Using Genetic Programming to Evolve Board Evaluation Functions

A Thesis

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment of the Requirements for the Degree of

Master of Science

Computer Science

by

Gabriel J. Ferrer

August 1996

Approvals

This thesis is submitted in partial fulfillment of the requirements for the degree of

Master of Science

Computer Science

Gabriel J. Ferrer

Approved:

Worthy N. Martin (Advisor)

James P. Cohoon (Chair)

Susan E. Carlson

(Minor Representative)

Accepted by the School of Engineering and Applied Science:

Richard W. Miksad (Dean)

August 1996

Abstract

Computer programs for playing boardgames typically utilize a static board evaluation function to select their moves for each turn. Devising a good board evaluation function is, in general, a difficult problem. Consequently, substantial work has been done on devising methods for automatically generating such functions.

Genetic programming is a variation on the genetic algorithm paradigm wherein solutions to problems are encoded as computer programs. A population of these programs evolves over time. The evolutionary process works by evaluating the quality of each program's solution to the problem at hand. This is referred to as the program's fitness. Programs with higher fitness are more likely to survive and propagate. Thus, over time, the overall fitness of the population should improve.

This work shows that a fairly straightforward application of genetic programming results in the evolution of board evaluation functions which can play strategy games with an appreciable level of skill. The games used for our experiments are the ancient Egyptian game of Senet and the modern game of Othello. This helps to demonstrate the general viability of the approach in two very different game environments.

To my family

Acknowledgements

Many people have provided me directly and indirectly with support and encouragement since I arrived here in Charlottesville two years ago. I'll start off by thanking my research advisor, Worthy Martin, for helping me to develop the ideas and concepts which have resulted in this thesis. His input was invaluable in transforming my rough ideas into the research results I have obtained.

Other faculty I would like to thank include Jim Cohoon and Susan Carlson for being on my thesis committee and providing many helpful suggestions for improving this document, and Gabe Robins for providing me with travel funding so that I could present some of this research at the 1995 IEEE Conference on Evolutionary Computation.

Through Charlie Viles, Joe Ganley graciously provided the LaTeX style files that were used to format this thesis.

I'd like to thank John Kelty for many helpful suggestions during the early versions of the research contained herein.

I'd like to thank all of my friends in the computer science department who have helped to insure that life here is fun and entertaining. I'd particularly like to thank Paco Hope, Sean McCulloch, Mike Nahas, John Regehr, John Karro, Dave Engler, Dave Coppit, Dave Bassett, Craig Chaney, Luis Nakano, Norm Beekwilder, and Chris Oliver, all of whom have been frequent compatriots on lunch and beer expeditions.

Other people I'd like to thank for their friendship over the last two years include Robert Hackenberg, Scott and Debbie Briercheck, Eddie D'Elicio, Robin Prudencio, Jennifer Johnson, and Jackie Emmerling.

Special thanks to God for creating such a fascinating universe.

Finally, I'd like to thank my family. I thank my mother and father for supporting me over the years, through love, friendship, and financial assistance. I thank my grandmother for helping take care of me as I grew up. And I thank my brother Bryan for being my friend throughout the years. It is to my family that this thesis is dedicated.

Contents

List of Figures

1

Introduction

Developing intelligent computer players of strategy games is a problem that AI research has been addressing since the field began. Because excellence in the play of strategy games has often been considered to be a sign of intellectual excellence, some researchers believe that developing an intelligent game player could well be a big step on the road to developing a more generally intelligent machine. This thesis examines using the genetic programming paradigm [13] to evolve board evaluation functions for game playing programs.

A fairly extensive literature on the subject of developing good computer strategy game players in general and developing good board evaluation functions in particular testifies to the difficulty of doing so (examples include [4] [20] [18] [15] [16]). Consequently, designing adaptive systems capable of automatically generating such functions is appealing. The notion of such adaptive systems can be found as early as Samuel's landmark checkers playing program [20] that was able to automatically adjust parameters of its evaluation function as it played in order to improve its performance. A number of other researchers have likewise attempted to develop self-improving board evaluation functions [20] [26] [17] [15] [22] [19] [1].

Genetic programming gives us an adaptive framework that can automatically specify possible board evaluation functions using a natural representation to evolve players using a survival-of-the-fittest mechanism. The formulation given in this work also has the advantage (over a Samuel-like formulation) of allowing the evolution of individuals, i.e., board evaluation functions, that are nonlinear in nature, thus enlarging the range of possible solutions [15] [4].

Our formulation for these functions allows the strategy encoded in each board evaluation function to compete against others for "survival" in a tournament format. The performance of each board evaluation function in the tournament will be quantified as an integer and be called the "fitness" of the strategy. A tournament over a population requires that every individual be a "feasible" solution, i.e., every individual exhibits legal play of the game. Our genetic programming framework maintains the property that every considered "solution" is feasible.

An additional benefit of this problem formulation is that we can incorporate and attempt to improve existing board evaluation functions by specifying them to be members of the initial population. This approach of population seeding can result in the evolution of better fit individuals in a shorter period of time when compared to an initial population consisting entirely of random individuals [9].

This work shows that a fairly straightforward application of genetic programming results in the evolution of board evaluation functions that can play strategy games with an appreciable level of skill. The games used for our experiments are the ancient Egyptian game of Senet and the more modern game of Othello. This helps to demonstrate the general viability of the approach in two very different game environments.

This thesis is organized in the following manner. Overviews of the history of board evaluation functions and of genetic programming are contained in Chapter 2. Discussion of the formulation of finding and improving board evaluation functions and their application to Senet and Othello can be found in Chapter 3. Descriptions of our experiments and the results obtained can be found in Chapter 4. Conclusions and future work are discussed in Chapter 5. The rules of Senet are described in Appendix A. The rules of Othello are described in Appendix B. A detailed discussion of Othello strategy and its implications for computer Othello programs can be found in Appendix C. An overview of other evolutionary approaches to automatically generating boardgame players can be found in Appendix D.

2

Overview

This chapter gives an overview of previous work in the areas of developing static board evaluation functions and in genetic programming.

2.1 The Board Evaluation Function

The time gap between the invention of the electronic computer and the implementation of schemes for computers to play strategy games was fairly short. As early as 1950, C. E. Shannon speculated on the possibility of using a computer to play chess, a revolutionary idea at the time [21]. He was the first to suggest (in print) the use of a board evaluation function in conjunction with a search mechanism. Arthur Samuel first implemented a checker program on the IBM 701 in 1952, recoded it for the IBM 704 in 1954, and demonstrated it on television in 1956. His 1959 paper on the subject [20] was one of the first attempts to devise a credible computer game player for a strategy game, and one of the first to incorporate some form of machine learning.

Samuel isolated several features of the checkers game that he considered to be useful information for deciding where to move. Examples of the features Samuel used include relative piece advantage and positional advantage. These features take on numeric values and are then used as terms in a linear polynomial with coefficients for each term. A board evaluation function in his system consists of 16 terms being selected out of a set of 38 possible terms. (In addition, a term representing the total difference in pieces between the two sides is always in use.)

Samuel devised a machine learning technique involving two tasks: to determine which terms to use and how each term should be weighted. The initial values of the coefficients used get altered slowly over time depending on the performance of the board evaluation function as the game progresses, with changes to the coefficients being made after each game in the training process. The terms themselves are ranked according to how their presence correlates with successful game play, and low ranking terms get replaced every eight games or so, on the average.

An overview of the design issues involved with creating effective board evaluation functions has been given by Hans Berliner, a noted backgammon player and programmer [4]. A good overview of heuristics for playing board games can be found in Barr and Feigenbaum's Handbook of Artificial Intelligence [3]. The international chess master David Levy has also written several books on this subject. His book *Computer Gamesmanship* [16] gives a good overview of the issues involved in developing computer game players and discusses specifics of implementing computer game players for a large number of games.

Among the papers which focus on the application of machine learning concepts to the development of board evaluation functions, one of the best known is the work of Gerald Tesauro on backgammon [26]. He used the $TD(\lambda)$ function developed by Sutton [24] for the training of multi-layer perceptrons to develop a board evaluation function for backgammon that learned strategy for the game by playing against itself hundreds of thousands of times. Each of these games has the perceptron, i.e., a neural network, select moves for both sides at each step. A move selection is made by having the network score every possible legal move. The move selected is the one with the maximum expected outcome for the currently moving side. At each time step, the weights of the network are modified using the $TD(\lambda)$ algorithm. The algorithm determines how much the expected outcome improved (or declined) when compared to the outcome expected at the previous time step, and uses a temporal credit-assignment formula multiplied by this difference in the expected outcome to adjust the weights within the network. A more rigorously mathematical description of the workings of this algorithm can be found in the papers of Sutton [24] and Tesauro [26]. Using this algorithm, from random initial weights, the neural net learned to play backgammon at a strong intermediate level, solely through this self-play mechanism.

Lee and Mahajan [15] used Bayesian learning to optimize the nonlinear evaluation polynomial which was used by their Othello playing program, Bill. Bayesian learning was used to decide whether a particular position represents a win or a loss based on features extracted from the board. A discussion of their use of Bayesian learning to optimize the board evaluation function can be found in Appendix C.

Genetic algorithms have also been used in various ways to generate or improve board evaluation functions. The work in this area is discussed in more detail in Chapter 3.

2.2 Genetic Algorithms

What is now called the *genetic algorithm* was first described by John Holland in 1975 in his book *Adaptation in Natural and Artificial Systems* [10]. The genetic algorithm was inspired by the process of evolution in the natural world, wherein a population of individuals competes for survival, and the individuals which are the most fit survive and propagate themselves.

The genetic algorithm constitutes an artificial analog to this process in the following manner [13]. (Many variations on the scheme described below have also been used.) Individuals are represented as fixed-length chromosonal strings. Each piece of the chromosonal string encodes some attribute of a potential solution to a problem that the creator of the genetic algorithm is attempting to solve. For instance, if one is attempting to optimize a polynomial of several variables by trying various combinations of values of the variables, a chromosonal string would contain a representation of a particular selection of variable values. An initial base population of individuals is constructed from randomly generated chromosonal strings.

To judge the fitness of individuals in the population for solving the problem for which we are using the genetic algorithm, a *fitness operator* is specified. This operator analyzes each chromosonal string in turn and assigns each string, i.e., individual, a fitness value. For example, if the task of the genetic algorithm is to minimize the total weights of all the edges in a weighted graph, the fitness function would sum all the weights of all the edges and assign a fitness value based on that computation. Better fitness values would be assigned to individuals having lower totals, while still satisfying any other constraints which may be present.

In the natural world, a phase of reproduction of individuals within a species followed by a phase of selection that results in another population is referred to as a *generation.* In the context of the genetic algorithm, the term means essentially the same thing. Once the fitness has been determined for every individual in the population, the genetic algorithm creates the population for the next generation. Some of the individuals in that population are survivors from the current generation. The higher the fitness of a particular individual, the more likely the individual is among the survivors.

