Evolving Board Evaluation Functions for a Complex Strategy Game

A Thesis

Submitted to the Faculty

of

Drexel University

by

Lisa Patricia Anthony

in partial fulfillment of the

requirements for the degree

of

Master of Science in Computer Science

September 2002

Dedication

This work is dedicated to those of you who have touched my heart; part of me will always be with you. The world, and my experiences in it, have been made more complex and interesting because of you, and I am grateful for all of it.

–

Acknowledgements

Heartfelt thanks are extended to Dr. William Regli, director of the Geometric and Intelligent Computing Laboratory (GICL) in Drexel University's Department of Mathematics and Computer Science, for his advisory role over this work and all my undergraduate research endeavors. Without his inspiration and guidance, none of this would have been possible. Thanks also to the other members of my thesis committee, Dr. Ali Shokoufandeh and Dr. Sean Luke for their time, knowledge, and opinions. Also, thanks to Susan Harkness Regli for her professional advice both as a woman and as a person.

Thanks are also extended to Luiza da Silva and Michael Czajkowski for their original work on the Acquire agent system. Thanks to Max Peysakhov for insights based on his own work in genetic algorithms; and to Olga, Nadya, Dmitriy, Vadim and Craig for being such good sports when asked to play this game over and over and over... To the members of GICL with whom I have experienced my intellectual growth and coming-of-age: good luck to all of you, thanks, and good luck to the GICL family in all its future endeavors.

Table of Contents

List of Tables

List of Figures

Abstract

Evolving Board Evaluation Functions for a Complex Strategy Game

Lisa Patricia Anthony

William C. Regli, Ph.D.

The development of board evaluation functions for complex strategy games has been approached in a variety of ways. The analysis of game interactions is recognized as a valid analogy to common real-world problems, which often present difficulty in designing algorithms to solve them. Genetic programming, as a branch of evolutionary computation, provides advantages over traditional algorithms in solving these complex real-world problems in speed, robustness and flexibility. This thesis attempts to address the problem of applying genetic programming techniques to the evolution of a strategy for evaluating potential moves in a one-step lookahead intelligent agent heuristic for a complex strategy-based game. This is meant to continue the work in artificial intelligence which seeks to provide computer systems with the tools they need to learn how to operate within a domain, given only the basic building blocks.

The issues surrounding this problem are formulated and techniques are presented within the realm of genetic programming which aim to contribute to the solution of this problem. The domain chosen is the strategy game known as Acquire, whose object is to amass wealth while investing stock in hotel chains and effecting mergers of these chains as they grow. The evolution of the board evaluation functions to be used by agent players of the game is accomplished via genetic programming. Implementation details are discussed, empirical results are presented, and the strategies of some of the best players are analyzed. Future improvements on these techniques within this domain are outlined, as well as implications for artificial intelligence and genetic programming.

Chapter 1: Introduction

1.1 Problem Statement

This thesis attempts to address the problem of applying genetic programming techniques to the evolution of a strategy for evaluating potential moves in a one-step lookahead intelligent agent heuristic to play the game Acquire.

Determining an appropriately intelligent strategy for an agent player is challenging for several reasons: first, there is often no proven ideal strategy for a given game, making it difficult to ascertain the degree of optimality one can expect to attain; second, strategy-based games are complex in the gameplay interactions and the number of possible moves is large (and hence so is the branching factor for any search approach to the problem); third, even where expert knowledge may be available, the inclusion of expert knowledge is wrought with problems itself. Games such as Monopoly and chess require a great deal of knowledge, planning, and past experience in order to play an effective game against a worthy opponent. When asked to outline a detailed heuristic to perform a task, however, experts in a given domain may often forget certain key aspects which are fundamental to the problem because they seem so trivial. They may not remember other aspects which come into play only during special circumstances or rarer operations. This can be termed the "expert dilemma" and caused many problems for the designers of expert systems in the 1980s, which is well-known, and supported by statements in [17].

Therefore, an acceptable alternative to teaching the computer how to operate within a given domain can be to teach the computer how to *learn* to operate within that domain. Simple versions of this are case-based planners; more complex methods include neural networks, decision trees, and reinforcement learning. Another interesting technique is that of evolutionary computation, specifically, genetic algorithms and genetic programming. Ge-

netic programming is the process of providing the computer with the building blocks to use in constructing some solution to a problem which the human programmer cannot himself solve, or can't solve in an acceptable time frame (i.e., exponential-time algorithms), and the means to evaluate the degree of "fitness" of a proposed solution. Genetic programming has been praised as a natural approach to developing algorithmic behaviors [27].

Acquire, a strategy game involving the building of and investing in hotel chains and effecting mergers of these chains to gain cash rewards, is a game whose dynamics seem comparable to other games that have been approached via genetic programming. In studies on other domains such as Backgammon, the "ergodicity" of the game (the reversibility of the possible outcome at any time during the game due to the randomness of dice rolls), means that the agents playing can actually learn from the progress of a game even when they lose [31]. Acquire has a similar random aspect, and this theoretically allows the genetic programming to drive forward toward some optimum solution, as more and more of the search space is explored due to the changing dynamics within different games. Aspects of Acquire's rules of gameplay make this domain a very interesting application of the use of genetic programming techniques to yield competent player strategies.

1.2 Overview of Approach

There are several steps involved in preparing a genetic programming specification of a problem. These are to enumerate, as outlined in [1], the architecture of programs to be evolved, the set of primitive programmatic ingredients, the fitness function, and the parameters for controlling the run.

In the case of the Acquire game, what are the building blocks? What makes an individual strategy more "fit" than another? Those familiar with the Acquire domain can recognize that, during gameplay, one makes use of observations about the state of the world to decide the potential benefit of possible moves. Human players also use knowledge about past moves and theories about the other players' strategies in their analysis, but these aspects

are not incorporated into the problem addressed in this thesis, as we have used a more basic approach. Probabilistic lookaheads and the concept of utility of certain moves can be built into future endeavors on the topic. In many applications of reinforcement learning or autonomous agents, the environment is not fully accessible or static. Simplistic "toy problems" which allow the agent to operate with full information about the environment are not representative of most real-world applications of autonomous mobile robots or in certain games, where the other players' hands are unknown. Human memory also is not perfect, and it is preferable not to allow agent players to maintain perfect memory of the progress of the game, so as not to give them an unrealistic, unfair advantage.

For these reasons, and also because the agents only use a one-step lookahead, and do not incorporate any long-term planning, our Acquire agents can only make a decision about a move to make based on the resultant state of the world after that move would be made, in other words, the utility of that move for the agent.

Although Acquire's outcome depends on a simple economic utility model (i.e., whichever agent has amassed the most money wins the game), providing only information about how much cash is on-hand would be an ineffective strategy, since in order to win money, one must spend money. (Section 3.1.1 has further details on the Acquire game.) Therefore, we provide our agents with a larger function set based on the parameters about the world which are available to the agent, with the intent that this will be more robust than simple reasoning based on cash assets. We hope to evolve an implicit "lookahead" based on these parameters about the world which the agent may combine in interesting ways via the evolution process. The key to this is providing the functions to extract the appropriate information about the world to allow this projection to take place, without imposing human biases on the elements to consider.

We considered several methods of evaluating the fitness of individuals, including the number of wins in the 10 games played against opponents, the total money earned in the series of 10 games, and, ultimately, the ratio of money earned by the agent to the total amount

of money earned in that game by the players. It is important to note that the evolution of strategies against other agents which are not themselves optimal will not necessarily lead to an optimal solution. Rather, the functions will only evolve enough to beat the provided opponents, be they random or hard-coded. In fact, hard-coded opponents are particularly ill-suited for use in GP because the population will simply evolve to exploit weaknesses in the hard-coded strategy, and are therefore brittle when placed against other strategies they had not faced before [27, 31]. Coevolution is designed to force the population toward optimality without falling into local minima in this way, by pitting the evolving strategies against other individuals in the same population (or in the other population, in the case of two simultaneously evolving populations). Therefore, we evolve one population of individuals whose fitness is judged on the basis of a given number of games played against randomly-selected opponents from the population, similar to the approach taken by [6].

1.3 Outline of Thesis

Chapter 1 is this introduction to the problem and our approach. Chapter 2 presents some background information on game theory and genetic programming, including coevolution. Chapter 3 discusses the approach we took to setting up Acquire as a genetic programming problem, and certain adaptations which were made, both to the genetic programming technique and to the Acquire game rules and operation.

Chapter 4 presents the results from our two genetic programming runs, and some discussion. Finally, Chapter 5 offers conclusions as to the contributions of this research, as well as its limitations, and a large helping of future and related work, pointing to avenues which might improve performance in this domain.

Chapter 2: Background

2.1 The Analysis of Game Interactions and Board Evaluation Functions

The study of games has been used to approximate problems in mathematics, economics, evolution, social interactions, and many other areas, since the first half of the 20th century [44]. A "game" is essentially any situation in which a decision has to be made [12]. The *players* determine how to allocate their resources based on stimuli from the other player(s) or the environment, and do so when it is their *move* (or *turn*, as in parlor games), choosing from a variety of available *plays*. Each player accumulates payoffs, which can take any form depending on the domain, and whether the game is competitive or cooperative. In competitive games, payoffs to players vary inversely—when one player wins an amount, the other player(s) either lose that amount or are penalized by not having been able to win that amount themselves; in cooperative games, the payoffs vary directly, and the players act as a team or other coordinating unit. Games can also be neutral, wherein the payoffs to each player are unrelated, as in single-player games against chance, nature or the environment [7], known as "disinterested players" [13].

During the course of the game, players make use of certain *strategies* to make their decision among the various plays available to them in any particular move. In typical computer approaches to the development of game strategies, certain conditions apply: the game is of full information (in that the full range of plays available to each player is known to all); it is competitive and zero-sum (in that the payoffs to one player are the other player's losses, or that the players' interests are diametrically opposed [13]); and it is not necessarily linear (in that the outcome of the game depends on a series of plays rather than only one).

In our approach, however, we simplify the game construct (to fit Acquire's model) to be that of a two-player game where there are a finite range of available plays at each move, and where the turns proceed in alternating fashion, rather than simultaneously. Our domain

is *not* a game of full information, and it is stochastic in nature. This means that a minimax strategy, whose purpose is to reduce (minimize) the (maximum) damages an opponent can do in a given play [44], is not appropriate to our domain. The minimax strategy does best in simple zero-sum games where an equilibrium point exists such that the minimax is the same point from both players' perspectives, forcing rational players to make the decisions at the equilbrium point. In games which incorporate a random element such as a die, however, there will be no equilibrium point, and trying to decide a move based on minimax does not help. Acquire is a zero-sum game (also called "constant sum" [12]), but not because there is a fixed amount of money to win, which is where the term arose. It is still zero-sum because an Acquire player cannot simply let his opponent play unchecked. For a player to ensure that he has the most assets at the end of the game himself, he must also curb the winnings of his opponent. Their interests are in this way "diametrically opposed".

A very good analogy to the Acquire game scenario which illustrates how the game fits the zero-sum paradigm is presented in [13] on pages 45 through 46: the best strategies for two political parties to use in trying to win state electoral votes during presidential elections. Each state is independent of each other, and so the contest for the votes in New Jersey is separate from the contest in California. But each party's candidate must win a majority of the individual contests to win the general contest for the presidency. Each party allocates resources for campaigning and publicity within states and must determine the best distribution strategy. As we will discuss in Section 3.1.1, this scenario is similar to the contests for majority stockholder in the individual chains on the Acquire board. There can be only one winner, so both players' interests are clearly "diametrically opposed", and the players must choose how to allocate their resources in order to win the contest for highest buy-in into the most chains on the board in order to win the game.

Regarding a strategy as a decision-making plan on a global game scale when the game can be made up of many small contests implies that each particular move can be approached independently. The ability to evaluate board positions as being either advantageous or

disadvantageous to a player's status in the game is part of a successful strategy for most complex strategy-based games. For our purposes, a strategy-based game is one in which there is a choice of moves and the outcome of the game depends on the combination of the values of a series of plays, as opposed to simpler games where the choice of next move is determined solely by the rules of the game or by chance (for instance, roulette). Often, a board evaluation function is chosen as a representation of the player's strategy or heuristic for gameplay; the player weighs the moves available to him based on the evaluation of the board state (either now, or as a result of the move being considered) and chooses the one with the highest value. This approach can succeed for even the most complex games because, in any finite game, each position or world state represents either a win for the player under consideration, a loss for that player, or a draw [13]. For this reason, one can develop a strategy for a game which is based on an analysis of the current board position's advantage to the current player.

Natural evolution is itself often viewed as a game, where the players are the biological species competing for resources in nature, and each adaptation is a move, the payoff being which species survives and reproduces [40]. The analogy between games and evolution leads to interesting possibilities in applying natural evolutionary techniques to certain games, especially since many games do not fit into the simplified minimax paradigm discussed above. These games may be solvable via certain heuristics which are not obvious, and computers can be used as tools to find strategies for these more difficult games. Even for games where minimax *can* be performed, but there is interest in reducing computational complexity caused by the high branching factor, as in Checkers and Backgammon, evolutionary algorithms yield advantages.

2.2 Evolutionary Computation, Genetic Programming and Coevolution

Machine learning is the larger umbrella under which evolutionary computation and genetic programming sit. Two areas of artificial intelligence of relevance are knowledge-based approaches, wherein we tell the computer precisely how to perform a given task; and machine learning, wherein we tell the computer "precisely how to learn" how to perform tasks (p. 8) [4]. Machine learning is often defined as the study of various techniques by which to teach computers to learn to do certain things, "without explicit representation of symbolic knowledge" and hand-coding by the programmer[37]. Evolutionary computation draws on analogies from Darwinian approaches to biology, ecology and genetics to describe how the computer system "evolves" toward the correct answer, being given only a means by which to determine the appropriateness of a certain response.

