. Hash tables in Cray blitz. *ICCA Journal*, 8(1):3–13, 1985.

[126] M. Newborn. Deep Blue's contribution to AI. *Annals of Mathematics and Artificial Intelligence*, 28(1-4):27–30, 2000.

[127] A. Newell and H. A. Simon. GPS, a program that simulates human thought. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 279–296. McGraw-Hill, New York, 1963.

[128] K. C. Ng, R. Scorcioni, M. M. Trivedi, and N. Lassiter. Monif: A modular neuro-fuzzy controller for race car navigation. In *IEEE International Symposium on Computational Intelligence in Robotics and Automation*, pages 74–79, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[129] P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6–22, 1996.

[130] E. Onieva, D. A. Pelta, J. Alonso, V. Milanés, and J. Pérez. A modular parametric architecture for the TORCS racing engine. In *CIG'09: Proceedings of the 5th International Conference on Computational Intelligence and Games*, pages 256–262, Piscataway, NJ, USA, 2009. IEEE Press.

[131] M. Orlov and M. Sipper. Genetic programming in the wild: Evolving unrestricted bytecode. In G. Raidl et al., editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1043–1050, Montreal, 8-12 July 2009. ACM.

[132] M. Orlov and M. Sipper. FINCH: A system for evolving Java (bytecode). In R. Riolo, T. McConaghy, and E. Vladislavleva, editors, *Genetic Programming Theory and Practice VIII*, volume 8 of *Genetic and Evolutionary Computation*, chapter 1, pages 1–16. Springer, 2010.

[133] M. Orlov and M. Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, April 2011.

[134] M. Orlov, M. Sipper, and A. Hauptman. Genetic and evolutionary algorithms and programming: General introduction and application to game playing. In R. A. Meyers, editor, *Encyclopedia of Complexity and Systems Science*. Springer-Verlag, Heidelberg, 2009.

[135] L. A. Panait and S. Luke. A comparison of two competitive fitness functions. In W. B. Langdon et al., editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 503–511, New York, 2002. Morgan Kaufmann Publishers.

[136] J. Pearl. *Heuristics*. Addison–Wesley, Reading, Massachusetts, 1984.

[137] E. P. D. Pednault. ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[138] D. Perez, G. Recio, Y. Saez, and P. Isasi. Evolving a fuzzy controller for a car racing competition. In *CIG'09: Proceedings of the 5th International Conference on Computational Intelligence and Games*, pages 263–270, Piscataway, NJ, USA, 2009. IEEE Press.

[139] L. Polgar. *Chess: 5334 Problems, Combinations, and Games*. Black Dog and Leventhal Publishers, New York, NY, 1995.

[140] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK, March 2008. URL `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

[141] J. B. Pollack, A. D. Blair, and M. Land. *DEMO Lab's HC-Gammon*, 1997. URL `http://demo.cs.brandeis.edu/bkg.html`.

[142] J. B. Pollack, A. D. Blair, and M. Land. Coevolution of a backgammon player. In C. G. Langton and K. Shimohara, editors, *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*, pages 92–98, Cambridge, MA, 1997. MIT Press.

[143] L. D. Pyeatt and A. E. Howe. Learning to race: Experiments with a simulated race car. In *Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference*, pages 357–361. AAAI Press, 1998.

[144] D. Qi and R. Sun. Integrating reinforcement learning, bidding and genetic algorithms. In *Proceedings of the International Conference on Intelligent Agent Technology (IAT-2003)*, pages 53–59. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[145] A. Reinefeld and T. A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[146] R. L. Rivest. Game tree searching by min/max approximation. *Artificial Intelligence*, 34(1):77–96, January 1988.

[147] E. Robertson and I. Munro. NP-completeness, puzzles and games. *Utilas Mathematica*, 13:99–116, 1978.

[148] E. M. A. Ronald, M. Sipper, and M. S. Capcarrère. Design, observation, surprise! A test of emergence. *Artificial Life*, 5(3):225–239, Summer 1999.

[149] P. S. Rosenbloom. A world-championship-level Othello program. *Artificial Intelligence*, 19(3):279–320, 1982.

[150] T. P. Runarsson and S. M. Lucas. Coevolution versus self-play temporal difference learning for acquiring position evaluation in small-board Go. *IEEE Transactions on Evolutionary Computation*, 9(6):628–640, 2005.

[151] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[152] Y. Sáez, D. Perez, O. Sanjuan, and P. Isasi. Driving cars by means of genetic algorithms. In G. Rudolph, T. Jansen, S. M. Lucas, C. Poloni, and N. Beume, editors, *Proceedings of the 10th International Conference on Parallel Problem Solving from Nature (PPSN X)*, volume 5199 of *Lecture Notes in Computer Science*, pages 1101–1110. Springer, 2008.

[153] M. Samadi, A. Felner, and J. Schaeffer. Learning from multiple heuristics. In D. Fox and C. P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 357–362. AAAI Press, 2008.

[154] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, July 1959.

[155] A. L. Samuel. Some studies in machine learning using the game of checkers II – recent progress. *IBM Journal of Research and Development*, 11(6):601–617, 1967.