The bulk of the next population is created by means of *crossover.* The individuals created by crossover are made in pairs. To create each pair of new individuals, two parents are selected in a fitness-proportionate manner. A *crossover point* is selected. The parent strings are then splintered into two pieces at the crossover point. The first piece of the first parent and the second piece of the second parent are joined to form a new chromosonal string. This is the first child. The two remaining pieces are joined to form the second child. This version of the crossover process is known as *one-point crossover* [13]. Other crossover operators also exist.

Once the new population has been created, a relatively small number of individuals are *mutated.* A mutation consists of randomly changing one of the elements of the chromosonal string representing an individual.

The evolutionary-like process, inspired by the concept of survival of the fittest, repeats for many generations until a stopping criterion is achieved. Example stopping criteria include: the passage of a preset number of generations, the “convergence” of the population to a particular solution, or the exhaustion of an execution-time limit.

A pseudocode description of the genetic algorithm is given in Figure 2.1. This description is only one version of the genetic algorithm, and an enormous number of variations on this basic scheme can be found in the literature.

2.3 Genetic Programming

The concept of genetic programming was first formulated by John R. Koza in 1989 [12] and further developed in his two books on the subject [13] [14]. The basic outline of the genetic programming paradigm bears a strong resemblance to that given above for the “standard” genetic algorithm. The key difference lies in how individuals are encoded. Rather than encoding individuals as chromosonal fixed-length strings, they are instead encoded as computer programs. The programs are typically expressed as LISP functions. These functions are constructed from a predetermined (and application specific) set of function templates (referred to as *non-terminals*) and constants (referred to as *terminals*). As a LISP program, each individual is a syntax tree with the root of the tree being a non-terminal from the non-terminal set and the children being arguments to this non-terminal. Each child can either be a terminal (making it a leaf of the tree) or another non-terminal (making it a subtree). An example of such an individual can be seen in Figure 2.2.

Given this representation, it is fairly straightforward to construct analogs in the context of genetic programming for the operations found in standard genetic algorithms. Aside from the differences in the details of representation, the outline for the operation of genetic programming is identical to that of the “standard” genetic algorithm. The crossover operator works as follows. A node is selected in a uniformly random manner from the tree of each

```
population = create-initial-population(population-size)

while (stopping criteria is not met)

  compute-fitness(population)

  for (i = 1; i <= number-of-crossovers; i += 2)
    parent1 = select-a-parent-in-proportion-to-fitness(population)
    parent2 = select-a-parent-in-proportion-to-fitness(population)
    crossoverpoint = randomly-chosen-index-into-chromosonal-string()
    new-children[i] = crossover(parent1, parent2, crossoverpoint)
    new-children[i+1] = crossover(parent2, parent1, crossoverpoint)

  for (i = number-of-crossovers + 1; i <= population-size; i++)
    new-children[i] = select-survivor-in-proportion-to-fitness()

  for (i = 1; i <= number-of-mutations; i++)
    mutate(new-children[randomly-selected-index()])

  population = new-children
```

Figure 2.1: Pseudocode for a Genetic Algorithm

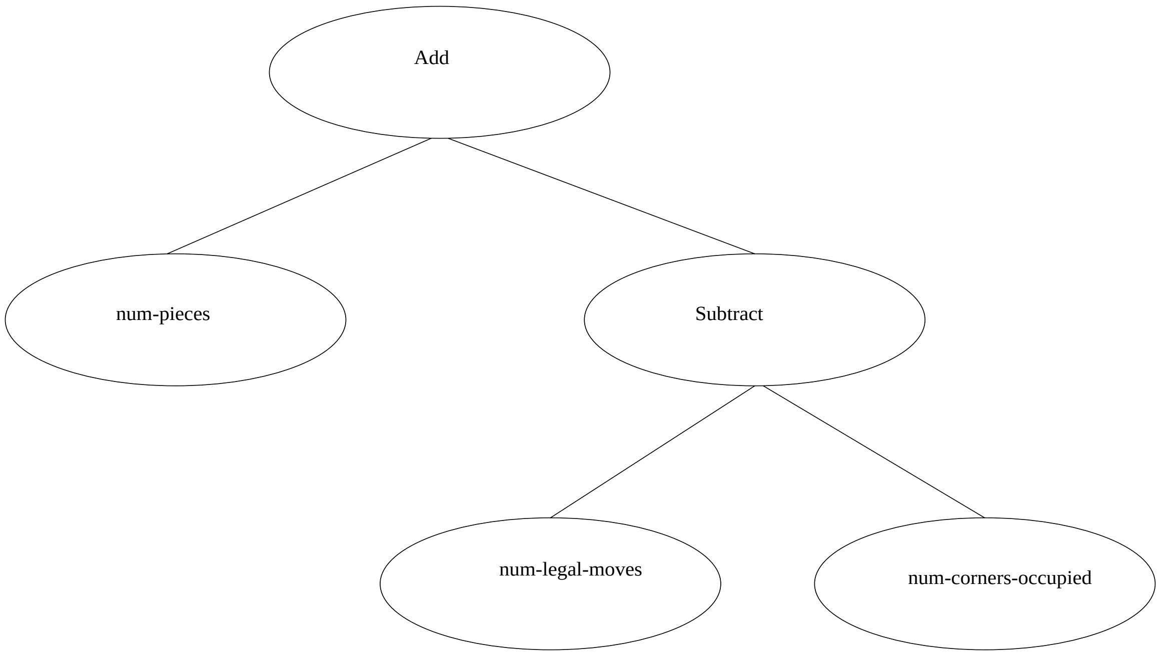

Figure 2.2: A Sample Tree

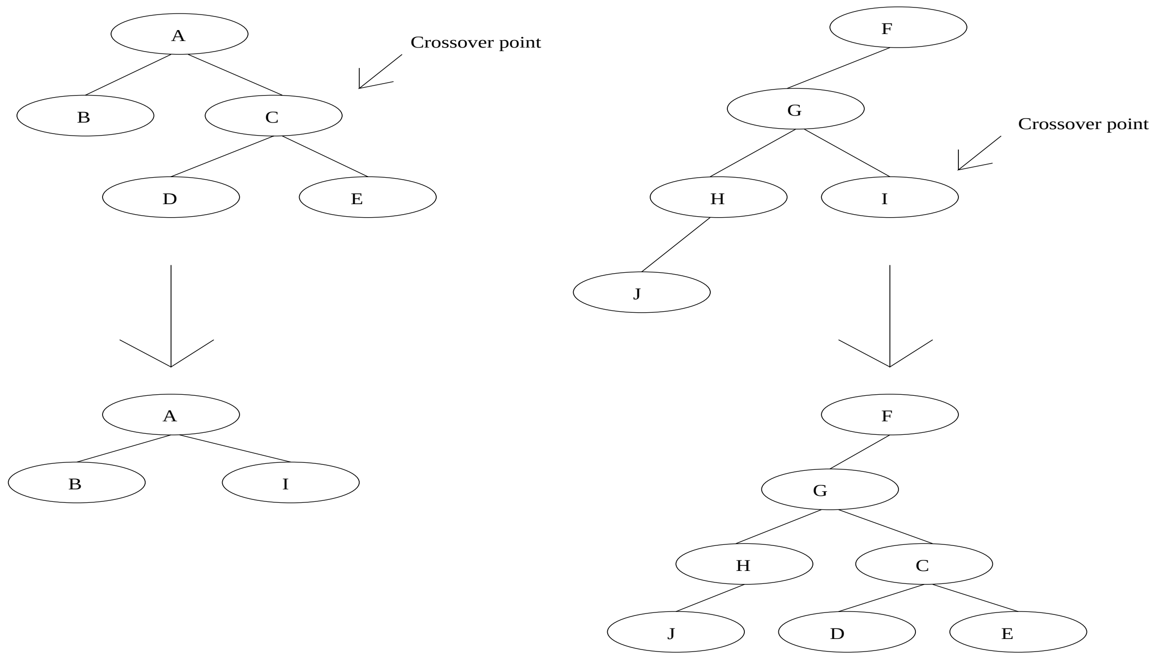

Figure 2.3: Example of Crossover

parent as the crossover point. The subtrees rooted at the crossover points of each parent are exchanged, generating two children. An example of crossover can be seen in Figure 2.3. The mutation operator replaces a randomly selected node or leaf with another randomly generated leaf or subtree.

The fitness operator executes the genetic program in the applicable environment with appropriate arguments (if any) and evaluates its performance to determine the fitness value. The precise mechanism for accomplishing this will depend on the task at hand. For example, if an individual happens to be a function which is being optimized, executing the fitness operator may simply consist of evaluating the function with certain parameters. If an individual is a control program for a robot, executing the fitness operator may involve judging the robot's performance on some task in a simulated environment. If an individual is a board evaluation function, executing the fitness operator may involve playing a complete game to determine the function's performance.

In general, when a problem lends itself well to representation as a computer program, applying genetic programming is a viable option. It has even been shown that inclusion of primitives to read and write memory in some fashion can make genetic programming Turing Complete, enabling the generation of any conventional computer program [25]. What programs can actually be evolved in practice remains an open problem. If the sets of non-terminals and terminals contain any kind of indefinite looping operator, it is possible (and, it would appear, not entirely unlikely) that individuals in the population might specify individuals that when executed fall into infinite loops. The conventional solution to this problem (assuming that indefinite looping constructs are incorporated at all) is to have a time limit on the execution of an individual to insure that fitness evaluation stops as it should.

A problem encountered with encoding individuals as trees and using the crossover operation as described above is that the individuals are of variable sizes. It is possible to create individuals so large that they significantly slow down the system as a whole, partly because they take a long time to evaluate and partly because dynamic memory allocation mechanisms slow down when large, dynamic data structures are involved. In spite of these and many other practical problems, we have implemented a working genetic programming system to explore the applications of GP to board evaluation function evolution.

3

Evolving Board Evaluation Functions

This chapter describes the form of the genetic programming paradigm that we used to evolve board evaluation functions in our experiments. Separate sections in this chapter describe the particulars of the application of our GP formulation to Senet and to Othello. An overview of other schemes to generate board evaluation functions based upon evolutionary computations can be found in Appendix D.

3.1 Problem Formulation

The genetic programming paradigm requires the formulation of potential solutions as evaluable programs. We have chosen to represent each individual player as a static board evaluation function.

The system can simulate the play of an individual by constructing the set of possible moves, applying the represented evaluation function to the board configurations corresponding to each move in the set, and selecting the move with the highest resulting evaluation. Thus, the individuals of our population are functions that take a board configuration and return an evaluation number. We happen to stipulate that larger numbers indicate board configurations, and thereby, moves, that the individual finds more preferable. In the event that the evaluation function returns the same largest number for multiple moves, the default behavior is to select one of the moves randomly by using a uniform distribution to select one from the set of largest-evaluation-number moves.

Note that with this formulation, any function that returns a number whenever executed thus becomes a feasible player. Our system will maintain this property for all individuals created. Maintaining this property avoids the potential complication of introducing individuals into the population who do not have a well-defined fitness, i.e., who do not play the game legally. For example, infeasible tic-tac-toe players can be created by Angeline and Pollack's system [1]. In their representation, if a player never invokes the function which causes a move to occur, the player will never move and the opponent will get bonus moves. Such individuals are not, strictly speaking, playing legal tic-tac-toe and can be considered infeasible as a result.