Evolutionary computation offers advantages over traditional algorithms when attempting to solve real-world problems for which there may not be a known optimal solution or even an optimal algorithm to use in solving it. Darwin's model of evolution essentially defines a complex search and optimization mechanism, designed to allow biological species to overcome the challenges of nature, including chaos, chance, temporality, and nonlinearity [17]. These challenges are often the ones which make the design of algorithms for real-world problems so intractable. Therefore, by virtue of its analogy with natural evolutionary systems, evolutionary computation possesses several distinct characteristics which make it a powerful choice of technique for such problems; several of these were discussed by Fogel in [17]:

- conceptual simplicity—the complex mechanisms of genetics are brilliant in their simplicity and therefore easy to follow;
- broad applicability—the genetics and evolution analogy has been found to extend to all areas of real-world optimization problems;
- out-performance of classical algorithms and ability to solve problems with no known

solutions—exponential-complexity problems can be solved or approximated in much less time by evolutionary computation methods;

- potential to hybridize with knowledge-based approaches—thereby taking advantage of what expert knowledge may exist in a domain;

- self-optimization—the population proceeds toward an optimum without the need for human intervention or guidance.

The genetic programming (GP) paradigm was first described by Nichael Cramer [10], and later expounded upon by John R. Koza in his 1992 book, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*[23]. GP was formulated as an extension of genetic algorithms [20]. Two good references on the subject are [4, 29]. In GP, the system optimizes a computer program or function to solve a given problem by generating individual programs or functions and rating their *fitness* to the solution, choosing those best fit to survive and allowing them to reproduce. The programs, also known as *genomes*, are made up of atoms known as *terminals* (e.g., state variables or functions of no arguments) and *nonterminals* (e.g., functions which take arguments). The first generation of GP is produced randomly, the individuals are tested via a given "evaluation function" or "fitness function", and they are then assigned a score. Next, those individuals with higher fitness are selected via certain means and allowed to pass on their genetic material to the following generation via certain procreation operations. See Figure 2.1 for an illustration of the GP life cycle, after [30, 17].

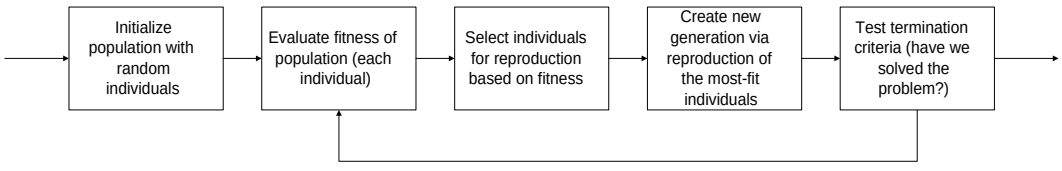

Figure 2.1: Control Flow of the Genetic Programming Paradigm

Koza's kernel for GP problems was written in CommonLISP; LISP S-expressions are natural syntactic ways of representing these individuals' genomes, since they can be represented as parse trees and easily manipulated. The LISP code fragments which make up the individuals form GP trees, such as the one shown in Figure 2.2. That example shows the following LISP statements:

```
(+ 4 5)
```

```
(if-chain-2-is-safe-from-merging
      (- (price-of-chain-1)
         (payoff-value-to-player-of-chain-1))
      (payoff-value-to-player-of-chain-1))
```

These statements are built from terminals and nonterminals as mentioned above. Each internal node is a nonterminal, which takes as arguments its children. Any number of arguments is permitted, although here only binary functions are shown. The leaves of the GP tree are terminals.

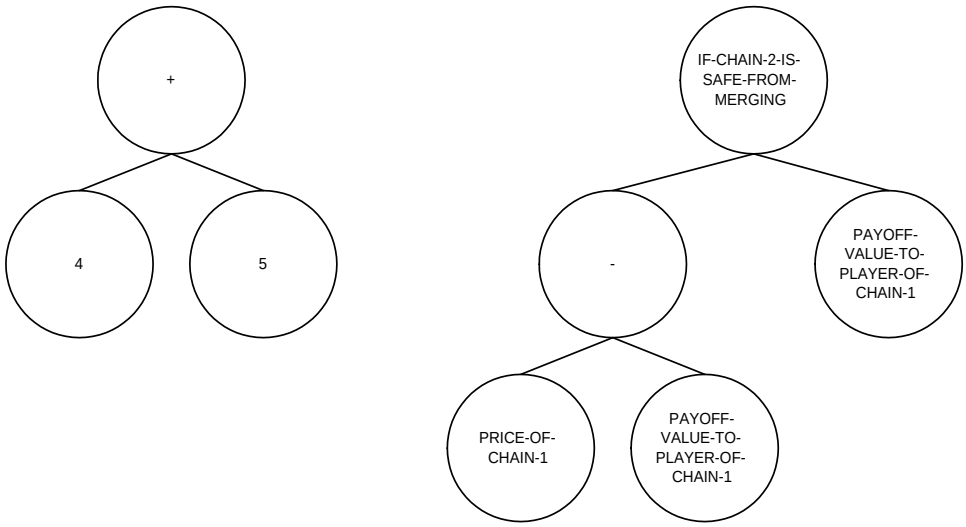

Figure 2.2: Examples of Parse Trees for Various LISP S-expressions

The evaluation function (fitness function) is also provided to the GP system during problem specification. After the random initialization of the population in the first generation, the GP algorithm determines the fitness of each individual via this fitness function, selects the most fit genomes, and breeds them to a new generation. The fitness function is highly problem-specific. In symbolic regression (the interpolation of a mathematical function based on a set of input and output values), for example, the fitness function is generally the number of data points the individual matches within some margin of error. In certain games, tournaments among the individuals are run and the fitness of each individual is defined to be the rank in the tournament that individual attains. The algorithm terminates when an optimally fit individual has been found, or when the predetermined number of generations has completed. Which termination criteria is more appropriate is also problem-specific.

The breeding operators which are most commonly used are reproduction, crossover and mutation. The *reproduction* operator chooses a relatively fit individual and passes it on to the next generation unchanged. When the most fit individuals, or some fraction thereof, are reproduced for every generation, it is called *elitism*. In *crossover*, shown in Figure 2.3, subtrees within two individuals are swapped at random; the subtrees do not have to be of the same size or occur in the individuals in the same position in the genome. *Mutation*, shown in Figure 2.4, is generally one-point mutation, and involves the replacement of a given subtree in the genome with some randomly generated subtree (not necessarily of the same depth as the tree it is replacing).

Some researchers may be tempted to dismiss the techniques of GP as little more than blind random search. Indeed, it is true that genetic programming can be thought of as a guided “beam search”, where the algorithm is limited to areas of the search space which satisfy a certain criteria; here, the fitness function acts as the beam [23]. The selection and reproduction operators open new areas of the search space to be explored by instantiating potential solutions within these subspaces to be considered. Consider that the solution(s)

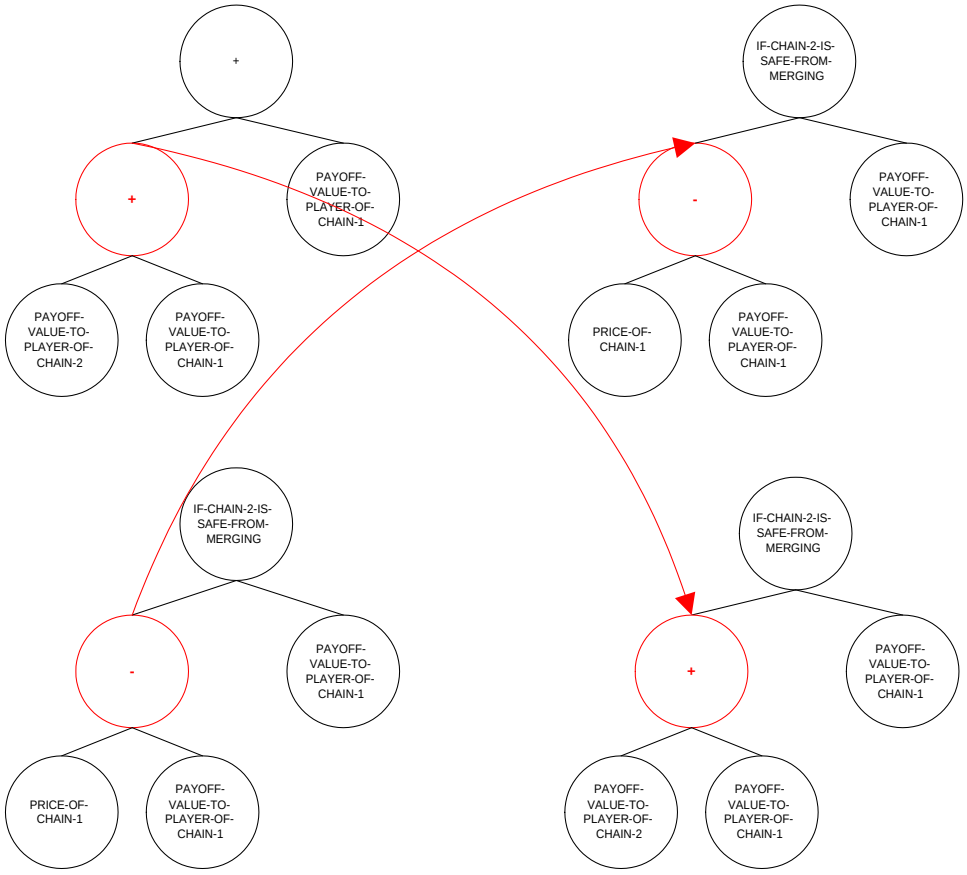

Figure 2.3: Examples of One-Point Crossover at a Function Point in a Parse Tree

to the type of problems genetic programming addresses are typically only one or a few points in a space with a vast number of possibilities, and these possibilities are determined by the number of possible programs which can be generated by the given set of terminals and nonterminals based on the branching factor determined by arguments taken by each nonterminal and by the allowable depth of individuals. While typical GP techniques may examine up to 10,000 or 100,000 individuals [28], this is still a far smaller number than the total number of possible individuals. Of the total number of individuals, vast stretches of the search space are filled with completely unfit individuals which cannot lead to promising end results. The GP evaluation loop acts as both a filter and a lens, sifting out areas of the search space which are unprofitable to explore and focusing the progress of evolution

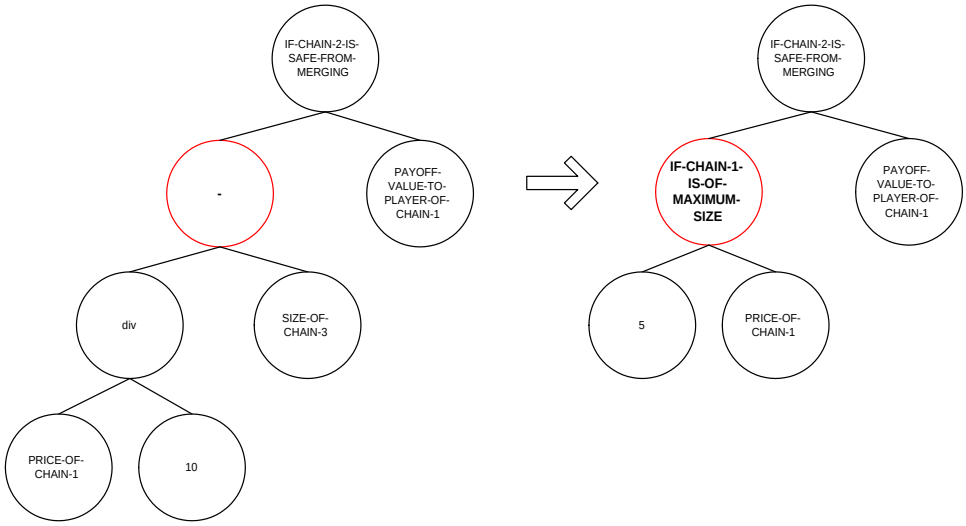

Figure 2.4: Examples of One-Point Mutation of a Parse Tree

in directions which lie along certain key paths through the space of possibilities. For a more in-depth discussion of this and other arguments supporting GP as an independent improvement over blind random search, see Chapter 9 of Koza's 1992 book [23].

GP was first used for symbolic regression, that is, the interpolation of a mathematical function based on certain data points. Individuals are scored based on the number of the data points they can match or approximate within some error margin. GP has had much success evolving the optimal solution, or individuals which are functionally equivalent to the optimal solution, as the generations progress. Some problems, however, do not have an easily available optimal solution or expert player (as in the case of game strategies). In this case, a fixed fitness function may not be appropriate. Coevolution has been proposed as a means by which to avoid the paradoxical need for an optimal solution while trying to evolve one [33]. Coevolution allows the population to evolve by competing against each other; this causes the problem to become more difficult naturally as the individuals in the population improve [25].

There are several different coevolution paradigms. The first is modeled after natural coevolution in the biological/ecological world, wherein two separate populations (i.e., two

separate species) balance the relationship between themselves, as in predator-prey relationships. In this case, the fitness of one population is determined by how well it performs (survives) against the other population. This sort of coevolution is seen throughout the natural world, and amounts to a sort of biological "arms race" [21], meaning both populations progress toward some unknown optimum, compounding adaptation upon adaptation. As one species of plant develops a tough outer shell to resist attacks by a certain insect predator, the species of insect develops a stronger mandible. The plant then develops a poison to kill the insect, but ultimately the insect develops an immunity to this poison, and so on, ad infinitum (example from [23]).

A second type of coevolution is also called "self-play", wherein the population's fitness is determined by competition amongst its own individuals [5]. This is analogous to resource competition in nature. This is the type of coevolution we chose to use; it is more logical in situations where the competition is symmetric. In a predator-prey competition, the relationship is not symmetric: each population has a different goal (the predator attempts to catch the prey, and the prey attempts to elude the predator). In many game domains, however, the competition is symmetric such that neither player has an advantage over the other merely by virtue of the domain constraints. While some games have distinct first-player advantages, Acquire is not one of these.