[156] S. Sanner, J. R. Anderson, C. Lebiere, and M. Lovett. Achieving efficient and cognitively plausible learning in backgammon. In P. Langley, editor, *Proceedings of the 17th International Conference on Machine Learning (ICML-2000)*, pages 823–830, Stanford, CA, 2000. Morgan Kaufmann.

[157] J. Schaeffer and A. Plaat. New advances in alpha-beta searching. In *Proceedings of the 1996 ACM 24th Annual Conference on Computer Science*, pages 124–130, New York, NY, USA, 1996. ACM.

[158] J. Schaeffer, R. Lake, P. Lu, and M. Bryant. Chinook: The world man-machine checkers champion. *AI Magazine*, 17(1):21–29, 1996.

[159] J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[160] F. Servais. private communication, 2009.

[161] Y. Shichel and M. Sipper. GP-RARS: Evolving controllers for the Robot Auto Racing Simulator. *Memetic Computing*, 3(2):89–99, 2011.

[162] Y. Shichel, E. Ziserman, and M. Sipper. GP-Robocode: Using genetic programming to evolve Robocode players. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, editors, *Proceedings of 8th European Conference on Genetic Programming (EuroGP2005)*, volume 3447 of *Lecture Notes in Computer Science*, pages 143–154. Springer-Verlag, Heidelberg, 2005.

[163] H. Simon and K. Gilmartin. A simulation of memory for chess positions. *Cognitive Psychology*, 5(1):29–46, July 1973.

[164] M. Sipper. On the origin of environments by means of natural selection. *AI Magazine*, 22(4):133–140, Winter 2001.

[165] M. Sipper. *Machine Nature: The Coming Age of Bio-Inspired Computing*. McGraw-Hill, New York, 2002.

[166] M. Sipper. Evolutionary games. In C. Sammut and G. I. Webb, editors, *Encyclopedia of Machine Learning*, pages 362–369. Springer-Verlag, Heidelberg, 2010.

[167] M. Sipper. Let the games evolve! In *Genetic Programming Theory and Practice IX*. Springer, 2011.

[168] M. Sipper and M. Giacobini. Introduction to special section on evolutionary computation in games. *Genetic Programming and Evolvable Machines*, 9(4):279–280, December 2008.

[169] M. Sipper, Y. Azaria, A. Hauptman, and Y. Shichel. Designing an evolutionary strategizing machine for game playing and beyond. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 37(4):583–593, July 2007.

[170] M. Smith and F. Sailer. Learning to beat the world lose checkers champion using TDLeaf(λ), December 2004.

[171] K. Stanley, N. Kohl, R. Sherony, and R. Miikkulainen. Neuroevolution of an automobile crash warning system. In *GECCO'05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, pages 1977–1984, New York, NY, USA, 2005. ACM.

[172] G. C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2):179–196, 1979.

[173] C. S. Strachey. Logical or nonmathematical programming. In *ACM '52: Proceedings of the 1952 ACM National Meeting*, pages 46–49, 1952.

[174] I. Tanev, M. Joachimczak, and K. Shimohara. Evolution of driving agent, remotely operating a scale model of a car with obstacle avoidance capabilities. In *GECCO '06: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 1785–1792, New York, NY, USA, 2006. ACM.

[175] L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *AAAI National Conference*, pages 756–761. AAAI Press, 1993.

[176] H. Terashima-Marín, P. Ross, C. J. F. Zárate, E. López-Camacho, and M. Valenzuela-Rendón. Generalized hyper-heuristics for solving 2D regular and irregular packing problems. *Annals of Operations Research*, 179(1):369–392, 2010.

[177] G. Tesauro. Neurogammon: A neural-network backgammon learning program. *Heuristic Programming in Artificial Intelligence*, 1(7):78–80, 1989.

[178] G. Tesauro. *Software Source Code of Benchmark player "pubeval.c"*, 1993. URL `http://www.bkgm.com/rgb/rgb.cgi?view+610`.

[179] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, March 1995.

[180] A. Tettamanzi and M. Tomassini. *Soft Computing: Integrating Evolutionary, Neural, and Fuzzy Systems*. Springer, Berlin, 2001.

[181] J. Togelius and S. M. Lucas. Evolving controllers for simulated car racing. In *Proceedings of the Congress on Evolutionary Computation*, 2005.

[182] J. Togelius, R. D. Nardi, and S. M. Lucas. Towards automatic personalised content creation in racing games. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[183] G. Williams. *Adaptation and Natural Selection*. Princeton University Press, Princeton, NJ, USA, 1966.

[184] K. Wloch and P. J. Bentley. Optimising the performance of a formula one car using a genetic algorithm. In *Proceedings of Eighth International Conference on Parallel Problem Solving From Nature (PPSN VIII)*, volume 3242 of *Lecture Notes in Computer Science*, pages 702–711, 2004.

[185] K. Wolfson, S. Zakov, M. Sipper, and M. Ziv-Ukelson. Have your spaghetti and eat it too: Evolutionary algorithmics and post-evolutionary analysis. *Genetic Programming and Evolvable Machines*, 12(2):121–160, June 2011.

[186] S. Yoon, A. Fern, and R. Givan. Learning control knowledge for forward search planning. *Journal of Machine Learning Research*, 9:683–718, April 2008.

Index