The *fitness* operator determines fitness using a competition scheme modeled on a sports tournament, similar to that used for the game of tic-tac-toe by Angeline and Pollack [1]. The individuals in the population are paired off arbitrarily and then play some number of games against each other. The individual who wins the most games is declared the winner and progresses to the next round of the tournament. The losers at each level all have the same fitness.

The initial population of board evaluation functions is generated as follows. For each individual, a root non-terminal is selected which is a non-terminal requiring at least 2 arguments. Another non-terminal or terminal is randomly selected for each argument (again, "randomly" means that a uniform distribution was used). If the selected item is a non-terminal, its arguments are determined in the same manner. This growing process terminates when terminals have been selected for all remaining argument slots.

The *crossover* operator works as follows. A node is selected in a uniformly random manner from the tree of each parent as the crossover point. The subtrees rooted at the crossover points of each parent are exchanged, generating two children. This is the standard genetic programming crossover operator as described by Koza [13].

The *reproduction* operator copies selected individuals from the old population into the new population. For each individual reproduced, there is a one in eight chance that that individual will mutate. A mutation consists of replacing a randomly selected subtree (with uniform distribution) of the individual with a non-terminal generated randomly in the same manner that the individuals in the initial population are generated.

A population of individuals is evolved in the following manner. Selection for both reproduction and crossover is done in a rank-proportionate manner [2]. One-eighth of the new population is reproduced from the original population at each new generation, with the remainder of the new population being generated via crossover. These proportions are based on those used by Koza in his first book [13]. In that work, his experiments generated 10% of each new population by reproduction and the remaining 90% by crossover. We used one-eighth because our population sizes were powers of two in order to facilitate the use of the fitness function which will be discussed in detail later in this chapter. This evolutionary process ends after a preset number of generations have been completed.

A pseudocode sketch of the above formulation is given in Figure 3.1.

Our implementation also had to address issues concerning the composition of individuals, namely, the allowable size of individuals, and the pool of terminals and non-terminals available. If the size of individuals is permitted to grow without bound, the system may run out of memory or run excessively slow. In order to prevent the size of individuals from getting too large, height limits are placed upon both newly created individuals and individuals generated by the crossover process. The limits used are a maximum tree height of six for a newly created individual, and a maximum tree height of seventeen for an individual which has been created through crossover or mutation. These same limits have been used by Koza [13]. We experimented briefly with removing height limits from the process to determine whether the presence of height limits negatively impacted the ability of individuals to evolve. No significant difference was observed between the fitness of individuals evolved with and without the standard height limits.

```
number-of-crossovers = (7/8 * population-size)
population = create-initial-population(population-size)

while (last generation not yet reached)

  run-tournament-and-rank(population)

  for (i = 1; i <= number-of-crossovers; i += 2)
    parent1 = select-a-parent-in-proportion-to-fitness(population)
    parent2 = select-a-parent-in-proportion-to-fitness(population)
    crossover-point-1 = randomly-chosen-node(string-length, parent1)
    crossover-point-2 = randomly-chosen-node(string-length, parent2)
    new-children[i] = crossover(parent1, crossover-point-1,
                                parent2, crossover-point-2)
    new-children[i+1] = crossover(parent2, crossover-point-2,
                                  parent1, crossover-point-1)

  for (i = number-of-crossovers + 1; i <= population-size; i++)
    new-children[i] = select-survivor-in-proportion-to-fitness(population)
    if (random(8) == 1) mutate(new-children[i])

  population = new-children
```

Figure 3.1: Pseudocode for the GP Formulation

Naturally, the collection of non-terminals and terminals has a profound influence on what evaluation functions might potentially evolve. It is important to make sure that this collection contains enough useful items so that good board evaluation functions have a good possibility of evolving. To this end, we incorporated non-terminals and terminals that automatically compute the values of certain strategic features. For example, in the Othello implementation, there is a non-terminal that computes how many corner squares are occupied by a specified player. At the same time, it is important to be sure that the collection does not predispose the evolutionary process towards a particular strategic school of thought based upon the non-terminals and terminals selected. Terminals and non-terminals that are singularly complex can create immediate strong local maxima. This can preclude the construction, through evolution, of complex combinations of other simpler terminals and non-terminals. Balancing these concerns is a key issue in deciding upon a worthwhile GP formulation. The next two sections address this and other game-specific concerns of this formulation.

3.2 Application to Senet

In the game of Senet each turn begins with a randomized construction (by the use of throw sticks) of a set of moves. The player then gets to select the sequence in which the moves in the constructed set are executed. (A summary of the rules can be found in Appendix A, while the detailed rules can be found in [11].) The board evaluation function then selects the most desirable move of those available from the present board position. Because of the probabilistic nature of the game, a scheme to search beyond one ply is too computationally expensive to be practical, as is also the case in backgammon [26]. Consequently, we limit our application of the board evaluation function to the moves which can be made immediately.

Each match in the Senet tournament is a best-of-three games match, with the winner of two or three games being declared the overall winner. We decided to use a three game match to reduce the chance of a superior individual losing to an inferior individual because of bad luck in the first game. We did not use more than three games because one of our major performance bottlenecks is that it takes a long time to execute a game between two individuals, and so we did not want each pair to play too many. Since it is theoretically possible for a Senet game to last indefinitely, a limit of 500 turns was imposed on the duration of a game. Our experiments have shown that Senet games rarely take even half that many turns to complete, so we believe that 500 turns is a very reasonable limitation. In a sample of 38,100 games, the average number of turns played per game was approximately 158. In the event that a game lasts as long as 500 turns, the victor is determined arbitrarily.

The non-terminals and terminals which were incorporated into the set available for the construction of Senet evaluation functions fall into the following categories:

- **Arithmetic Operators**: The addition, subtraction, and multiplication operators enable the board evaluation function to manipulate integers in order to express desired preference numbers for the board positions being evaluated.

- **Logical and Decision Operators**: The `if` operator enables the board evaluation function to make decisions regarding what argument to execute, based on the result of a boolean computation. The `and`, `or`, and `not` operators enable the board evaluation function to manipulate the results of boolean terminals and operators.

- **Integers**: The integers from 1 to 30 are included. These enable the function to express preferences and also to specify board locations for the board querying non-terminals.

- **Board Querying Operators**: Information regarding the current state of the board can be obtained using the board querying operators. Examples of these include operators to determine the contents of a board location or to find the number of moves available to a player.

A complete description of the set of terminals and nonterminals can be found in Appendix E.

3.3 Application to Othello

This section discusses the issues involved in applying genetic programming to evolving board evaluation functions for Othello. The game of Othello has been the target of a fair amount of work aimed at automatically generating or adjusting board evaluation functions. An overview of using computers to play Othello can be found in Appendix C. Some examples of using other evolutionary computation techniques to evolve Othello board evaluation functions are discussed in Appendix D.

In the game of Othello, on each turn a player gets to place exactly one piece on the board if the player has a legal move available. The board evaluation function is used to select the most desirable move by evaluating each resulting board position. The board evaluation function can be used in conjunction with an alpha-beta search mechanism to evaluate board positions at a deeper level. This is not possible for Senet because the element of probability at each move makes it impossible to accurately predict which moves will be available more than one level ahead of the current position. The deterministic nature of Othello, by contrast, makes the use of a search mechanism straightforward.

For fitness evaluation, each tournament match between two individuals consisted of four games, two with each player going first. The player with the highest sum of scores over the four games was declared the winner. We wanted the individuals to play an even number of games against each other, so that each player would be each color twice. This is an attempt to cancel out potential advantages involved with being assigned a particular color. Note that although the game of Othello is deterministic in nature, we still chose to play two games with each player starting. We did this because of the potentially nondeterministic results of the default behavior mentioned in section 3.1. Note that the players themselves are deterministic and will always return the same value for any given board configuration. Finally, in the (rare) case where both players had the same total score for all four games, the winner was picked arbitrarily.

The non-terminals and terminals incorporated into the set available for the construction of Othello evaluation functions fall into the following categories:

- **Decision, Arithmetic, and Logical Operators**: These are the same as those used in the Senet formulation.
- **Integers**: The integers from 0 to 63 are included. These enable the function to express preferences and also to specify board locations for the board querying non-terminals.
- **Board Querying Operators**: As with the Senet formulation, these return information about the current state of the board. Examples include queries about the contents of a board location and how many pieces a particular player has.

A complete description of the set of terminals and nonterminals can be found in Appendix E.

4

Experiments and Results

This chapter discusses the experiments we ran and the results we obtained in evolving board evaluation functions. These results show that the problem formulation described in Chapter 3 was reasonably effective for evolving players for both Senet and Othello. Our scheme to empirically evaluate the results of the evolutionary process is described. The incorporation of handcrafted individuals into the starting population in an attempt to improve the final results of the evolutionary process is also discussed.

4.1 Experimental Design

Due to the combinatorial explosion involved in computing potential future board states in any non-trivial boardgame, it is difficult at best to know what an optimal board evaluation function should be. We also lack a distance measure on the space of possible functions. These facts require us to create an empirical evaluation scheme. The fitness of the players for our evolutionary purposes is determined by how good they are at defeating each other. But how do we know that the overall winner, that is, the tournament winner in the last generation, has any objective value whatsoever? Our empirical evaluation scheme uses a series of baseline players to evaluate the quality of our results. Each baseline player is implemented from the same set of terminals and non-terminals available to the individuals being evolved through genetic programming, and they are subject to the same default behavior.

For the game of Senet, the following players were devised. One group of baseline players is a set of eight individuals (referred to as RND1 through RND8 below) generated randomly by the code that creates initial populations. The other group consists of five handcrafted players (referred to as HC1 through HC5 below) inspired by strategies we have used in playing the game. The players in this group use a variety of combinations of some of the following strategies:

1. Minimize the number of my pieces on the board.
2. Maximize the number of my opponent's pieces on the board.
3. Avoid the Waterhouse pitfall.
4. Place my opponent in the Waterhouse pitfall.
5. Keep the spread of my pieces close together.
6. Maximize the distance between my opponent's pieces and the goal.
7. Capture an opponent's piece.
8. Set up a barricade.

Player HC1 incorporates the first six strategies mentioned above. HC2 incorporates all eight. HC3, HC4, and HC5 are, by comparison, much more simplified. HC3 concentrates on item 7, HC4 concentrates on item 8, and HC5 combines the two, giving preference to obtaining captures.

One additional player we included is an individual that always returns a constant for each board configuration, which in conjunction with the system's default behavior has the effect of selecting a random move at each turn, with a uniform distribution over the set of moves constructed for the current turn. (This player will be referred to as Const below).