However, the changing fitness function that coevolution provides as an advantage for game domains presents a problem in itself: that of determining whether the population is actually making any progress toward the optimum at all. Because the population is judged against itself, as generations pass, the fitness function itself changes. This is called the "fitness landscape" and alters as more fit individuals are judged against their neighbors, who are also more fit (assuming the population is progressing). [9] describes what is known as the "Red Queen effect" in coevolutionary interactions: the populations, by nature of their interaction alter the fitness landscape. The effect is named after the Red Queen character in Lewis Carroll's *Through the Looking Glass*, who was always running but never getting any-

where, because the landscape moved as she did. In essence, the horizon is always the same distance away. A graph of the performance of the individuals over time, which is relative to the others, may not have the upward slope we are expecting, even though progress may be occurring. This problem was pointed out by [9, 35, 36]. As the population gets better, it's more difficult for them to beat each other. Techniques have been developed to monitor the monotonicity of the progress more carefully [9]: tournaments of the population champions against all previous champions ("ancestral opponent contests"); and distance metrics using similarity of the parse trees developed ("genetic distance measures"). These visualization and measurement techniques help allow the true progress of the population to emerge.

Coevolution presents the corollary issue of collusion, wherein the players "cooperate" to repeatedly draw against one another, or wherein the problem scope becomes so narrow over the course of generations that, effectively, the same game is played by the players each time. In this case, one population has fallen into a subarea of the search space which the second population does not use (cover), and in this way they will avoid competition. Certain domains will prevent this naturally, however, via certain characteristics: ergodicity (i.e., reversibility), stochasticity, and continuity of the domain combine to make this simple partitioning of the search space not possible [5]. It follows from familiarity with the Acquire domain that it fits these characteristics. The random elements of gameplay alone ensures that different areas of the board will be developed at different times, forcing strategies to be robust enough to compensate for various situations and interactions.

Chapter 3: Approach: Architecture and Implementation

3.1 Core Acquire Agent System

The Acquire Agent System began as a simple distributed agent planning system implemented in Common LISP and Java. The purpose of the project was to design and implement an agent system which was capable of playing the game Acquire, a trademark of Hasbro, formerly Avalon Hill [19]. Eventually that system was modified and updated to function as the core for the genetic programming experiments which are the object of this thesis.

3.1.1 Discussion of Acquire Game Play

Acquire is a multi-player strategy game which has been called similar to Monopoly in concept and objective (see Figure 3.1). For two to six players, the game's objective is to build up hotel chains on the board squares, buy stock in these chains, and effect mergers which yield payoffs to those who own stock in the merged chains. The game operates on the simple economic model of monetary gain; the player with the most money at the end of the game is the winner.

Each agent's turn proceeds in three phases. The first is the "place-tile" phase, during which the player must choose one of the 6 labeled tiles he is holding in his hand and place it on the corresponding square on the board. This tile placement can have one of several effects: it can create a new hotel chain if it is adjacent to another tile which is not yet in a chain; it can increase the size of an existing hotel chain; it can cause a merger if it is adjacent to tiles which are in two or more different chains; or, most simply, it can be placed alone on the board.

During a merger, the larger chain "wins" and consumes the smaller one. Each hotel/tile which was previously in the now-defunct chain is subsumed by the larger chain.

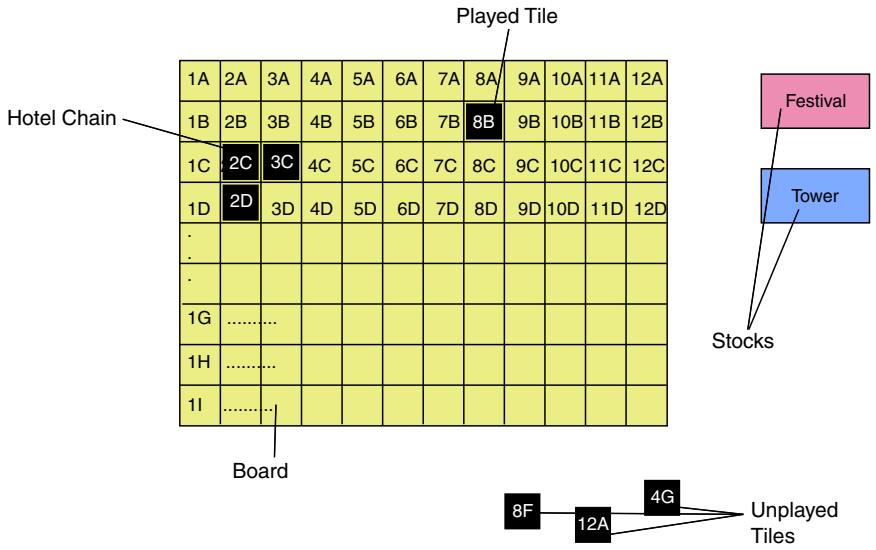

Figure 3.1: Diagram of the Acquire Board and Some Key Aspects of Game Play

Stockholder bonuses are distributed, one to the player which owns the most stock in the now-defunct chain, and one to the runner-up. These bonuses are determined by the classification of that chain (low-value, medium-value, high-value) and its size at the time of merging. Because the smaller chain is no longer active on the board (although it can be played again in the future), all players who own stock in this chain may choose to either sell their shares or trade them in for stock in the winning chain.

After the merger has been handled, the player's turn continues. Next, the player must decide what stock, if any, to buy in any of the hotel chains currently on the board (including any created during this turn). Because the stockholder bonuses are the only way to make money during the game, it is to the player's advantage to be the majority stockholder in as many chains as possible. In addition, whenever a player's tile placement creates a new chain, he receives one stock in that chain for free. After buying stock, the player draws a new tile at random from the central pot. Play proceeds in this fashion until the end of the game occurs under one of the following circumstances: a player plays his last tile; a chain reaches a certain large size; or all chains are above a certain no-merger threshold.

Due to the interaction between the place-tile phase and the buy-stock phase, there is a heavy element of strategy to gameplay. Because the overall monetary assets of the players at the end of the game determine who the winner is, and because the only way to gain money is to receive shareholder bonuses for being the primary or secondary stockholder when a chain merges, all decisions must revolve around what stock a player owns, and how close to catching up in quantity one's opponent(s) may be. Losing the majority shareholder's bonus in a valuable chain may have a severe damaging effect on the outcome of that player's game. If one player wins a majority shareholder's bonus, the other players are unable to win this money, so it is a loss in status as far as game standings. In the same way, if one player buys three stocks in a particular chain, the other player has three less available to him, which may lead to his eventual inability to gain a majority stockholder's bonus in that chain.

3.1.2 The System

The Acquire system is implemented in CommonLISP, modeled as a central game controller and an agent interface for each player. Each agent-player has the ability to manipulate a hypothetical state of the world which is updated after each event in the game. The Acquire system uses some aspects of the CommonLISP Object System (CLOS) in an attempt to reduce memory costs; however, space constraints were never a major concern. LISP speed optimizations were also not focused on, in spite of the high computational demands of genetic algorithms and genetic programming. To manage these high time demands, we used a compiled version of the system when playing games with our evolved strategies. This compiled version ran nearly a factor of 10 faster than the interpreted version, and this time frame was sufficient for our needs.

To make the choice of what move to make in a given turn, an agent player considers each possible move available to him on his current turn, both for the tile-placement decision and the stock-purchase decision, and how this move would change the state of the world.

He uses the evolved evaluation function on each of the resulting states of the world (i.e., one for each possible move), to determine which one is most advantageous to him. Three types of agent players are supported by our Acquire system: a random agent picks moves at random without considering how they may affect the world; a smart agent actually uses a function to evaluate the state of the world; and a human agent requests input from the user as to which move to make.

The game is by nature stochastic in that a player does not know what moves will be available to him in the future due to the randomness of the tile drawing, or what moves his opponents have available. We decided to focus on the evolution of a board evaluation function, rather than a full-on strategy. The difference is that a strategy is typically thought of as some organized set of condition-action pairs, where the condition is based on some test or evaluation of the state of the world, and the action is the move to make. An evaluation function, however, is simply the condition part of this; the evaluation function tests the state of the world and returns a number representing the value assigned to that state of the world. Because Acquire players have a widely varying choice of moves to make from turn to turn, there is the desire to constrain moves the agent is allowed to consider to be legal moves only. In evolving a strategy, the agent will be able to recognize what moves would be *best* to make, but these may not necessarily be ones currently available to him. In the remainder of this thesis, however, we will use the term “strategy”, so as not to be confused with the genetic programming individual fitness evaluation function.

3.2 The Genetic Programming Extensions

The Acquire system was adopted as the core of a genetic programming problem in the vein of those from Koza’s 1992 book [23]. Koza’s genetic programming kernel code, implemented in CommonLISP, was taken and slightly adapted for our purposes. These adaptations are explained below. When designing a problem to be solved via GP, there are several fundamental steps. First, the architectural representation of the program or algorithm to

be developed must be specified; in our approach, we use Koza's LISP S-expression parse trees. Second, one must provide the language elements the GP system can use in evolving the programs (in our case, board evaluation functions). Third, one must provide the fitness function to use in determining which individuals perform better at the given problem than others, such as a metric based on the ability of certain individuals to win. Fourth and finally, one must define certain GP operation parameters which may have values specific to the problem at hand [1]. The formulation of the Acquire strategy-evolution problem as a GP problem drew on Koza's explanation of the required elements for such a problem [23]:

The terminal set. Our intuition about the structure of the state-evaluation function which we were evolving directed us to consider it as some combination of certain parameters about the current state of the world; therefore, we decided to use these parameters as our terminal set, to be combined with simple arithmetic operators. Anything which is visible to a human player of Acquire at a given time point in the game, minus the effect of human memory of opponent moves, is available to be queried by the smart agent. The program evolved by the GP system is plugged into the Acquire system as the evaluation function for one of the agent players, so the parameters are accessed via functions which the Acquire system can execute during gameplay. See Appendix A for the complete table of all terminals used and their descriptions.

The board-querying functions are of the form `size-of-chain-1`, `size-of-chain-2`, `distance-to-nearest-neighbor-of-chain-1`, `cash-held-by-player`, and so on, where `chain-1` refers to the largest chain, `chain-2` refers to the middle-sized chain, and `chain-3` refers to the smallest chain. The game actually uses seven possible hotel chains, which would result in a combinatorial explosion of parameters to consider (over 50). The difficulty with this is the computational time it could take to evaluate the relationships between these parameters. It is tempting to reduce the number of parameters based on our own assumptions about which ones may be relevant to a given decision. However,

once the decision has been made to allow the computer to evolve its own strategy, one cannot make any judgments about what parameters about the world "don't apply" to certain move-making decisions. We made an initially rather arbitrary choice to reduce the problem to allow only three possible hotels as described above, which decreases the computational complexity considerably, while still allowing the system to use all possible observations about the state of the world in fixing its strategy. As we later discovered with human players, though, the game with only three hotels is both slow and uneventful over long periods of time, and the potential to earn money is drastically reduced. However, although the rules of the game have been changed by this stipulation, the application of the GP technique is the focus of this thesis, and the minor changes to the game which result from reducing the number of hotel chains available do not affect the GP system's evolutionary progress.

The function set. There is some precedence for a genetic programming approach to evolving board evaluation functions for complex strategy games [16, 15]. The nonterminal sets in these cases included conditional operators to reflect the complexity of the strategies needed to play certain games (i.e., in certain combinations of situations, the board value would be higher than in certain others, or perhaps two conflicting set of situations would affect the value of a given state). However, our original idea was that this evaluation function we were evolving would be similar to a utility function. According to Russell and Norvig [37], multi-attribute utility functions where the attributes are independent of each other can be combined (in theory) via simple arithmetic functions. While we had intended to explore the possibility of incorporating probabilistic reasoning in the strategy, the representation utilized both by the GP system and the Acquire system did not lend itself readily to the evolution of neural networks and other reinforcement learning techniques which have been discussed in the literature (see Section 5.3).

However, we did want to enable complex strategizing based on less fundamental board state operations. For example, if a player is the majority stockholder in chain X, and the

move under consideration would merge chain X with another, larger chain, the resulting board state should have a high value compared to one in which the player's opponent owns the majority. Therefore, in addition to the four basic arithmetic operators: `+`, `-`, `*`, and `div` (where `div` is a protected division operator to guard against divide-by-zero errors), we have also included three conditional operators that hinge on certain board states being true. We did not simply include logical and decision operators like `if`, `and`, `or`, and `not`, because this would have required strongly-typed GP in our implementation of the problem (see Section 5.3 for more on this).

See Appendix A for a complete list of all functions used and their descriptions.

Problem-specific functions. Often, some domains require other functions which the GP system uses to help in its creation of new individuals or its evaluation of them. In our case, the entire Acquire system is used as the method of evaluation, and so it could be said that the Acquire game functions are problem-specific functions needed for this problem. In addition, the `div` function and functions for choosing a random set of opponents were included in the kernel.

Fitness cases. We entertained the idea of several options as to how to evaluate the fitness of individuals, including the number of wins in the 10 games played against opponents, the total money earned in the series of 10 games, and, ultimately, the ratio of money earned by the agent to the total amount of money earned in that game by both players. Fitness of individuals in the population can be evaluated simply based on a binary "win-lose" statement when the individual is pitted against an opponent in a game. One can also use a "degree" of fitness, based on how much of a monetary advantage (or disadvantage) the individual had over his opponent at the end of the game. In these situations, the opponents could be random agents, or agents which are also using some sort of evolved strategy.

Ultimately, our problem specification did not use fitness cases in the strictest sense of word and as intended in Koza's kernel. The fitness cases for this domain, rather, consist of

several games played within the Acquire system using the current individual as the evaluation function for one of the agent players. We chose 10 games to be fairly representative without causing the time needed to complete a run to explode. The agents played against a random sample of other agents from that generation of the population, as a simple form of coevolution using competitive fitness. The fitness of an individual was defined to be the average over 10 games of the ratio of money earned in each game to the total money earned in that game by both players. This ratio penalized players for losing disgracefully, and rewarded players who won by a large margin more than those that won by a very small margin.

Result from evolved program. The wrapper function which obtains the result from the evolved program (the evaluation function) plays a game of Acquire and extracts the final standings. The ratio of earnings in that game is calculated.

Test result against fitness cases and return a score. Here, our problem is also different. When all 10 games are played, the average ratio is determined and adapted as the standardized fitness. The standardized fitness is defined such that lower numbers mean higher fitness.

Problem-specific parameters. As has been noted in other literature on genetic programming and evolutionary computation, parameter-setting is often a "black art", with no real justifiable reason to choose certain parameters over others, other than they yield better results after empirical testing [11].

In our case, we chose 10 fitness cases because playing only one game would certainly be too random; the agent could have had a particularly lucky game and not won based on its strategy at all, or vice versa for its opponent. We did not want a currently high-ranked individual to be unseated by a lucky novice challenger. This is actually referred to by Pollack and Blair as the "Buster Douglas effect" [31]. However, any number of games

over 15 would take too much computer brawn than was available to us to evaluate in any reasonable amount of time (we were attempting runs in the space of hours or days, rather than weeks).

The maximum depth to search and the maximum depth for individuals was set at five to avoid highly bloated individuals which would slow down evaluation. Crossover ratios and mutation indices were kept as the defaults which Koza himself used in his other sample problems [23]: crossover at any point was kept at 20%, and at function points was kept at 70%, to bias crossovers to occur at function points. The mutation rate during the reproduction operator was kept at 90%.

The termination criterion. Typically, a genetic programming run is allowed to terminate when an ideal individual has been found (i.e., one that satisfies all the fitness cases). When evolving a game strategy, however, there is no ideal example, and any number of individuals could perform well. Coevolution is meant to drive the evolution forward, so the best choice is to push through a given number of generations. Therefore, our only termination criterion was the completion of the number of generations which we forced our GP system to execute.

See Table 3.1 for a summary of the problem outline, including parameter values.

As mentioned in Section 1.2, it is important to realize that the evolution of strategies against other agents which are not themselves optimal will not necessarily lead to an optimal solution. The functions will only evolve far enough toward optimality to beat the provided opponents, be they random or hard-coded. Hard-coded opponents are particularly ill-suited for use in GP because the population will simply evolve to exploit weaknesses in the hard-coded strategy, and would therefore be brittle when placed against other strategies they had not faced before [27, 31]. On the other hand, coevolution is designed to force the population toward optimality without allowing it to fall into local minima, by pitting the evolving strategies against other individuals in the same population (or in the other

Table 3.1: The Acquire GP Problem Specification

Objective:	To find a board evaluation function for playing the game of Acquire.
Terminal Set:	A comprehensive-as-possible list of board feature extraction functions, the digits from 0..9, and a random floating-point constant.
Function Set:	The four simple arithmetic functions, and three conditional operators based on desirable board features, all taking 2 arguments.
Fitness Cases:	n/a
Raw Fitness:	The average ratio of money won in the 10 games played where each ratio is the proportion of the total money earned in that game which the given individual is responsible for.
Standardized Fitness:	We subtract the raw fitness from 1.0 to force smaller values to imply higher fitness.
Hits:	n/a
Wrapper:	The wrapper extract the amount of money won by that agent compared to its opponent in a particular game of Acquire from the final standings.
Parameters:	Population size = 1000, Number of Generations = 50.
Success Predicate:	No success predicate is used; the GP run proceeds until all generations have been executed.

population, in the case of two simultaneously evolving populations). We did not use two populations, as is done in much of the literature on competitive coevolution [23, 36, 33], because in those cases, the two opponents are asymmetric. For instance, in the predator-prey problem, an opponent is evolved to be either the predator *or* the prey. In games such as Tic-Tac-Toe or Nim (pick-up sticks), the player to move first in a game has an advantage over the second player, and the overall strategies therefore differ slightly [35]. This is largely true because the games are so simple and the branching factor is small, lending the game to global analysis. In games such as Acquire, Backgammon, and Checkers [6], however, the opponents are effectively symmetric. In any given state, it is true that one player might have an advantage over the other based on who gets to move next, and a localized strategy might be more effective for each turn. As this advantage can pass from one to the other as turns progress, any general, global strategies developed (as we are attempting) must not overfit to these potentially transient advantages. Therefore, we use one population whose fitness is judged on the basis of a given number of games played against randomly-selected opponents from the population, similar to the approach taken by [6].

Initially, we actually perceived our domain as providing two separate GP problems to solve. During an agent's turn, as mentioned in Section 3.1.1, he faces two choices. The first is which tile to place; the second is which chain to invest in. These strategies are sufficiently separated (we believed) to support their separate evolution. It was unclear in our early stages of familiarity with GP techniques, how to evolve two separate trees using Koza's GP code without altering it significantly, as Luke did for his kick and dash trees in evolving soccer-playing softbots for the Robocup simulation league [26]. In hindsight, this may not have been as difficult as was believed when the project was begun. In fact, there are strong reasons justifying the evolution of one player strategy with both functions incorporated, centering around the "credit assignment problem", mentioned in much of the literature where coordinating programs must be evolved [27, 18]. The credit assignment problem refers to the difficulty of assigning fitness to the individuals in a team based on a win or

loss. How much of the win can be credited to any one individual? How much of the loss can be claimed to be a certain individual's "fault"? While computationally more complex, and requiring some amount of structural constraints (see Section 5.3), evolving teams as a unit sidesteps this credit assignment problem. In effect, the tile-placement function and the stock-purchase function pair must coordinate with each other to maximize the winning potential of the agent using them. To avoid the credit assignment problem, one would evolve the two functions as a unit, to avoid the later problem of figuring out how to combine them.

Nevertheless, before the credit assignment problem emerged as an issue, it was decided to evolve populations of place-tile evaluation functions separately from populations of buy-stock evaluation functions. We provided them both identical terminal sets and function sets, in an attempt to decrease the effect of our human prejudices as to which factors may be important in either decision. Of course, we recognized that, to have a real strategizing Acquire agent, it must include functions for both of decisions to be made in a turn. Therefore, at the conclusion of our GP runs to evolve these decision functions separately, we chose a run for each population at random and also ran an all-pairs competition to see which pair of buy-stock and place-tile functions worked the best together. See Section 4.3 for details.

We used tournament selection because, as is justified by [21], it reduces the number of evaluations needed, and because there is no exhaustive set of test cases for Acquire, making fitness-proportionate selection (selection based on the proportion of fitness cases matched) meaningless. Tournament selection was introduced by Angeline and Pollack [3], in order to reduce the number of competitions needed and to force the population to avoid local minima traps.

3.2.1 Adaptations to the Acquire Domain

Certain adaptations to the original game of Acquire were incorporated into our system, for various reasons, and are discussed below. First, the number of hotel chains in the game is actually seven, but we decreased this number to three, to decrease the number of functions our GP would have to manipulate during evolution. With seven distinct hotels, our terminal set size was greater than 50. We determined that this would unnecessarily tax our system. In addition, because strategies should not be dependent on certain chain *names*, but rather should be based on their current positions on the board (i.e., size and neighbors), we simplified the problem to *largest chain*, *medium chain*, and *smallest-chain*. Evaluation functions based on these parameters rather than hard-coded for chain names are inherently more robust in evolution. It is easy to see that in a certain game where `CHAIN-1` may be the largest and the agent's evaluation function has a reference to its size, the agent may do well and advance to the next generation, but then suddenly do much more poorly in the next game because the circumstances are very different.

Second, we reduced the game to a one-on-one competition rather than allowing multiple opponents to play against the evolving agents. This was done to reduce the time and computational complexity of playing through one game. Since the fitness evaluation of each agent requires 10 games to be played, it was essential that each game not be a bottleneck for system performance. Reducing the number of players in the game changes the dynamics of a human-played game, as well as affecting the total amount of money possible to win (but not decreasing the validity of measuring a ratio). The multi-player version of this game is a much more complex, chaotic problem that could be interesting to study in future work in this area.

Another simplification involved the other, more minor in-game choices a player may sometimes have to make. When a chain is created, the player is allowed to choose which one it will be; when two chains of equal size merge, the player is allowed to choose which one will triumph; when certain end-game conditions are met, the player is allowed to

choose to end the game or not. Because of our desire to focus on the core of the strategy of this complex domain, these minor decisions are relatively unimportant, and in the current system, the engine performs a random choice of the available options. We did not want to have to evolve behaviors for dealing with these other three minor decisions as well, since only the place-tile and buy-stock decisions are the driving force behind the game progress. While it can be argued that the three minor choices above have the potential to turn the tide of a game, the extent to which this is possible was deemed inconsequential to the overall strategy.

3.2.2 Adaptations to Koza's Genetic Programming Kernel

To support the execution of our evolutionary experiments, we also made a few changes to Koza's GP kernel. In our preliminary experiments, we experienced a lack of progress due to certain issues. One problem was that potentially good individuals were often lost as time went by because the best individuals were not guaranteed to survive to the next generation. This is acceptable for fixed-fitness-based domains such as symbolic regression, or deterministic games, like the examples in Koza's book [23], because, when an individual is found which satisfies all fitness cases, it can be recognized as *the* optimum, or a functional equivalent. However, for Acquire, it can be argued, as for many other complex strategy games, that there exists no optimum strategy. Therefore, we want to encourage the promotion of evaluation functions which do very well against others, in case that a particularly strong strategy is discovered in a very early generation. To this end, we implemented a version of forced elitism within Koza's kernel, which did not otherwise support it. We saved the best 2% individuals from each generation and added them to the next generation's population, unchanged.

The second change we made to Koza's kernel was to add a simplified version of coevolution, which was also not supported by his code. In this version, to determine an individual's fitness, we chose 10 individuals at random (without replacement) from the population

and played the games against these agents. Since no optimal strategy was known, and since hard-coding strategies only leads to the evolution of brittle, inflexible strategies capable of exploiting the weaknesses in the provided opponents but not others [31, 27], coevolution was the logical alternative. To reduce the number of evaluations we needed to perform, both players receive credit for the game, even the individual not currently being evaluated by the GP system. This meant that some players never had to enter an evaluation loop, as they had already played 10 games against other players during *their* evaluation loops.

Chapter 4: Experimental Results

4.1 Genetic Programming Runs

We ran several preliminary experiment sets, to try to determine what set of parameters gave us the best up-slope in the early generations. After this, we performed the actual GP runs. We ran 10 experiments for each decision population (the Place-Tile function and the Buy-Stock function), and analyzed the means over these 10 runs. This redundancy was to allow us to more effectively argue for any trend line that may have resulted, and to eliminate the effect of noise and domain stochasticity on the results. Preliminary results indicated that the two decision populations varied from each other; the shape and magnitude of the curves differ enough to show the two populations have evolved separately. See Figures 4.1 and 4.2 for illustrations of the differences in individuals' size between the two decision populations. This shows that, although the two populations were provided with the same terminal and nonterminal sets, they evolved very differently due to the nature of the problem to which the individuals were applied.

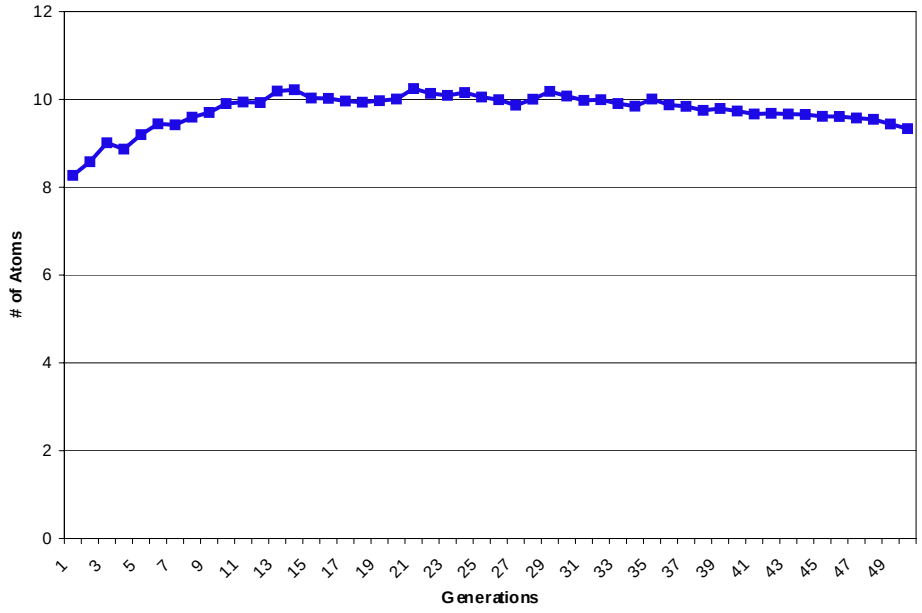

Figure 4.1: Average Size of Individuals in each Generation for Place-Tile Population, Mean Over 10 Runs

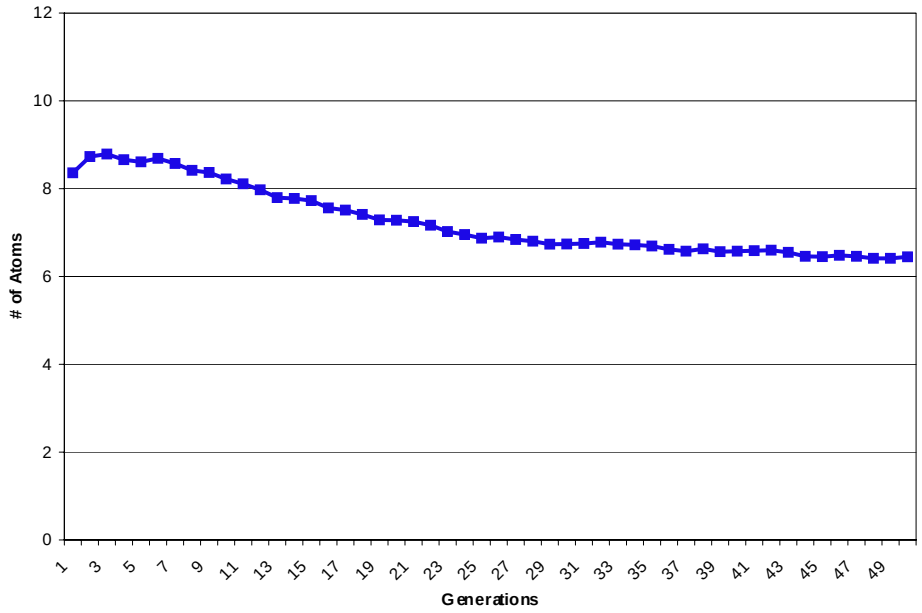

Figure 4.2: Average Size of Individuals in each Generation for Buy-Stock Population, Mean Over 10 Runs

4.1.1 Place-Tile Decision Population Evolution

The first graph for the Place-Tile decision population is shown in Figure 4.3. It shows the mean over 10 runs of the performance of the best individual from each generation (not necessarily the same individual from one generation to the next) over time, as a function of the fitness metric used to guide evolution: the ratio of money earned by the player in his games compared to the total amount of money earned by both players in these games. A *higher* number in this case reflects a more fit individual; for instance, a ratio of 0.75 represents an individual which is keeping a good majority of the profits available in a game for himself.