We used a similar evaluation scheme for Othello. A detailed overview of Othello strategy as it has been used in constructing Othello board evaluation functions can be found in Appendix C. For the purposes of performance evaluation here, we implemented some very simple and straightforward Othello strategies. Each of the following strategies is represented in this group:

- OHC1 tries to maximize the difference in material between the two players at all times.
- OHC2 tries to maximize the number of moves it has available.
- OHC3 maximizes its moves and tries to minimize its opponent's moves at the same time.
- OHC4 computes the product of its number of moves available and the number of corners it occupies and subtracts from that the corresponding product for its opponent.
- OHC5 combines OHC1 and OHC4 by running OHC4 until move 56, and then running OHC1 for the rest of the game.

As with Senet, we also included an individual that effectively makes a random move on every turn.

For the results presented below, we compared pairs of players by having them play a set of 200 games against each other, so as to give us statistical confidence in the results of the test. This works for the Senet players because the game of Senet is not deterministic. This works for the Othello players because the default behavior in the event of an evaluation which results in multiple board configurations being assigned the same best evaluation number, the move is selected randomly. Note that this testing phase is different than the tournament structure used for fitness calculation and is done after the fact as an external quality measure. Some of our experiments utilized initial populations which consisted entirely of randomly generated individuals. The rest of our experiments contained a small number of seed individuals in the initial population.

We wanted to see how population seeding would work in the context of genetic programming, because it has been used successfully before with more standard genetic algorithms [9]. Our expectation was that incorporating some of our knowledge about the games into the initial population would improve the results of the search process. Incorporating hand-crafted players which encode useful strategies for game play can provide promising directions for where the search might proceed. They interact with the rest of the population in the tournament format, and when successful, get selected for reproduction and crossover. When crossover and mutation operate on these individuals, there is the possibility that an improvement on the seed individual will emerge. We decided to seed the population with only a small number of such individuals, so as to encourage diversity in the population.

Our experiments took a long time to complete. Evolving populations for 400 generations took around 6 weeks. Consequently, we did not have enough time available to experiment with a large number of different values for the parameters of evolution. For example, we would like to have experimented with different values for the mutation rate, the proportion of the population created by crossover, the number of games used in each match in the tournament, the number of seed individuals in the starting populations where they were used, and a greater range of population sizes.

4.2 Results

4.2.1 Explanation of the Graphs

Each of the graphs in this section describes the performance of a given set of evolved players against a given set of test players, e.g. RND7 and HC3. The evolved players are referred to in the graphs by numbers, e.g. One, Two, or Three. Each point on the x-axis indicates a test player against which an evolved player competed. The y-axis of each graph represents the total number of victories out of 200 won by each evolved player. In the case of Othello, ties are counted as one-half of a victory. Thus, values greater than 100 indicate increasingly better performance for the evolved players, while values lower than 100 indicate progressively worse performance. Each of the evolved players is indicated by a point designating its score against a particular test player.

The above description suffices for most of the graphs. A few of the graphs (Figures 4.9, 4.10, 4.20, and 4.21) chart the performance of the best individuals in a population as that population evolved. So for those graphs, the x-axis indicates from which generation the "current best individual" at that position has been extracted. The y-axis represents the number of victories out of 200 won by the "current best individual". Each line charts how the "current best individual" performed against a particular test player as the population evolved.

A few other graphs (Figures 4.11, 4.22, 4.25, and 4.28) are plots of the average scores against the baseline players of one set of winning individuals vs. the average scores of another set of individuals. For these graphs, the performance of the randomly generated players is averaged together into a single composite baseline player. "256" is the designation for the average score of all the winning individuals which emerged from populations of size 256. "512" similarly designates the average score of all the winning individuals from populations of size 512.

4.2.2 Purely Random Initial Populations

4.2.2.1 Senet

We evolved a total of 16 populations of Senet players with purely random initial populations. Nine of them had populations of size 256 and the other seven had populations of 512. Out of the populations of size 256, four were evolved for 100 generations, four for 200 generations, and one for 400 generations. Out of the populations of size 512, four were evolved for 100 generations, two for 200 generations, and one for 400 generations.

Against the randomly generated individuals and Const, the best individuals from each population, i.e. the winners of the last tournament in the last generation, all performed extremely well, typically winning 75% or more of games played against those individuals, as can be seen in Figures 4.1, 4.3, 4.5, 4.7, 4.9, and 4.10. Against the handcrafted baseline players, performance was still quite good, with none of the evolved individuals winning less than half the games against any of the baseline players, and more often than not winning about 60% to 70% of the time. These results can be observed in Figures 4.2, 4.4, 4.6, and 4.8.

One aspect of the dynamics of the evolutionary process is shown in Figures 4.9 and 4.10. In both graphs, the best individual in the population, i.e., the individual which won the tournament during the fitness evaluation of that population, is taken every five generations and tested against the test individuals. Looking at the figures, we can observe that the best individual in the larger population tends to be consistently slightly better than the best individual in the smaller population until soon after the 200th generation. From that point on, the best individuals in each population tend to perform similarly, with occasional anomalous individuals demonstrating poor performance. The non-deterministic nature of genetic programming is what permits such anomalies to occur. The competitive nature of the fitness function encourages the resurgence of superior individuals and helps to minimize the overall impact of these anomalies.

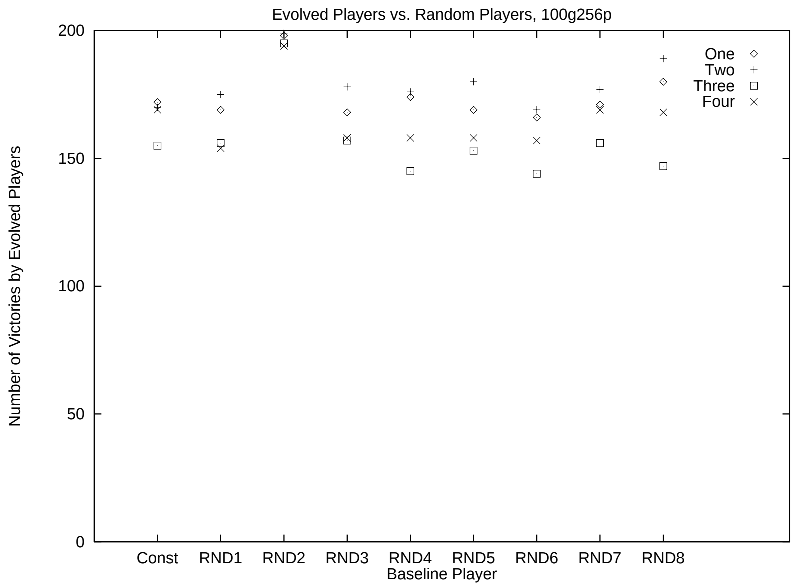

Figure 4.1: Unseeded Evolved Senet Players vs. Random Players, Population 256, 100 Generations

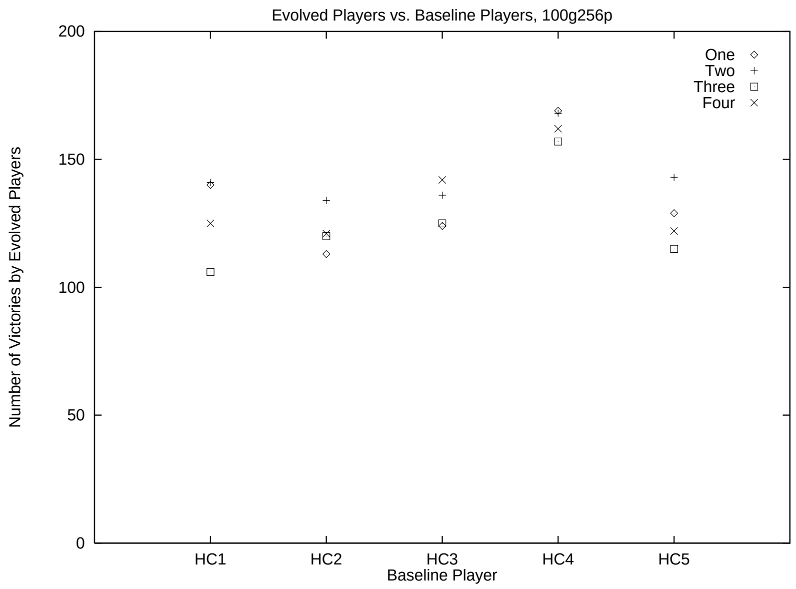

Figure 4.2: Unseeded Evolved Senet Players vs. Handcrafted Heuristics, Population 256, 100 Generations

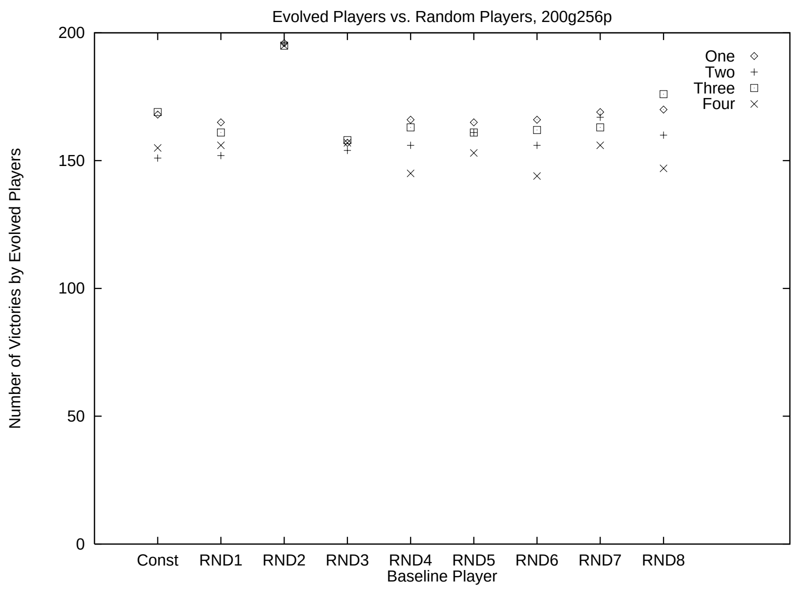

Figure 4.3: Unseeded Evolved Senet Players vs. Random Players, Population 256, 200 Generations

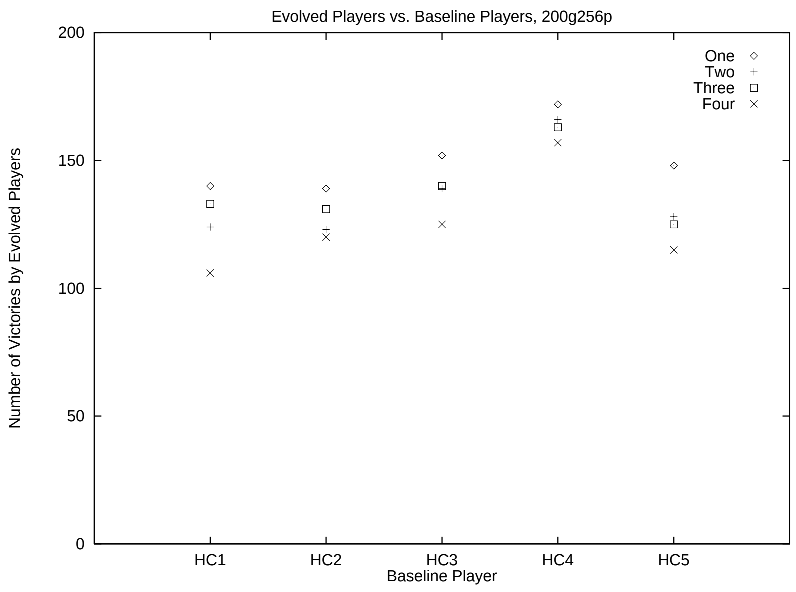

Figure 4.4: Unseeded Evolved Senet Players vs. Handcrafted Heuristics, Population 256, 200 Generations