We hope that, as time goes on, the strategies are able to achieve higher levels of income in a game, as a result of more intelligent stock purchases and the effecting of timely mergers. The data from any one run is quite noisy, which can be seen from the scatterplot shown in Figure 4.5, probably due to the random nature of the Acquire game domain (randomness forces other areas of the search space to be explored, forcing strategies toward more generality and penalizing those which are too specific). The cloud shown in the scatterplot illustrates that there are no outlying runs obscuring trends in the data overall. Analysis of the means plotted in Figure 4.3 is not much more promising; there is a very slight upward trend as the generations progress, but even the means are still showing a lot of noise between generations. Note the small scale used on the y-axis in this graph; although the trend is increasing, it is extremely gradual, implying that more generations may be needed.

Another metric of progress we considered was the average standardized fitness for the population in each generation as reported by the GP system during the run. As above, recall that raw fitness is the average ratio of earnings of an individual in a game to the total earnings in that game by both players, and that the fitness is then standardized by subtracting it from 1.0, meaning that *lower* numbers reflect more fit individuals. The graph shown in Figure 4.7 displays the means over 10 runs of the average standardized fitness over all individuals in a population for each generation. Again, the means graph still suffers

from noise, and here the downward trend is not as steep (note that the scales are different between the two figures).

These results are discouraging, because one would hope that any noise present at varying points throughout the runs would be counteracted via using the means analysis of several runs. That the jagged lines still occur suggest that the populations are evolving in very similar fashions throughout each run, across runs. Each run's own data does not in fact have the same shape, so the runs are not exact replicas of each other, but they do appear to exhibit parallel behavior across runs in the same generations. Further analysis of these results is left to Section 4.2.

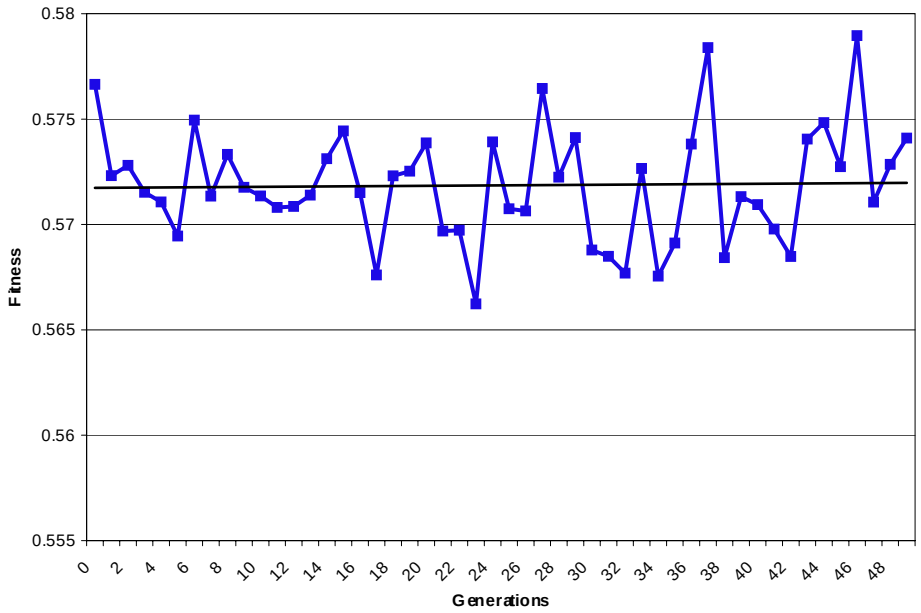

Figure 4.3: Performance of the Best Individual from Each Generation for Place-Tile Population, Mean Over 10 Runs *(Note: graph is analyzed in Section 4.1.1.)*

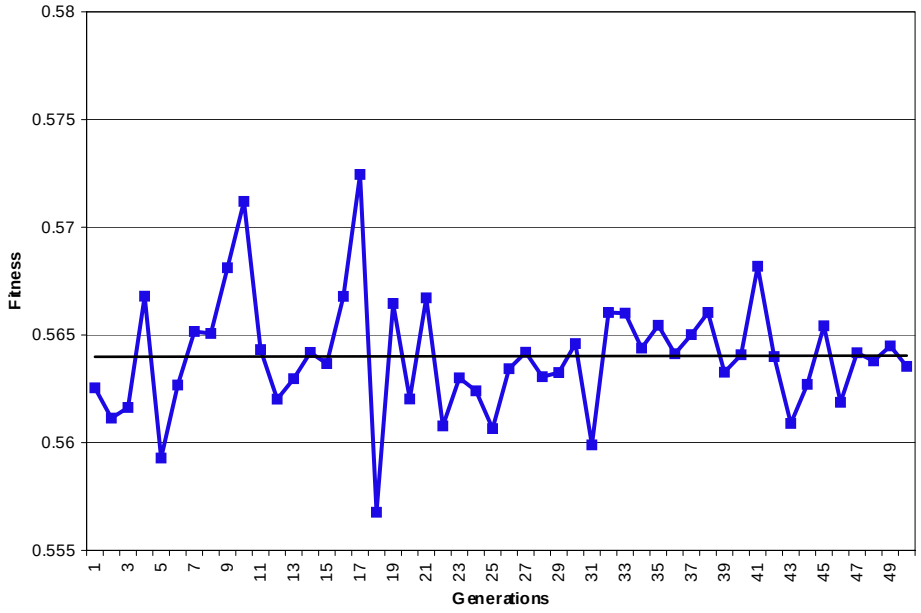

Figure 4.4: Performance of the Best Individual from each Generation for Buy-Stock decision, Mean Over 10 Runs *(Note: graph is analyzed in Section 4.1.2.)*

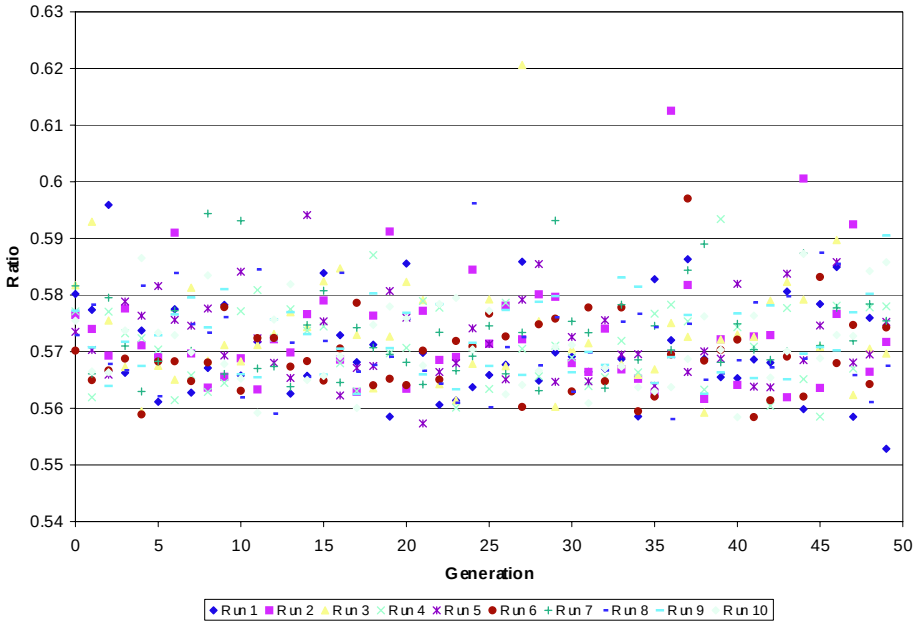

Figure 4.5: Scatterplot of the Fitness Ratio of the Best Individual over Time for Each Run for Place-Tile Population *(Note: graph is analyzed in Section 4.1.1.)*

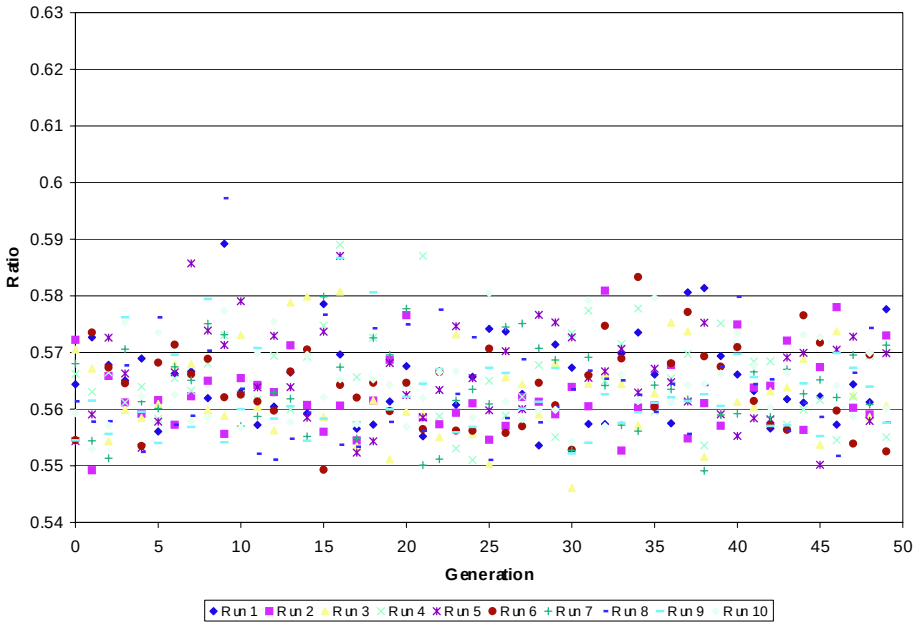

Figure 4.6: Scatterplot of the Fitness Ratio of the Best Individual over Time for Each Run for Buy-Stock Population *(Note: graph is analyzed in Section 4.1.2.)*

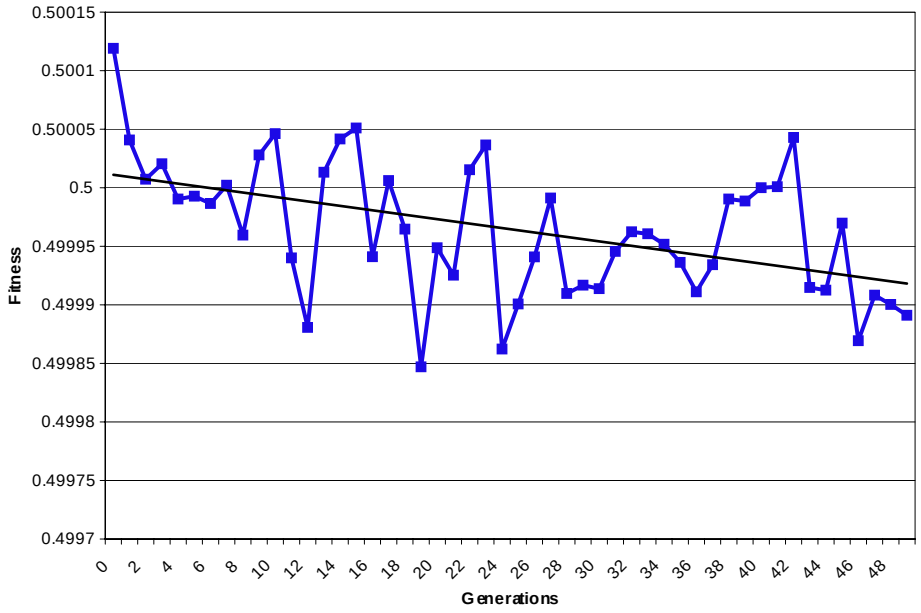

Figure 4.7: Average Standardized Fitness of each Generation for Place-Tile Population, Mean Over 10 Runs *(Note: graph is analyzed in Section 4.1.1.)*

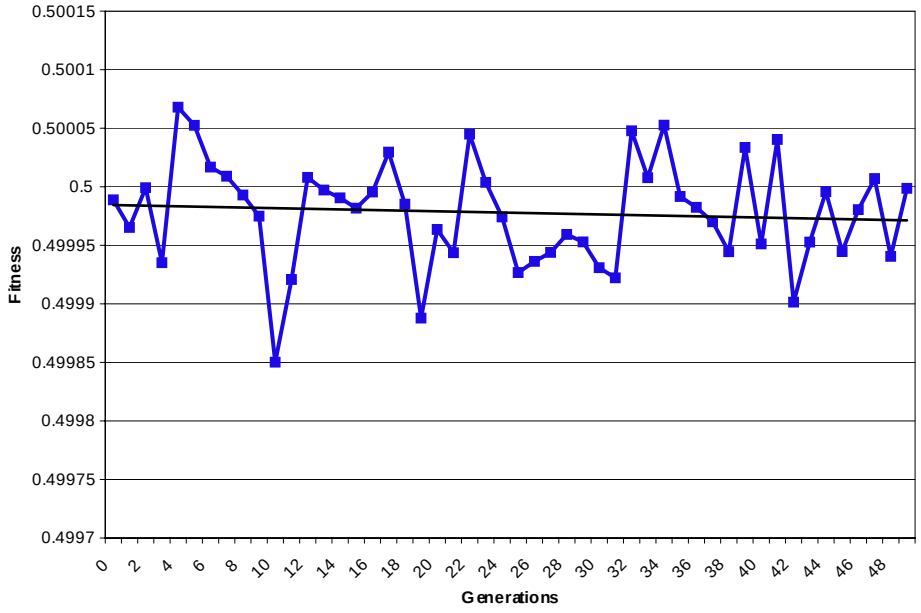

Figure 4.8: Average Standardized Fitness of each Generation for Buy-Stock Population, Mean Over 10 Runs *(Note: graph is analyzed in Section 4.1.2.)*

4.1.2 Buy-Stock Decision Population Evolution

The performance of the best individual in each generation for the Buy-Stock decision population over time is shown in Figure 4.4. This graph is even less encouraging than that for the Place-Tile population because, in addition to the noise still interfering, the trend line extrapolated from this data does not even have a positive slope, but is rather much more flat. Figure 4.6 again shows the scatterplot of each individual run's data. The cloud again is stable, showing that no one run acts as an outlier confusing the means, but rather each run behaves in approximately the same manner as the others. The graph of the average standardized fitness of the population over time is shown in Figure 4.7. The graph's noise makes it difficult to see a trend, but it appears to be also averaging out to a level slope. Again, note the small scale of the y-axes in these figures.