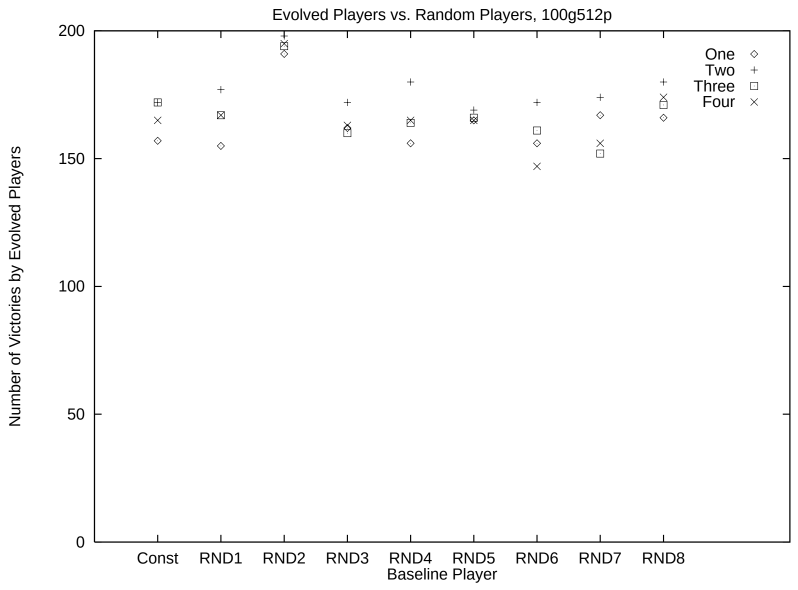

Figure 4.5: Unseeded Evolved Senet Players vs. Random Players, Population 512, 100 Generations

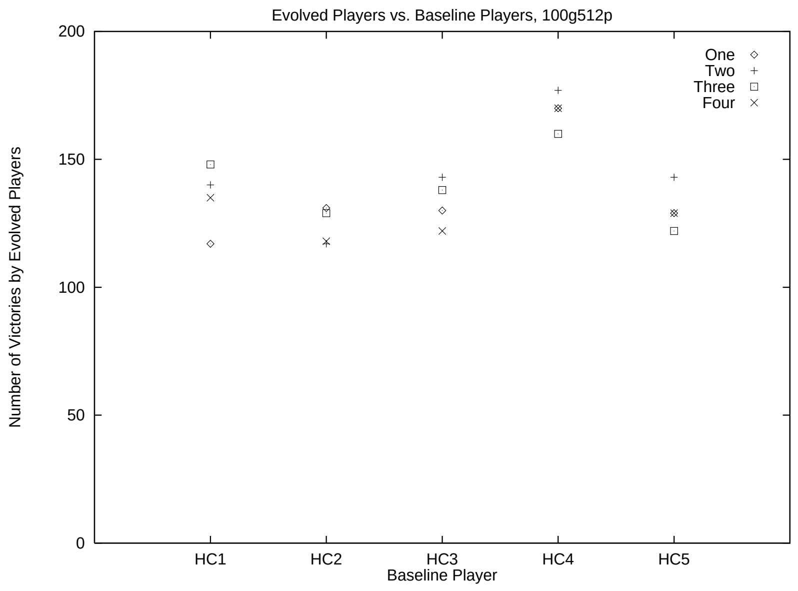

Figure 4.6: Unseeded Evolved Senet Players vs. Handcrafted Heuristics, Population 512, 100 Generations

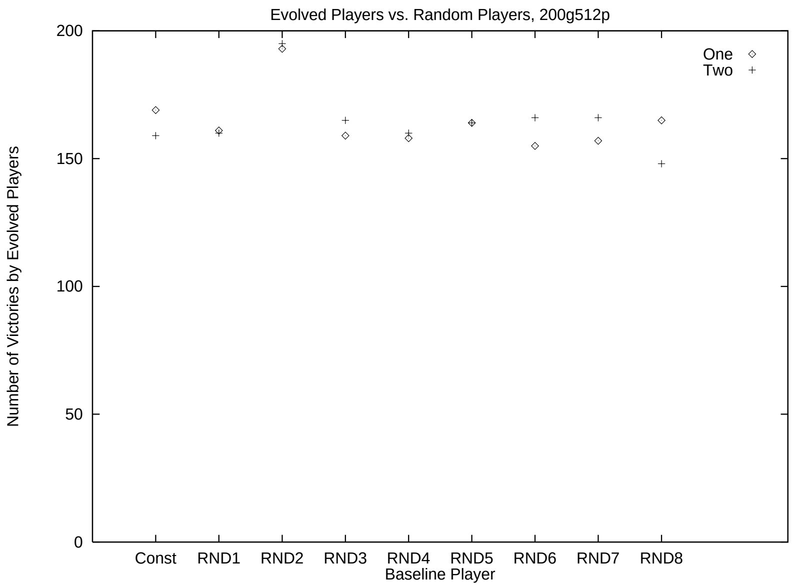

Figure 4.7: Unseeded Evolved Senet Players vs. Random Players, Population 512, 200 Generations

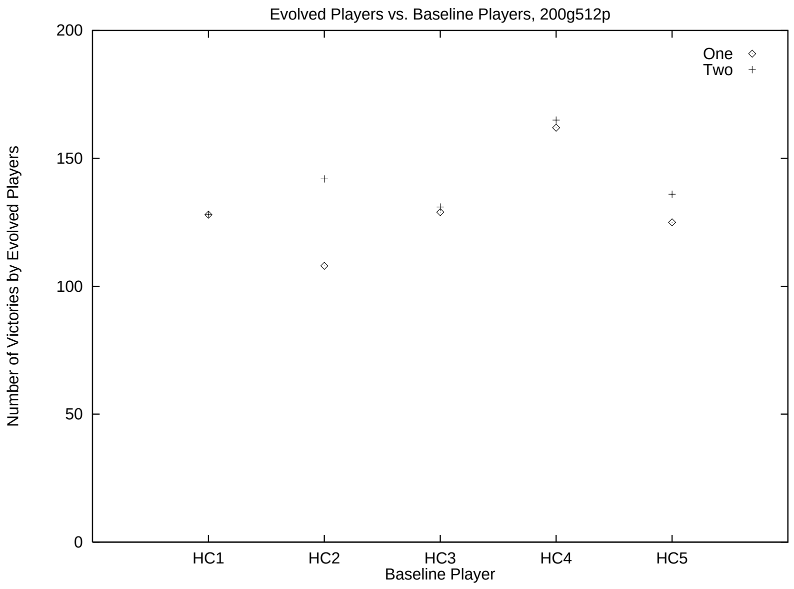

Figure 4.8: Unseeded Evolved Senet Players vs. Handcrafted Heuristics, Population 512, 200 Generations

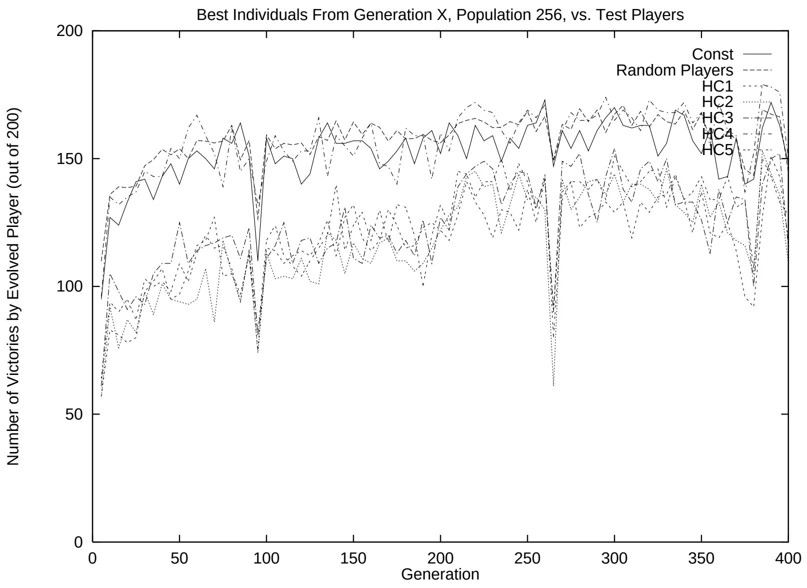

Figure 4.9: Evolved Senet Players Every 5 Generations vs. Heuristics, Population 256, 400 Generations

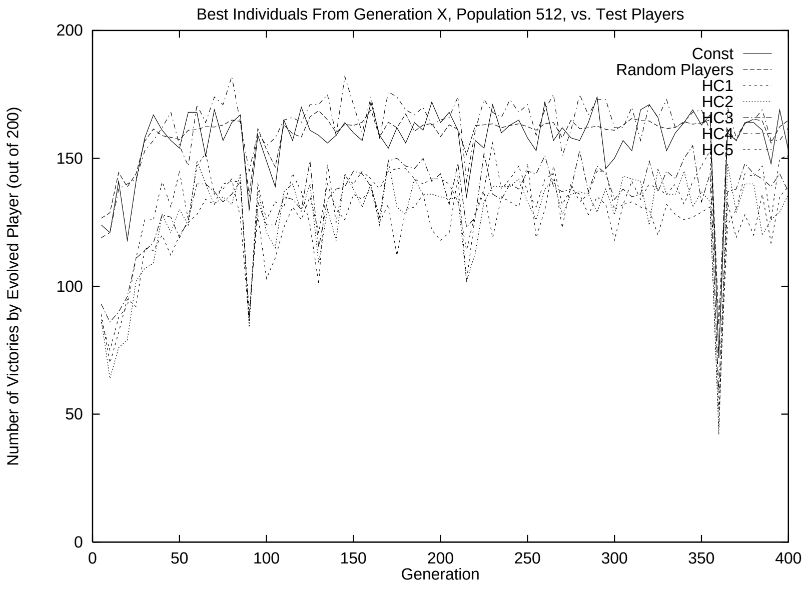

Figure 4.10: Evolved Senet Players Every 5 Generations vs. Heuristics, Population 512, 400 Generations

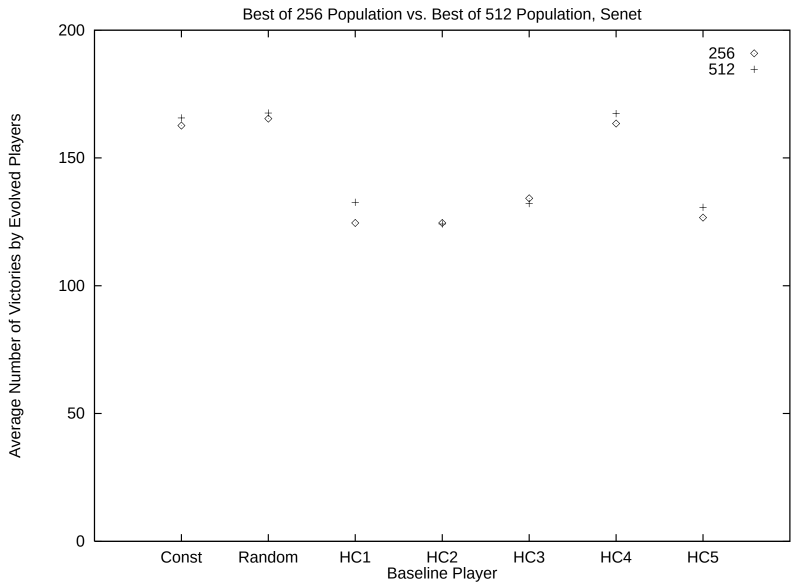

Figure 4.11: Comparison of Winning Senet Individuals Evolved for 256 Generations vs. 512 Generations

4.2.2.2 Othello

We evolved a total of 14 populations of Othello players with purely random initial populations. Seven of them had populations of size 256 and the other seven had populations of size 512. For each of those population sizes, four individuals were evolved for 100 generations, two for 200 generations, and one for 400 generations.