While discouraging from a superficial (i.e., shape of the graph only) standpoint, the results from this and the Place-Tile population do not necessarily show absolutely no improvement of the populations, however, as discussed in Section 2.2. We deal with the issues of coevolution and how it can affect the fitness landscape with further experiments outlined in Section 4.2.

4.1.3 Case Studies of Individuals

In a randomly-chosen Place-Tile evolution run, the best-of-run individual program was found on generation 37. It was:

```
(- (STOCK-HELD-BY-PLAYER-IN-CHAIN-3)
   (+ (+ (IF-I-AM-MAJ-STOCKHOLDER-IN-CHAIN-2 9
           (MERGER-VALUE-OF-CHAIN-3))
         (- (PRICE-OF-CHAIN-1) (STOCKS-LEFT-OF-CHAIN-1)))
      (IF-CHAIN-3-IS-OF-MAXIMUM-SIZE
        (- (PRICE-OF-CHAIN-1) (STOCKS-LEFT-OF-CHAIN-1))
        (STOCKS-LEFT-OF-CHAIN-1))))
```

This parse tree can be rewritten in the following English words: “If I am the current majority stockholder in the middle chain and if the smallest chain is large enough to satisfy the end-game condition, the value of the board state is given by the difference between the quantity of stock I hold in the smallest chain and twice the difference between the price of the largest chain and the number of stocks left in the largest chain to buy, plus a constant (9). If the smallest chain is *not* large enough, the value is the difference between the quantity of stock I hold in the smallest chain and the sum of the price of the largest chain plus a constant (9). If I am *not* the majority stockholder in the middle chain, the constant in the above conditions is replaced by the merger value of the smallest chain (the majority stockholder’s bonus, guaranteed to be more than 9).”

This can be further analyzed as follows. If the smallest chain is large enough to end the game, that means the other chains must be also at least as large. Given that the smallest chain must be of size 41 to meet the end-game condition, the other two chains must also be at least of size 41, which makes 123 tiles belonging to the three chains. However, there are only 108 squares on an Acquire board. Therefore this piece of the strategy will never apply. Moving to the case where the smallest chain is *not* yet large enough to end the game, the value of the board is derived by subtracting the price of the largest chain plus the constant 9 from the amount of stock the player currently holds in the smallest chain. If the player is not currently the majority stockholder in the middle chain, the value is determined by subtracting the price of the largest chain plus the merger value of the smallest chain from the amount of stock the player currently holds in the smallest chain. These seems to create a drive toward merging the smallest and largest chains if the player does not stand to gain a great deal of money by merging the largest and middle chains (i.e., he is not the majority stockholder in the middle chain). When a merger occurs between the smallest and largest chains, the player will be forced to trade in his stock in the smallest chain for stock in the largest chain. Therefore, the amount of money he stands to gain from this transaction is directly given by the amount of stock he owns in the smallest chain and the current price of

the largest chain, which would be the value of the stock he receives during his trade-in.

In a randomly chosen Buy-Stock evolution run, the best-of-run individual program was found on generation 44. It was:

```
(IF-CHAIN-2-IS-SAFE-FROM-MERGING
  (+ 7
     (IF-CHAIN-3-IS-SAFE-FROM-MERGING 2
       (STOCK-HELD-BY-PLAYER-IN-CHAIN-1)))
  (IF-CHAIN-3-IS-OF-MAXIMUM-SIZE 8
    (IF-CHAIN-3-IS-SAFE-FROM-MERGING (PERIMETER-OF-CHAIN-2)
      (DIST-TO-NEAREST-NEIGHBOR-OF-CHAIN-3))))
```

This parse tree can be rewritten in the following English words: “The value of any given board state resulting from my available stock purchases is dependent upon whether the middle chain is safe from merging. If it is, the value is 9 if the smallest chain is also safe from merging, or 7 plus the amount of stock I hold in the largest chain. If it is not, the value is 8 if the smallest chain is large enough to cause the end of game conditions to be met, otherwise it is the growth potential of the middle chain if the smallest chain is safe from merging, and the unlikelihood of an immediate merger of the smallest chain if it is *not* safe from merging.”

This individual can be further analyzed as follows. The purpose of this decision is to determine which chain to purchase stock in during this turn. If the middle chain is safe from merging, and the smallest chain is also safe from merging, this entails that the largest chain is also safe from merging (of size 11 or greater). In this case, the end-game conditions have been met and the game could end at any time; a player at this point only stands to gain money based on the stocks left in his hand which he will sell at the end of the game. The case where the smallest chain is not yet safe from merging means that it could still be merged, and this strategy seems to encode a preference for increasing the amount of stock held in the largest chain, which will pay off if the smaller chain grows or is swallowed up

and the game ends, forcing the players to sell their stock. The largest chain's stock price will be very high if it is that large. The case where the middle chain is not yet safe from merging and the smallest chain does satisfy the end-game condition is not possible, and therefore this branch of the strategy is unused.

However, if the middle chain is not yet safe from merging, the value of the board depends on the growth potential of the middle chain if the smallest chain is also safe from merging (and therefore, the end of the game could occur at any time). This implies that the strategy will prefer investing in the middle chain in this case, which may be more affordable than investing in the largest chain, and may be made larger in future turns (growth potential). Finally, if the middle chain is not yet safe from merging, and the smallest chain is *not*, the smallest chain could still merge with the larger or middle chain, and therefore the immediacy of this potential merger determines the value of the board. This seems to push for investing in the smallest chain until it merges, in hopes of becoming the majority stockholder and receiving that payoff.

These both may seem complex, but that is representative of the domain of Acquire. These strategies make sense in the context of the game. Other strategies which performed well also had similar structures and complexity. The functions do an adequate job of identifying the most profitable situations and rating them highly, which is, ultimately, the purpose of the board evaluation function in the first place. While these strategies may not capture all of the aspects of an Acquire game, the fact that they have been able to identify the situations they have is promising. These results in themselves justify the purpose of this thesis, which was to determine if GP could yield competent Acquire players which learn how to exploit the rules of the game to make money for themselves.

4.2 Measuring Coevolutionary Adaptive Progress

Section 2.2 mentions the so-called "Red Queen effect" which alters coevolutionary interactions: the populations, by nature of their interaction, alter the fitness landscape. Various

measurement and visualization techniques have been developed to account for this effect when reporting on genetic programming runs, also discussed in Section 2.2, and we have adopted a simplified version of one of them here, to attempt to ascertain if any remarkable progress is occurring during our evolutionary runs at all. [9] and [36] created pictographs showing the outcomes of round-robin tournaments of the best from each generation against samples from every other generation; the expectation is that, the more "ancient" an opponent, the better current individuals should do when pitted against them. To ensure monotonicity of the results, it is essential to test later generation players against earlier ones. [5] used random samplings of players from each generation against benchmark players from varying epochs of generations, to test if the population was truly evolving continuously toward an optimum or simply getting stuck in local optima. The approach we took was similar but less complex: we played the best individual from the final generation against the best individual (according to standardized fitness) from each previous generation, where each pair was played against each other 10 times.

Since we had only one set of data points, we simply plotted the matches as a line graph (of course, we used the means over all 10 runs). See Figure 4.9 and 4.10 for the graphs for each decision population. In the Place-Tile population's graph, there is still the noise issue, even after sampling over the mean. Yet there is a clear downward trend line, in spite of this noise. This downward trend means that, as the generations progress back from generation 49 to generation 0, the final generation's best player is capable of beating them more often, implying that there is in fact progress of the population as a whole. Yet in the Buy-Stock population graph, the trend is flat and only starts to be negative toward the end of the runs, as is shown by the moving average line. It is unclear whether these results are definitive enough to recommend this technique as it is as a reliable method of evolving these sorts of board evaluation functions. We would like to see a much more obvious and consistent trend throughout the generations. Areas for exploration into possible improvements on this technique are discussed in Section 5.

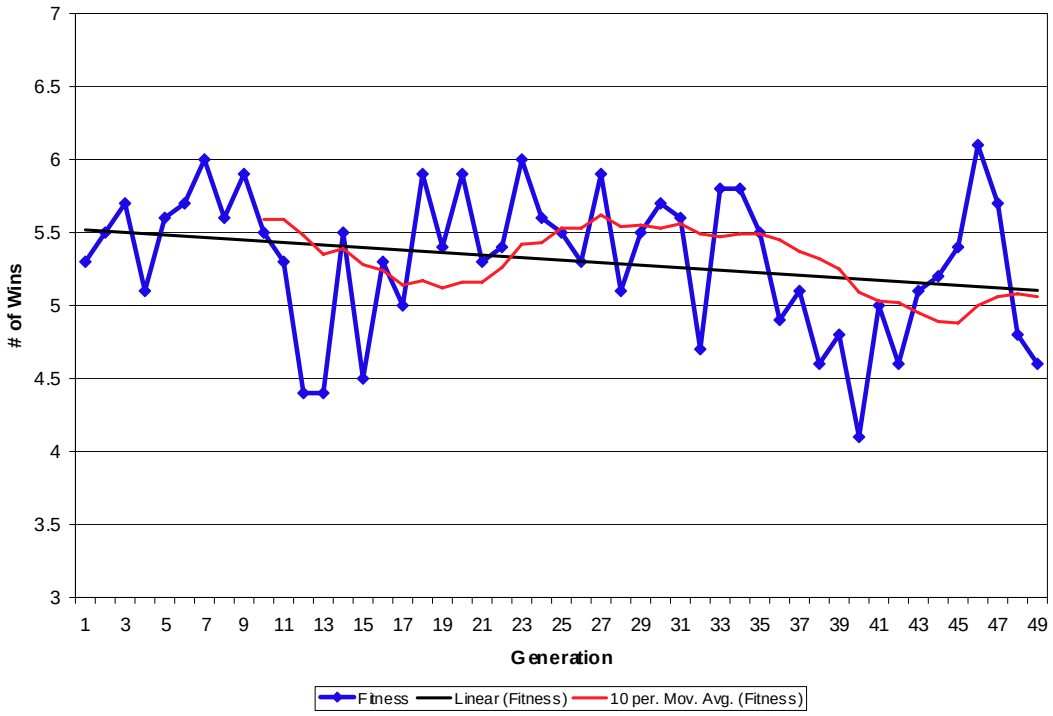

Figure 4.9: Results from the "Tournament of Bests", as represented by the Number of Wins Recorded for Place-Tile Population, Mean Over 10 Runs *(Note: graph is analyzed in Section 4.2.)*

4.3 Obtaining a Combined Strategy

Because we did not evolve the two functions simultaneously (see Section 5.3 for a discussion of this), and because a competent Acquire player must concentrate on both aspects of gameplay in order to perform well, the final experiment we conducted was a simple all-pairs competition, using the reports on the GP runs of the best individual from each generation. We chose the same run we had chosen at random for each population in Section 4.1.3, to see how a sample of the best functions from each population would rank when paired with each other. The competition matched every pair of the best individuals from each generation (the best individuals from certain generations may be the same or functionally equivalent) and played 10 games against a random sampling of other pairs. The pair with the best score (as represented by the number of wins) is considered to be the strongest

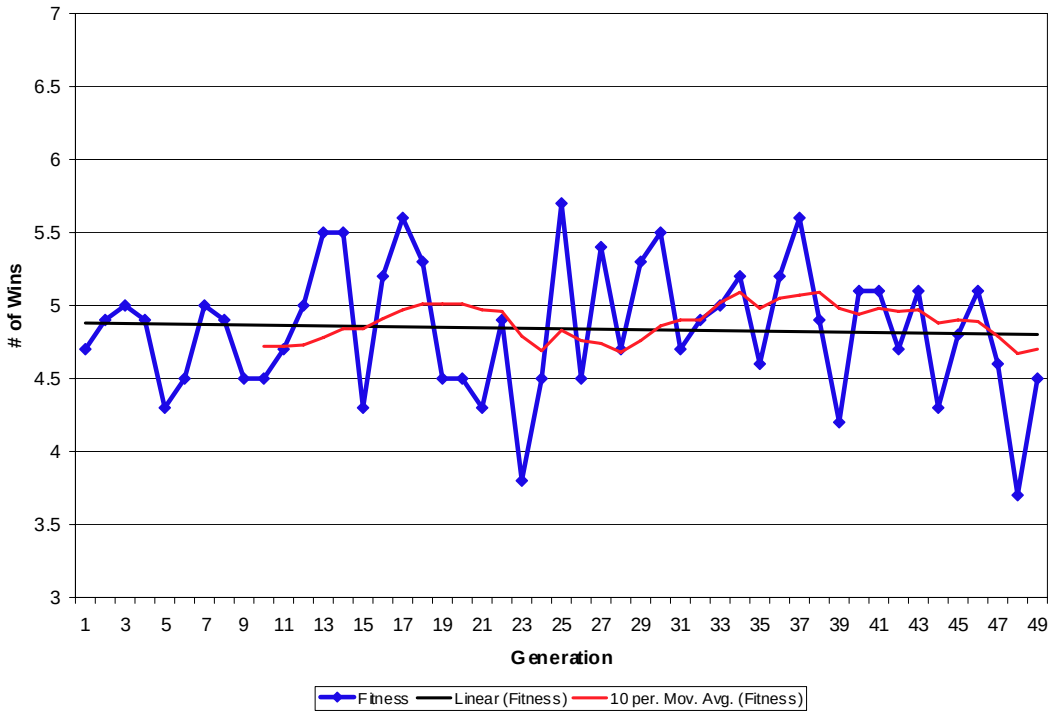

Figure 4.10: Results from the "Tournament of Bests", as represented by the Number of Wins Recorded for Buy-Stock Population, Mean Over 10 Runs *(Note: graph is analyzed in Section 4.2.)*

Acquire player which we have been able to develop through the technique described in this thesis.