Against the randomly generated individuals and Const, they all performed reasonably well, typically winning about 65% or so of games played against those individuals, as can be seen in Figures 4.12, 4.14, 4.16, 4.18, 4.20, and 4.21. Against the handcrafted baseline players, performance was not particularly good, with none of the evolved individuals typically winning more than half the games against any of the baseline players. These results can be observed in Figures 4.13, 4.15, 4.17, 4.19, 4.20, and 4.21.

Figures 4.20, and 4.21 show how the best individuals in those two populations changed over the course of 400 generations. In both cases, the rate of improvement slows down relatively early in the run. In Figure 4.21, we can observe erratic but somewhat consistent improvement until between generations 50 and 100, where improvement, for the most part, levels off and performance remains fairly steady for the remainder of the run, with the exception of a couple of "spikes" of bad performance. Figure 4.20 shows similarly erratic yet somewhat consistent improvement until around generation 50. In both graphs, the improvement is most visible against Const and the random players. An apparent anomaly which can be observed in Figure 4.20 is that the best individual at generation 5 shows much better performance across the board than just about all the best individuals from subsequent generations. It seems reasonable to conclude that this formulation does not do a particularly good job of preserving the best individuals in a population. On the other hand, the short lifespan of the individuals represented by the spikes in Figures 4.20 and 4.21 demonstrates that the fitness function is good enough to insure that such individuals do not persist long enough to dominate the population.

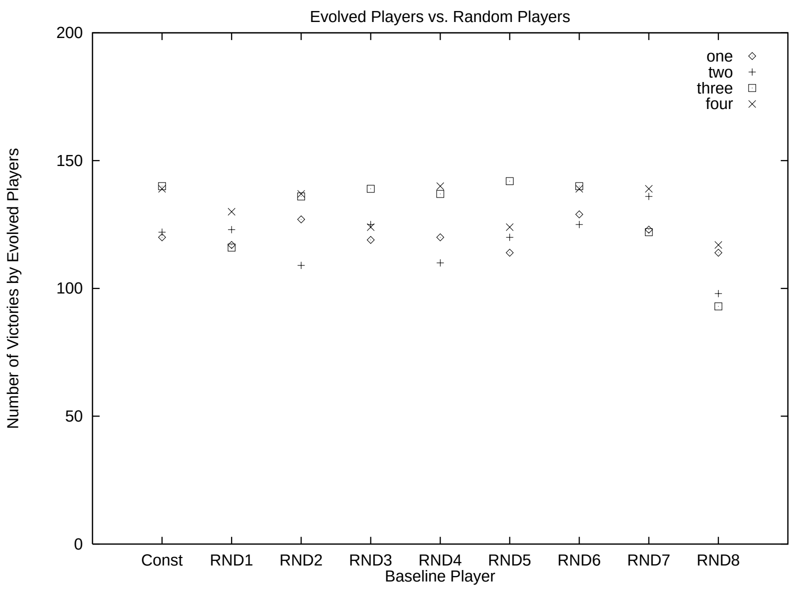

Figure 4.12: Unseeded Evolved Othello Players vs. Random Players, Population 256, 100 Generations

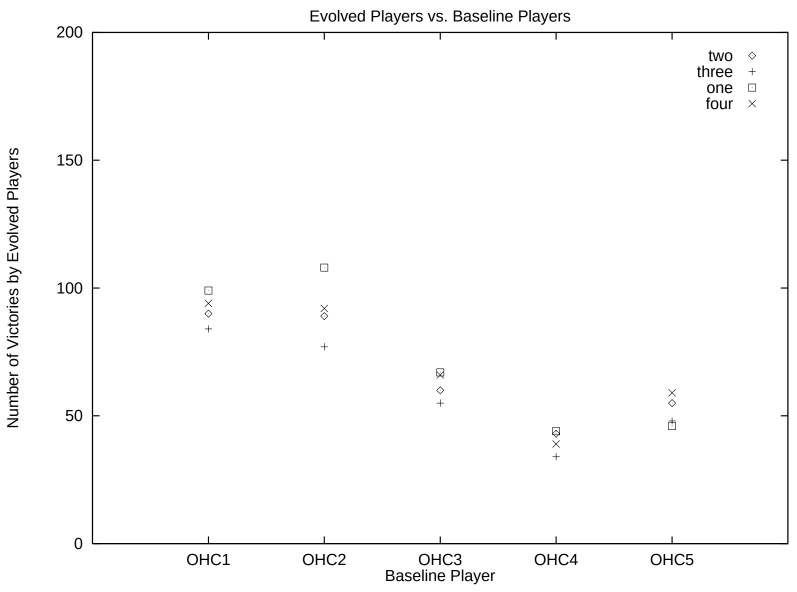

Figure 4.13: Unseeded Evolved Othello Players vs. Handcrafted Heuristics, Population 256, 100 Generations

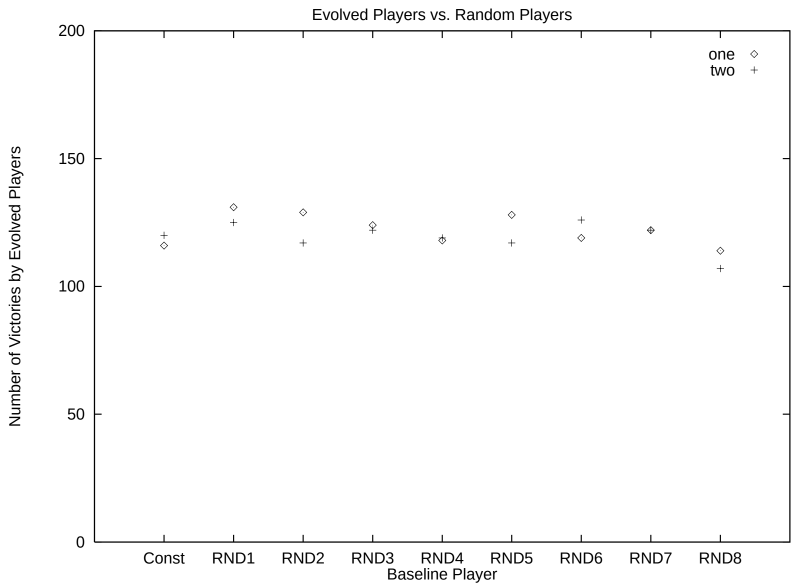

Figure 4.14: Unseeded Evolved Othello Players vs. Random Players, Population 256, 200 Generations

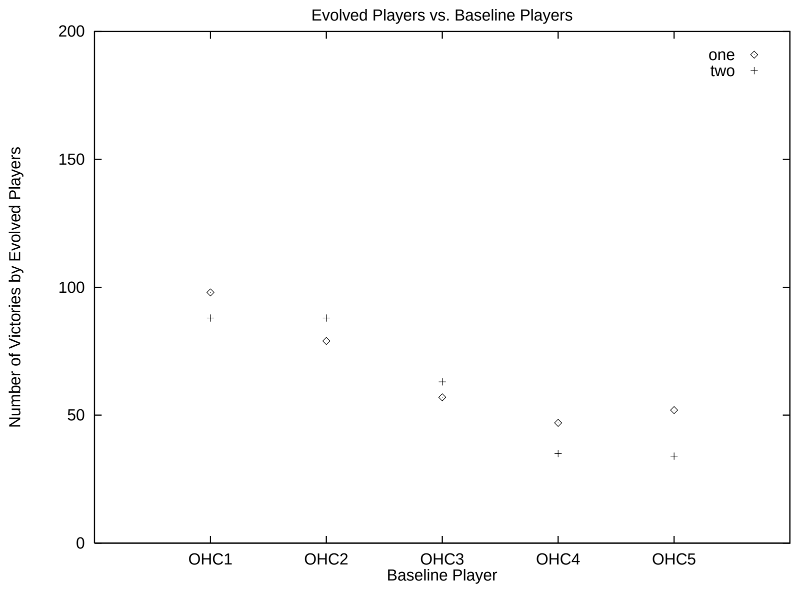

Figure 4.15: Unseeded Evolved Othello Players vs. Handcrafted Heuristics, Population 256, 200 Generations

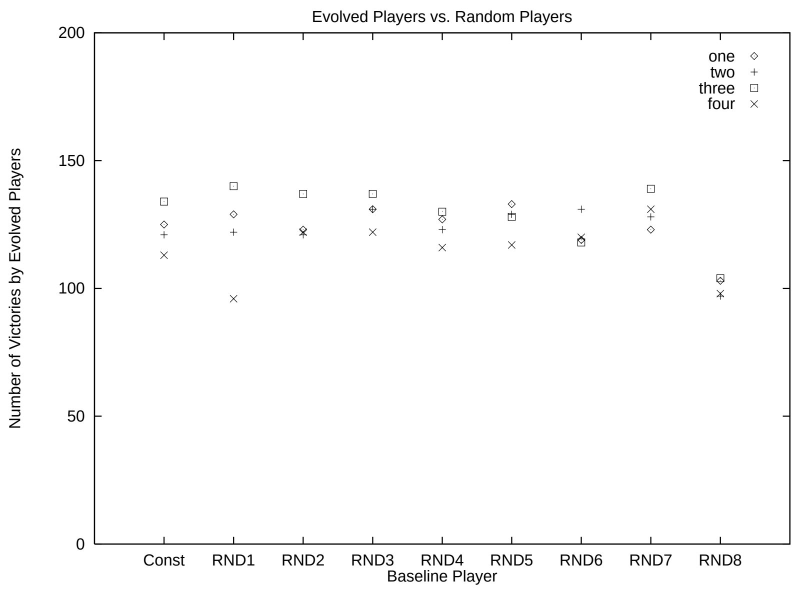

Figure 4.16: Unseeded Evolved Othello Players vs. Random Players, Population 512, 100 Generations

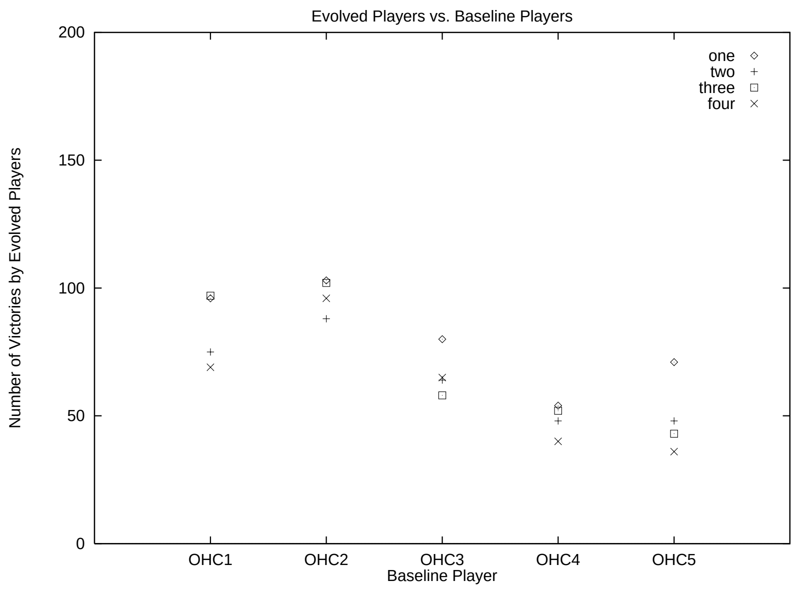

Figure 4.17: Unseeded Evolved Othello Players vs. Handcrafted Heuristics, Population 512, 100 Generations

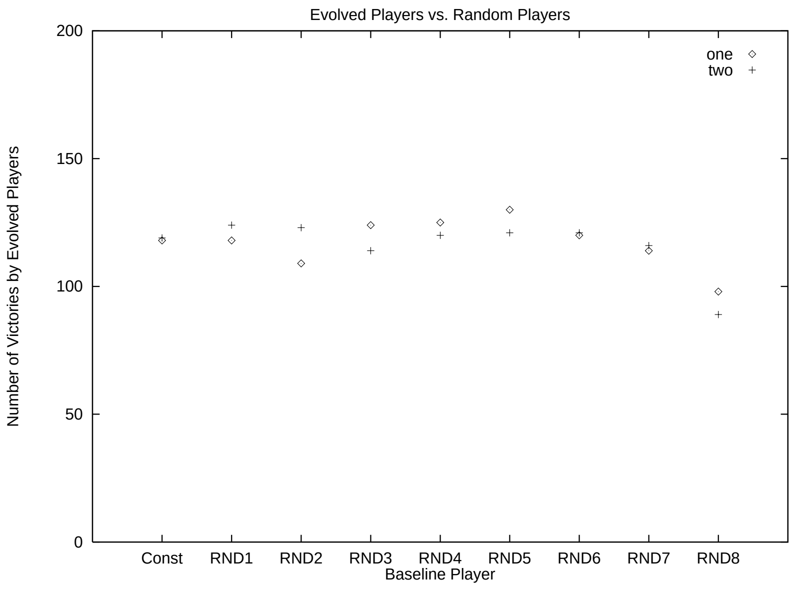

Figure 4.18: Unseeded Evolved Othello Players vs. Random Players, Population 512, 200 Generations

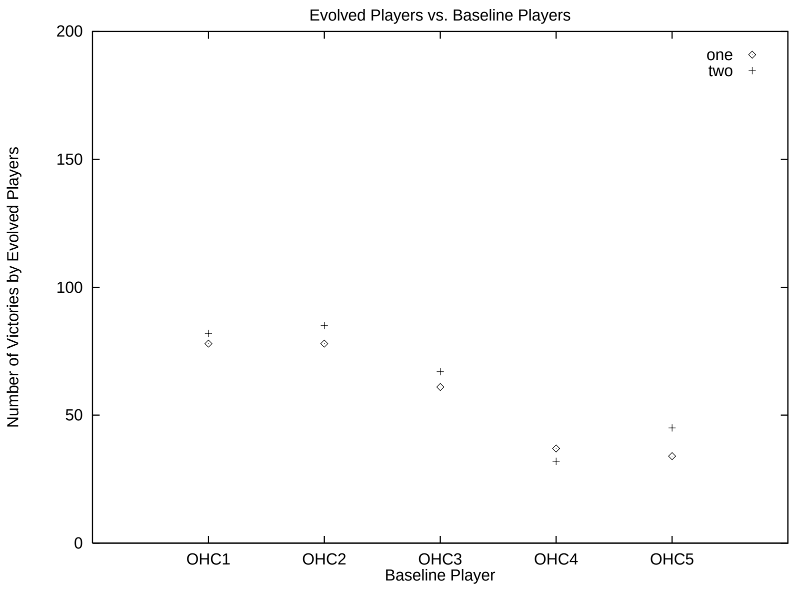

Figure 4.19: Unseeded Evolved Othello Players vs. Handcrafted Heuristics, Population 512, 200 Generations

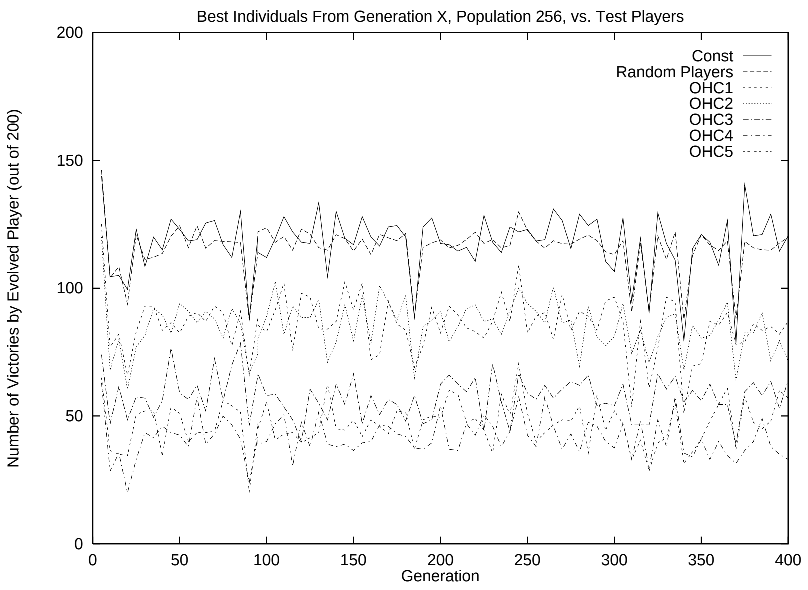

Figure 4.20: Evolved Othello Players Every 5 Generations vs. Heuristics, Population 256, 400 Generations

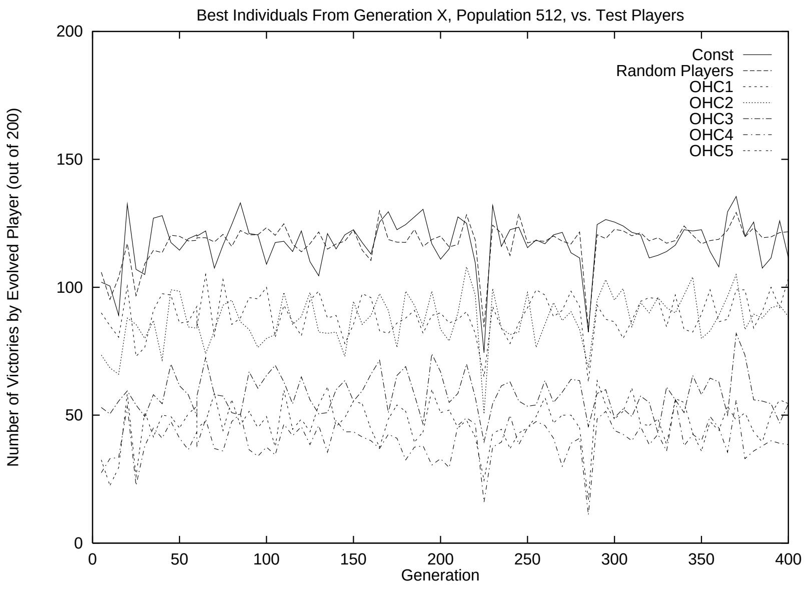

Figure 4.21: Evolved Othello Players Every 5 Generations vs. Heuristics, Population 512, 400 Generations

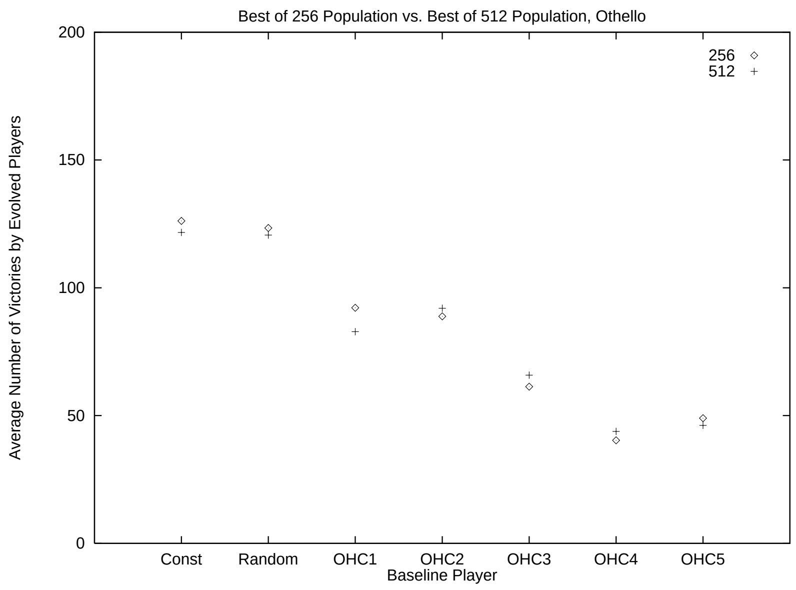

Figure 4.22: Comparison of Winning Othello Individuals Evolved for 256 Generations vs. 512 Generations

4.2.3 Initial Populations with Seed Individuals

4.2.3.1 Senet

We evolved five populations of 256 individuals for 100 generations. Each initial population contained 248 randomly generated individuals and eight seed individuals. The seed individuals were two copies each of HC1, HC2, HC3, and HC5. For each run, the randomly created part of the initial population was different than any of the others.