There were four pairs that won all 10 games played against their opponents; they were from generations 17 and 45, 19 and 0, 38 and 47, and 46 and 32, where the first number is the generation of the Place-Tile individual chosen and the second number is the generation of the Buy-Stock individual chosen. For the programs of these pairs, see Appendix B. We matched these four pairs against four intermediate human Acquire players, with comparably the same amount of experience with the game, and had them play five games. While it is not possible to draw firm conclusions about the results of the evolution from this alone, it is interesting to note that the agents did in fact win about half the games they played against the humans. The first player lost once, the second and third players lost twice, and the fourth player actually lost four times, against the agent players. This shows a great

deal of promise for the evolution of competent Acquire players via this technique. While the players evolved may not have approached expert level (shown by their apparent lack of overall population progress), the building blocks to produce good novice players appear to have been available through this technique.

Ironically, the best-of-run individuals (from generations 37 for Place-Tile and 44 for Buy-Stock) do not appear in these four top-ranked pairs. The first occurrence of the Place-Tile individual from generation 37 in the pair rankings is tied with 181 other pairs with 8 wins; the first occurrence of the Buy-Stock individual from generation 44 in the pair rankings is in another pair in that bracket with 8 wins. This implies that evolving them separately leads to mediocre results, and that there is a greater degree of intertwining of the individuals than we had initially foreseen. This point is a strong recommending factor for the evolution of Place-Tile/Buy-Stock pairs in the same population, rather than as individuals. See Section 5.3 for more discussion on this topic.

Chapter 5: Conclusions

5.1 Contributions

This work has potentially served to further the evidence that genetic programming is a viable method to use in the development of strategies or board evaluation functions for complex domains such as Acquire, granted some much-needed investigations into the technique used and the application of various related methods described in the literature on the subject. We have managed to evolve novice Acquire players using GP techniques; the players may never have passed a certain level of expertise, but they could hold their own in test games against humans. We have shown that this technique has promise for this domain, and have opened the doors for further future work in this area.

5.2 Limitations

Time was a serious limiting factor in the breadth of the experiments we were able to conduct. There are many fascinating issues in GP which would have been interesting to explore. The inability for us to be able to make claims about the objective efficacy of the strategies developed by our GP runs is a limitation which we expected, as our goal in performing this work was not to create a world-class Acquire player, but simply to see how possible it was to evolve players for this complex domain in the first place. One observation which we *can* make, however, is that, through a series of five games with a group of novice players and one expert (the author), monetary gains of the players did in fact hover around the amounts won by the evolved strategies in their own games, when the human players played with the same set of restrictions on the rules of the game as the agents did.

There are many areas of this work which could be improved, and admittedly, we did not have fully rationalized reasons for the various GP parameter values we used, other than that they were Koza's defaults. However, our domain is significantly different from symbolic

regression problems, and therefore these parameters may very well be poor choices. Much literature refers to the problem of parameter-setting as a “black art” [11]. [14, 11] both discuss the problems inherent in how to choose technique-specific and problem-specific parameters, such as crossover and mutation ratios, population size, et cetera. It is pointed out that they are highly interdependent, and that changing one may have effects on the GP run that are unexpected, due to this interdependence. Parameter control, or tuning of the parameters during the actual evolutionary run, may provide some better way to guess-and-check parameter settings. However, this is very computationally expensive in itself, and may not yield results [11]. The parameter setting conundrum requires many more empirical results than we quote here to set them in a definitive fashion.

Another limitation was the simplistic nature of the nonterminal set used. For complex strategies, more robust decision-making is required. There were several logical and decision operators which we would have liked to include, such as `and`, `or`, `not` and `if`. These are known to be useful in evolving decision-making strategies, and there is support in the literature for their inclusion when evolving strategies or board evaluation functions [16, 15]. Reasons why we did not choose to include them in our approach are discussed in 5.3. Future approaches to this problem, and ones like it, will benefit from an exploration of a more complex function set.

5.3 Related and Future Work

5.3.1 Richer Terminal and Nonterminal Sets (Strongly-Typed Genetic Programming)

As mentioned above, the functions chosen for the nonterminal set were more simple than this problem may potentially require. However, Koza's GP kernel which we used has the unfortunate requirement of "closure": all functions must be able to take as their arguments any terminal or return type from other functions which may result, and the root node must return the expected type for the program [23]. While this is feasible for symbolic regression problems, decision problems are too severely limited by this constraint. It would require rewriting of the decision operators, in our case to accept numeric arguments and return numeric arguments, which is not feasible and sidesteps their advantages as decision operators in the first place. In Koza's examples, the non-hamstrung squad car problem (a variation of the predator-prey game) for which he evolves strategies in Appendix B of his 1992 book [23] uses several conditional functions which have been tailored to his domain in order to satisfy this constraint (e.g., `ifX` which takes three arguments, determining what to do when `x` is less than 0, equal to 0 or greater than 0). Given that our function set is already so large as a result of the complexity of the domain and the amount of information available to an agent during gameplay, the number of conditional operators would be unacceptably large.

This problem has been discussed in the literature, however, and as a result of this limitation of Koza's technique for more complex domains, Montana developed "strongly-typed genetic programming" (STGP) [30], which operates as an extension to the original GP paradigm, but allows the return types and argument types of all functions and terminals to be specified along with each terminal and nonterminal. This not only allows elements such as logic and decision operators to be included without being specifically tailored to the domain or the problem, but it can cut down on the size of the terminal and nonterminal set as well, through the use of generic functions. While Montana implemented his extensions in C++, Koza's LISP kernel could be extended itself to provide STGP via the CommonLISP

Object System (CLOS). This would relieve some of the computational stress of generating the population and the possible dilution of the GP technique due to the very large function sets. For example, the function set may be so large that the GP system cannot fully explore the space of possible individuals for a given population size. Our population size of 1000 hopefully relieved that issue to a degree by allowing many more individuals to be generated. With strongly-typed genetic programming, on the other hand, the functions which must now be specific to each hotel chain in the game could be rewritten as generic functions and the STGP system could evolve the correct arguments. This would allow all seven chains to be used instead of the three-chain simplification we opted to use in these experiments, and which undoubtedly reduced the amount of money available to be won by the evolving agents.

5.3.2 Evolution of a Unified Strategy for Acquire via Structurally Constrained GP, Dynamic GP and the Evolution of Lambda Functions

As mentioned in Section 3.2, we ran an all-pairs competition to see which pair of buy-stock and place-tile functions worked the best together. This was purely for curiosity's sake, and the semi-round-robin tournament as played does not provide an adequate means of determining the best pair, compared to the possibility of evolving the two functions simultaneously. The credit assignment problem prevents the two strategies from being evolved as individuals in separate populations from being a workable solution. The ideal way to pursue this problem in future would be to adopt a form of structurally-constrained GP, which forces the individuals being evolved to have two parts, the order of which is predetermined for ease-of-use within the Acquire system. Then, the individuals are marked as fit or unfit relative to the population as a whole, rather than as a result of either half of the strategy. In this way, the credit assignment problem is avoided, but a fully-integrated agent strategy can be evolved.

Although our goal in providing the large terminal and nonterminal sets that we did was to avoid biasing our results with preconceived notions from the authors about what would

make a good strategy, it is of course possible that we still may have inadvertently done so, especially when one considers the choice of conditional operators we made. With this potential limitation in mind, several sources in the literature discuss allowing the GP system to evolve lambda functions [34] or higher-level functions and their sub-behaviors [2, 32], in addition to being given certain discrete terminals and nonterminals. Since it seems likely that the function set we have provided for the Acquire domain is in fact too limited to allow innovative and robust strategies to develop, incorporating this idea of dynamic GP may ameliorate the results by allowing the GP system to explore areas of the search space that the authors may not have recognized as useful or necessary at the time of problem formulation.

A tangent to this idea is the fact that GP operators tend to promote convergence, causing all individuals to become more similar over time [22]. GP systems tend to require large populations of individuals to ensure a sufficiently wide sample of the possible search space, so that encountering appropriately fit individuals is more likely. Therefore, our problem specification used a population size of 1000. [22] uses an operator called “splitting” which those authors claim actually discourages convergence; such an approach may help in the case where Acquire strategies which are evolved are not allowed to become complex enough because not enough of the search space is explored.

5.3.3 More Comprehensive Measures of Competitive Coevolution

Enforcing the criteria that new individuals must always be able to defeat previous individuals will ensure that the population is always progressing, and not stagnating in a local minimum [36]. Building beyond our very simplistic "tournament of bests", a "hall of fame" of sorts can be used to test current individuals against a sample of the past individuals which were best in their respective generations; a second possibility is to force the fitness of an individual to be based on its ability to beat the best individual from the previous generation [39].

Perhaps all that is needed is the use of better visualization techniques to monitor how the fitness landscape is changing over time and, therefore, to judge the degree of progress within the population. Several methods have been used in the literature [9, 35, 36, 24]: tournaments of the population champions against all previous champions ("ancestral opponent contests") and distance metrics using similarity of the parse trees developed ("genetic distance measures"). Although we did use a simplified version of the ancestral opponent contests in our approach (Section 4.2), the more complete ones in the literature provide a better global perspective on the population over time. When applied to results from Acquire GP runs, perhaps more definitive patterns of progress would emerge.

5.3.4 Annealing Schedule of Fitness Metrics as Population Evolves

As was discussed in Section 4.2, a problem with determining progress in competitive coevolution is that, as the population progresses through the generations, the coevolution alters the fitness landscape as the population as a whole improves. In later generations, when many of the individuals are similar in overall fitness, but still compete against each other, it will take more games to determine which of two very close competitors is actually better. Some studies have therefore used an "annealing schedule" of fitness functions [31]. In this model, the number of games played is increased as varying epochs of generations are reached (i.e., 10 games from 0-100, 15 games from 100-300, 20 games from 300 on).

Unfortunately, this annealing schedule will be highly domain- and problem-dependent, and literature which discusses it also states that the schedule was the result of preliminary empirical results [31]. To apply this to Acquire would require further investigation into the progress of our GP runs.

5.3.5 Use of Internal State to Promote Learning Through Experience

Some literature also makes reference to the possible usefulness of a notion of internal state throughout the GP run, to allow the individuals in the population to learn via experience what strategies may have more promise [25, 28]. However, it seems that this is most useful in static environments where the evolving strategies can operate on the same configuration of the world in each iteration, such as in the wumpus world [42, 41] or in the Map-maker/Map-user domain of [1]. Agents operating in the Acquire domain may be able to profit from these methods, even though the world is highly variable from game to game, because the rules of the game and dynamics of the domain remain the same. A strategy which does well in one game may do poorly in another and be mutated in some fashion, causing it to no longer perform well in the game scenarios it initially encountered, because they are no longer accessible. Guiding the evolution process aided by learning through experience with past game situations may yield more robust individuals in the long run, capable of handling many strategic nuances.

5.3.6 Changing the Representation to Neural Networks

Much work has been done where strategies for games or board evaluation functions are evolved using neural networks as the representation. The representation of the problem is a limiting factor on how well a technique will do in solving a particular problem; linear functions may not be complex enough to yield appropriately complex strategies for domains like Acquire. Neural networks, as nonlinear functions, are much more flexible and capable of abstracting the behavior of a strategy [7]. Neural networks have already been applied

to the games of Checkers [38, 8, 6], Backgammon [43, 31], Go [24], Tic-Tac-Toe [7], and others. This avenue may hold promise for the evolution of more adaptive, more robust, and stronger Acquire strategies or board evaluation functions. As [7] states, "it is a fundamental problem to treat non-linear problems as if they were linear" (p. 3). Perhaps that was our mistake here.

5.4 Summary

This thesis presented work which applied genetic programming techniques to the evolution of board evaluation functions for a complex strategy game called Acquire. We demonstrated that it is in fact possible to develop strategies for the Acquire domain using these techniques, but that these techniques have a long way to go before they can produce players on a high level of competence. Evolution was not encouraged enough during our experiments, and for this reason, further exploration into modified techniques is needed. Our experiments have laid the groundwork for future studies in these areas, by expanding the current experiments to include more runs, a richer function set, and ultimately applying these techniques to more complex domains than Tic-Tac-Toe and Othello. The problem is by no means solved—there are many avenues which were only touched upon during our work. It is our hope that these efforts open up an interest in non-minimax games like Acquire, and lead to the possibility of further explorations of these topics.

Bibliography

[1] David Andre. The automatic programming of agents that learn mental models and create simple plans of action. In *Proceedings of the 1995 International Joint Conferences on Artificial Intelligence*, pages 741–747, 1995.

[2] P. J. Angeline and J. B. Pollack. Coevolving high-level representations. In Christopher G. Langton, editor, *Artificial Life III*, volume XVII, pages 55–71, Santa Fe, New Mexico, 1994. Addison-Wesley.

[3] Peter J. Angeline and Jordan B. Pollack. Competitive environments evolve better solutions for complex tasks. In *Proceedings of the 5th International Conference on Genetic Algorithms (GA-93)*, pages 264–270, 1993.

[4] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming, an Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1998.

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

[6] Kumar Chellapilla and David B. Fogel. Co-evolving checkers playing programs using only win, lose, or draw. In *SPIE's AeroSense'99: Applications and Science of Computational Intelligence II*, Orlando, Florida, USA, 1999.

[7] Kumar Chellapilla and David B. Fogel. Evolution, neural networks, games, and intelligence. In *Proceedings of the IEEE*, volume 87, pages 1471–1498, September 1999.

[8] Kumar Chellapilla and David B. Fogel. Anaconda defeats hoyle 6-0: A case study competing an evolved checkers program against commercially available software. In *Proceedings of the 2000 Congress on Evolutionary Computation*, pages 857–863, Piscataway, NJ, USA, 2000.

[9] Dave Cliff and Geoffrey F. Miller. Tracking the red queen: Measurements of adaptive progress in co-evolutionary simulations. In *European Conference on Artificial Life*, pages 200–218, 1995.

[10] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *International Conference on Genetic Algorithms and their Applications*, pages 183–187, Pittsburgh, PA, 1985.

[11] P. Darwen. Black magic: Interdependence prevents principled parameter setting. In S. Halloy and T. Williams, editors, *Applied Complexity: From Neural Nets to Managed Landscapes*, pages 227–237. Institute for Food and Crop Research, Christchurch, New Zealand, 2000.

[12] Morton D. Davis. *Game Theory: A Nontechnical Introduction*. Basic Books, Inc., New York, NY, 1983.

[13] Morton D. Davis. *Game Theory: A Nontechnical Introduction*. Dover Publications, Inc., Mineola, NY, 1997.

[14] Ágoston Endre Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *IEEE Trans. on Evolutionary Computation*, 3(2):124–141, 1999.

[15] Gabriel J. Ferrer. Using genetic programming to evolve board evaluation functions. Master's thesis, Department of Computer Science, School of Engineering and Applied Science, University of Virginia, August 1996.

[16] Gabriel J. Ferrer and Worthy N. Martin. Using genetic programming to evolve board evaluation functions. In *IEEE Conference on Evolutionary Computation*, Perth, Australia, 1995.

[17] David B. Fogel. The advantages of evolutionary computation. In D. Lundh, B. Olsson, and A. Narayanan, editors, *Bio-Computing and Emergent Computation 1997*, pages 1–11. World Scientific Press, Singapore, 1997.

[18] Thomas Haynes, Sandip Sen, Dale Schoenefeld, and Roger Wainwright. Evolving a team. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, Cambridge, MA, 1995. AAAI.

[19] Avalon Hill. Acquire. `http://www.avalonhill.com/default.asp?x=games_acquire`, 2000.

[20] John Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1975.

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

[22] Takuya Kojima, Kazuhiro Ueda, and Saburo Nagano. An evolutionary algorithm extended by ecological analogy and its application to the game of go. In *15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 684–689, Nagoya, Japan, 1997.

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

[24] Alex Lubberts and Risto Miikkulainen. Co-evolving a go-playing neural network. In *GECCO-01 Workshop on Coevolution: Turning Adaptive Algorithms upon Themselves*, pages 14–19, San Francisco, CA, 2001.

[25] Sean Luke. Evolving soccerbots: A retrospective. In *Proceedings of the 12th Annual Conference of the Japanese Society for Artificial Intelligence*, 1998.

[26] Sean Luke. Genetic programming produced competitive soccer softbot teams for robocup97. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222, University of Wisconsin, Madison, Wisconsin, USA, 1998. Morgan Kaufmann.

[27] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Co-evolving soccer softbot team coordination with genetic programming. In *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997.

[28] Sean Luke and Lee Spector. Evolving teamwork and coordination with genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 150–156, Stanford University, CA, USA, July 1996. MIT Press.

[29] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

[30] David J. Montana. Strongly typed genetic programming. Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, July 1993.

[31] Jordan B. Pollack and Alan D. Blair. Co-evolution in the successful learning of backgammon strategy. *Machine Learning*, 32(1):225–240, 1998.

[32] Mitchell A. Potter, Kenneth A. De Jong, and John J. Grefenstette. A coevolutionary approach to learning sequential decision rules. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 366–372, San Francisco, CA, 1995. Morgan Kaufmann.

[33] Craig W. Reynolds. Competition, coevolution and the game of tag. In R. Brooks and P. Maes, editors, *Artificial Life IV*. MIT Press, 1994.

[34] J. Rosca and D. H. Ballard. Evolution-based discovery of hierarchical behaviors. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*. AAAI / The MIT Press, 1996.

[35] Christopher D. Rosin and Richard K. Belew. Methods for competitive co-evolution: Finding opponents worth beating. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 373–380, San Francisco, CA, 1995. Morgan Kaufmann.

[36] Christopher D. Rosin and Richard K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.

[37] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.

[38] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–219, 1959.

[39] Karl Sims. Evolving 3d morphology and behavior by competition. In R. Brooks and P. Maes, editors, *Artificial Life IV*, pages 28–39. MIT Press, 1994.

[40] John Maynard Smith. *Evolution and the Theory of Games*. Cambridge University Press, Cambridge, UK, 1982. ISBN 0-521-28884-3.

[41] Lee Spector and Sean Luke. Cultural transmission of information in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 209–214, Stanford University, CA, USA, July 1996. MIT Press.

[42] Lee Spector and Sean Luke. Culture enhances the evolvability of cognition. In G. Cottrell, editor, *Proceedings of the Eighteenth Annual Conference of the Cognitive Science Society*, pages 672–677, Mahwah, NJ, 1996. Lawrence Erlbaum Associates.

[43] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[44] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ, 1944. 1st presentation of game theory.

Appendix A Terminals and Nonterminals Used in the Acquire Problem

Table A1: Some Terminals Used in the Acquire GP Problem Formulation

Terminal	**Description**
`(cash-held-by-player)`	this function returns the amount of cash held by the current player as a result of the move being considered (i.e., if the player buys stock, how much money will he have left)

`(dist-to-nearest-neighbor-of-chain-1)`
`(dist-to-nearest-neighbor-of-chain-2)`
`(dist-to-nearest-neighbor-of-chain-3)`
	these functions return the Manhattan distance to the nearest neighboring chain to the respective chain, as a result of the move being considered

`(how-many-opponents)`	this function returns how many opponents the current player is facing

`(merger-value-of-chain-1)`
`(merger-value-of-chain-2)`
`(merger-value-of-chain-3)`
	these functions return the majority stockholder's bonus for the respective chain as a result of the move being considered

`(payoff-value-to-player-of-chain-1)`
`(payoff-value-to-player-of-chain-2)`
`(payoff-value-to-player-of-chain-3)`
	these functions return the payoff to the current player if a merger were to take place (i.e., if he receives a bonus, how much, plus how much money he would receive for selling his stock) of the respective chain, as a result of the move being considered

`(perimeter-of-chain-1)`
`(perimeter-of-chain-2)`
`(perimeter-of-chain-3)`
	these functions return the perimeter measurement of the respective chain on the board (i.e., all adjacent tiles which, if played, would add to the size of the chain), as a result of the move being considered

`(price-of-chain-1)`
`(price-of-chain-2)`
`(price-of-chain-3)`
	these functions return the price of stock in the respective chain as a result of the move being considered

Table A2: Some Terminals Used in the Acquire GP Problem Formulation

Terminal	**Description**

`(size-of-chain-1)`
`(size-of-chain-2)`
`(size-of-chain-3)`
	these functions return the size of the respective chain as a consequence of the move being considered

`(stock-held-by-player-in-chain-1)`
`(stock-held-by-player-in-chain-2)`
`(stock-held-by-player-in-chain-3)`
	these functions return how much stock will be held by the current player in the respective chain as a consequence of the move being considered

`(stocks-left-of-chain-1)`
`(stocks-left-of-chain-2)`
`(stocks-left-of-chain-3)`
	these functions return the number of stocks available to purchase in the respective chain as a result of the move being considered

`0..9`	the integer digits from 0 to 9
`:floating-point-random-constant`	a ephemeral random floating point constant, as per [23]

Table A3: Nonterminals Used in the Acquire GP Problem Formulation

Nonterminal	**Description**
`+`	addition of 2 arguments
`-`	subtraction of 2 arguments
`*`	multiplication of 2 arguments
`div`	protected division function (returns 0 if divide by zero is attempted)

`(if-i-am-maj-stockholder-in-chain-1)`
`(if-i-am-maj-stockholder-in-chain-2)`
`(if-i-am-maj-stockholder-in-chain-3)`
	if the player is the majority stockholder in the respective chain, executes the first argument, otherwise executes the second

`(if-chain-1-is-of-maximum-size)`
`(if-chain-2-is-of-maximum-size)`
`(if-chain-3-is-of-maximum-size)`
	if the respective chain is big enough to meet the endgame condition, executes the first argument, otherwise executes the second

`(if-chain-1-is-safe-from-merging)`
`(if-chain-2-is-safe-from-merging)`
`(if-chain-3-is-safe-from-merging)`
	if the respective chain is too big to be taken over by another chain, executes the first argument, otherwise executes the second

Appendix B Parse Trees of the Top Four Ranked Pairs in the All-Pairs Tournament

- Pair 1—Place-Tile from generation 17:

```
(- (IF-I-AM-MAJ-STOCKHOLDER-IN-CHAIN-2
    (* (DIV (PERIMETER-OF-CHAIN-2)
    (STOCK-HELD-BY-PLAYER-IN-CHAIN-2))
       (SIZE-OF-CHAIN-2))
    (IF-CHAIN-1-IS-OF-MAXIMUM-SIZE
     (IF-CHAIN-3-IS-OF-MAXIMUM-SIZE
      (SIZE-OF-CHAIN-3) (STOCK-HELD-BY-PLAYER-IN-CHAIN-1))
     (IF-I-AM-MAJ-STOCKHOLDER-IN-CHAIN-3
      (PAYOFF-VALUE-TO-PLAYER-OF-CHAIN-2) 2)))
   (+ (+ (IF-I-AM-MAJ-STOCKHOLDER-IN-CHAIN-2 9
  (MERGER-VALUE-OF-CHAIN-3))
 (- (PRICE-OF-CHAIN-1) (STOCKS-LEFT-OF-CHAIN-1)))
      (IF-CHAIN-3-IS-SAFE-FROM-MERGING
       (STOCK-HELD-BY-PLAYER-IN-CHAIN-3)
       (IF-CHAIN-3-IS-OF-MAXIMUM-SIZE
(SIZE-OF-CHAIN-3) (MERGER-VALUE-OF-CHAIN-1)))))
```

—Buy-Stock from generation 45:

```
(IF-CHAIN-3-IS-SAFE-FROM-MERGING
 (IF-CHAIN-2-IS-SAFE-FROM-MERGING
  (IF-CHAIN-1-IS-SAFE-FROM-MERGING  1
   (IF-CHAIN-3-IS-SAFE-FROM-MERGING 2 1))
  (IF-CHAIN-2-IS-OF-MAXIMUM-SIZE 9 (SIZE-OF-CHAIN-2)))
 (+ (IF-CHAIN-3-IS-OF-MAXIMUM-SIZE 1 7)
    (IF-CHAIN-3-IS-SAFE-FROM-MERGING 2 1)))
```

- Pair 2—Place-Tile from generation 19:

```
(- (IF-CHAIN-3-IS-SAFE-FROM-MERGING
    (DIST-TO-NEAREST-NEIGHBOR-OF-CHAIN-3)
    (STOCK-HELD-BY-PLAYER-IN-CHAIN-1))
   (+ (+ (IF-I-AM-MAJ-STOCKHOLDER-IN-CHAIN-2
  (DIST-TO-NEAREST-NEIGHBOR-OF-CHAIN-1)
  (MERGER-VALUE-OF-CHAIN-3))
 (- (PRICE-OF-CHAIN-1)
    (PAYOFF-VALUE-TO-PLAYER-OF-CHAIN-3)))
      (IF-CHAIN-3-IS-SAFE-FROM-MERGING
       (IF-CHAIN-2-IS-SAFE-FROM-MERGING
(DIST-TO-NEAREST-NEIGHBOR-OF-CHAIN-3)
(STOCK-HELD-BY-PLAYER-IN-CHAIN-3))
       (IF-CHAIN-3-IS-OF-MAXIMUM-SIZE
(SIZE-OF-CHAIN-3)
(STOCKS-LEFT-OF-CHAIN-1)))))
```

—Buy-Stock from generation 0:

```
(- (* (STOCK-HELD-BY-PLAYER-IN-CHAIN-1)
      0)
   (IF-CHAIN-3-IS-OF-MAXIMUM-SIZE
    4
    (SIZE-OF-CHAIN-1)))
```

- Pair 3—Place-Tile from generation 38:

```
(- (IF-I-AM-MAJ-STOCKHOLDER-IN-CHAIN-2
    (IF-CHAIN-2-IS-SAFE-FROM-MERGING
     (DIST-TO-NEAREST-NEIGHBOR-OF-CHAIN-3)
     (STOCK-HELD-BY-PLAYER-IN-CHAIN-3))
    (IF-I-AM-MAJ-STOCKHOLDER-IN-CHAIN-2
     (STOCK-HELD-BY-PLAYER-IN-CHAIN-2)
     (PRICE-OF-CHAIN-1)))
   (+ (+ (MERGER-VALUE-OF-CHAIN-3)
 (- (PRICE-OF-CHAIN-1)
    (STOCKS-LEFT-OF-CHAIN-3)))
      (PERIMETER-OF-CHAIN-2)))
```

—Buy-Stock from generation 47:

```
(* (IF-CHAIN-2-IS-SAFE-FROM-MERGING
    (IF-CHAIN-3-IS-SAFE-FROM-MERGING
     (DIST-TO-NEAREST-NEIGHBOR-OF-CHAIN-3) 8)
    1)
   (+ (MERGER-VALUE-OF-CHAIN-3) 9))
```

- Pair 4—Place-Tile from generation 46:

```
(- (DIST-TO-NEAREST-NEIGHBOR-OF-CHAIN-1)
   (+ (IF-I-AM-MAJ-STOCKHOLDER-IN-CHAIN-2
       (SIZE-OF-CHAIN-2)
       (MERGER-VALUE-OF-CHAIN-3))
      (IF-CHAIN-3-IS-OF-MAXIMUM-SIZE
       (MERGER-VALUE-OF-CHAIN-3)
       (STOCKS-LEFT-OF-CHAIN-1))))
```

—Buy-Stock from generation 32:

```
(DIV (MERGER-VALUE-OF-CHAIN-2)
     (IF-I-AM-MAJ-STOCKHOLDER-IN-CHAIN-3
      (IF-CHAIN-3-IS-SAFE-FROM-MERGING 2 1)
      (PERIMETER-OF-CHAIN-3)))
```