Against the randomly generated and randomly moving individuals, the best individuals (once again, the winners of the last tournament of the last generation) from these populations performed extremely well, with some individuals scoring a perfect 200 out of 200 games played against some of the test players. For the most part, these individuals were typically winning between 80% and 90% of their games against the random individuals. Detailed results can be seen in Figure 4.23. Against the handcrafted heuristics from which

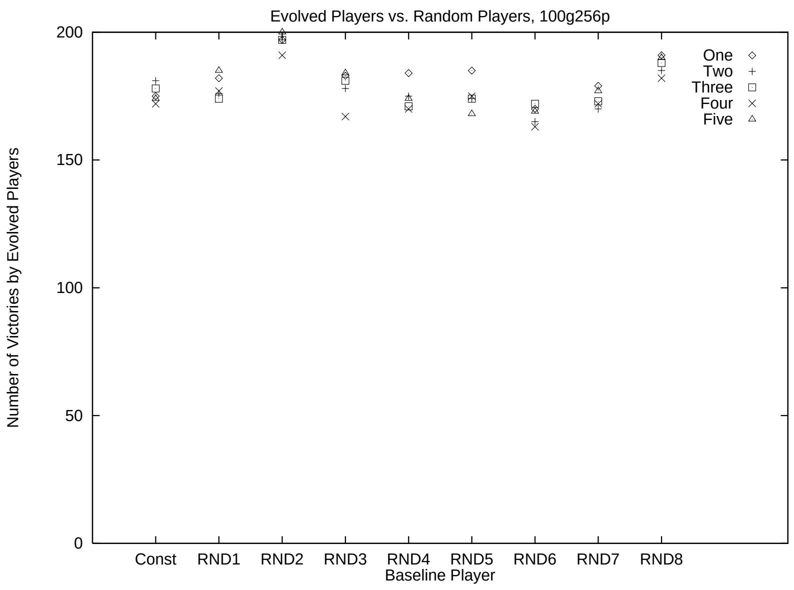

Figure 4.23: Seeded Evolved Senet Players vs. Random Players

these individuals were evolved, performance remained very good, as can be seen in Figure 4.24. Individuals typically won around 65% to 80% of games against these test heuristics.

4.2.3.2 Othello

We evolved two populations of 256 individuals for 100 generations. Each initial population contained 248 randomly generated individuals and eight seed individuals. The seed individuals were two copies each of OHC1, OHC3, OHC4, and OHC5. (The slow running time of our implementation precluded more than two runs of this experiment.)

Against the randomly generated baseline players and the randomly moving player, the winning individuals from these populations performed reasonably well, typically winning about 60% to 75% of the time, as can be seen in Figure 4.26. Figure 4.27 shows that against the handcrafted baseline players, performance was not good at all. For the most part, the winning individuals from these seeded populations did considerably worse than the seeds.

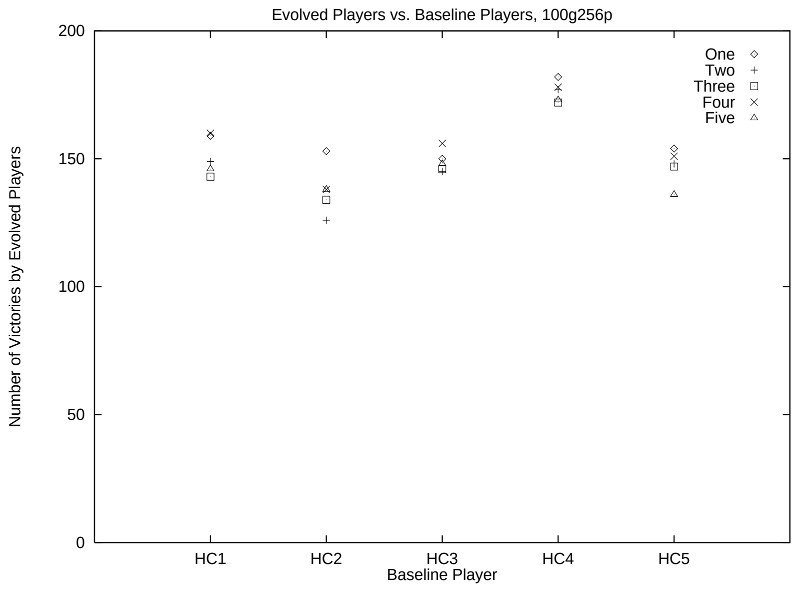

Figure 4.24: Seeded Evolved Senet Players vs. Handcrafted Heuristics

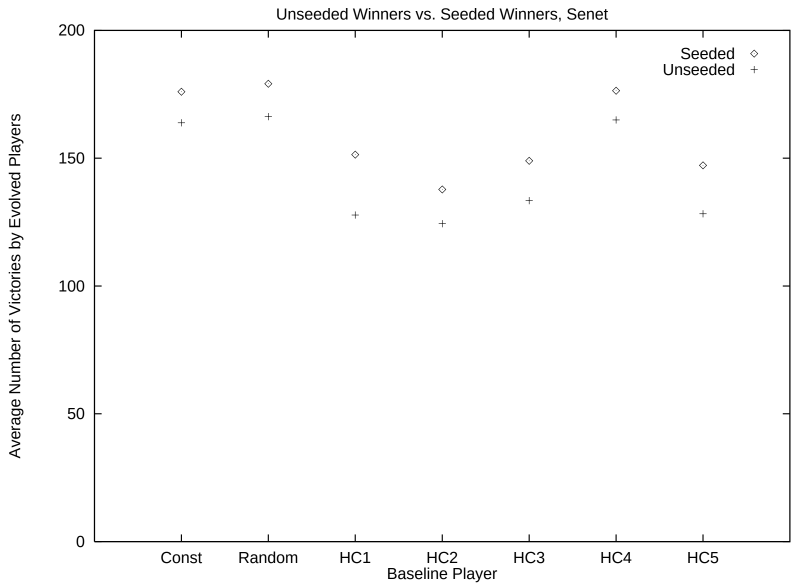

Figure 4.25: Comparison of Winning Senet Individuals Evolved With Seeds to Those Evolved Without Seeds

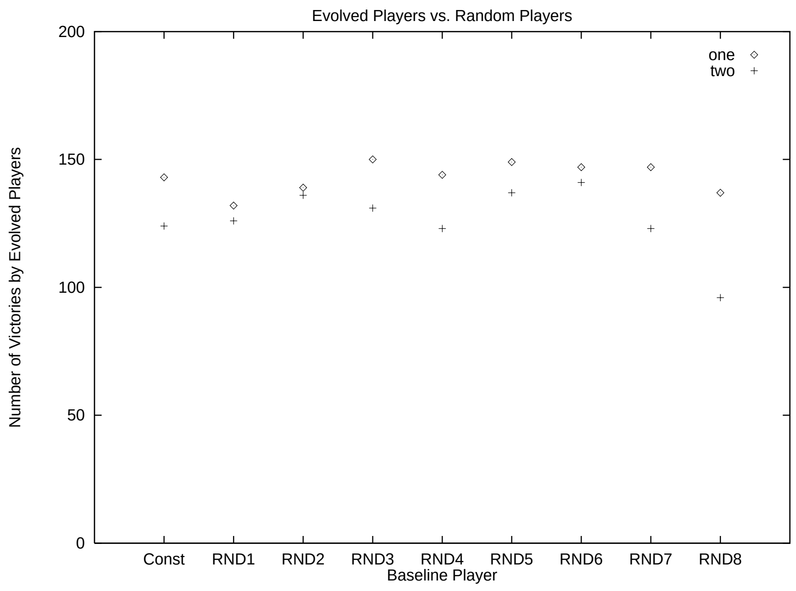

Figure 4.26: Seeded Evolved Othello Players vs. Random Players

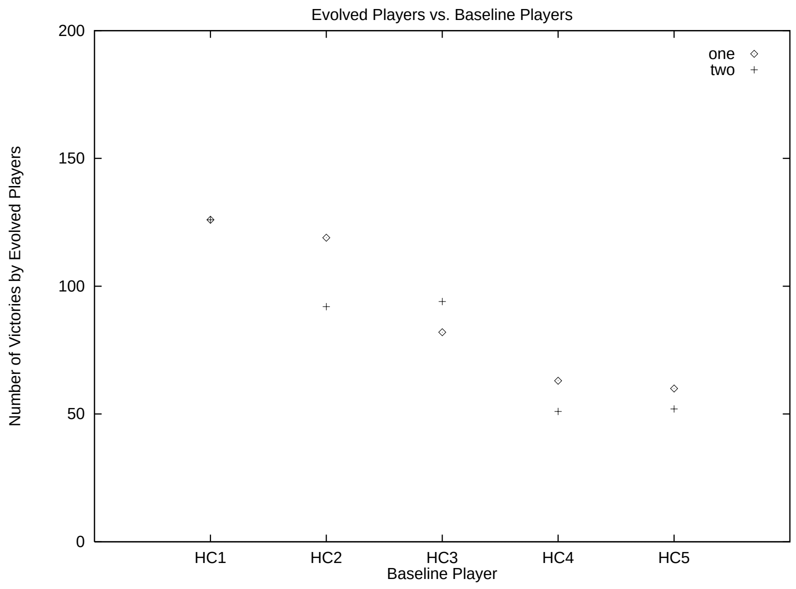

Figure 4.27: Seeded Evolved Othello Players vs. Handcrafted Heuristics

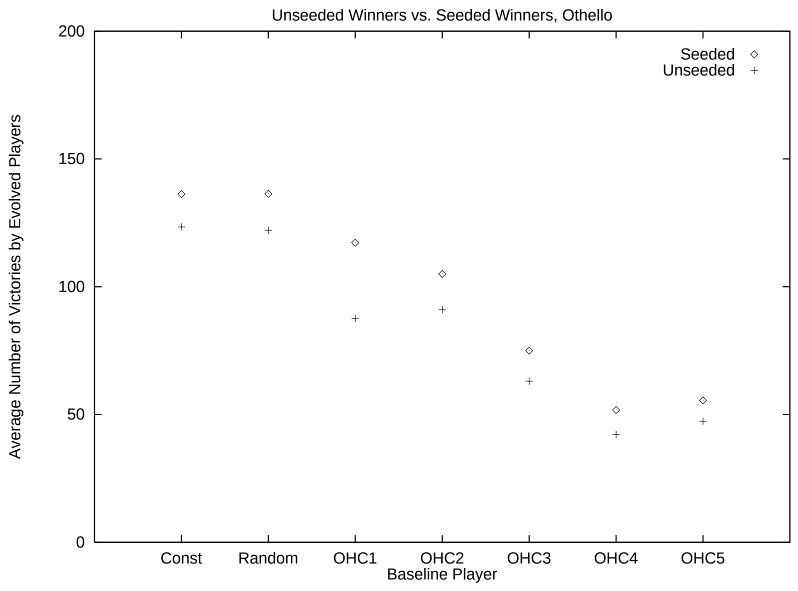

Figure 4.28: Comparison of Winning Othello Individuals Evolved With Seeds to Those Evolved Without Seeds

5

Conclusion

5.1 Contributions

We have shown that using genetic programming has the potential for being a viable technique for devising board evaluation functions. The genetic programming framework enables the functions to evolve towards whatever organizational structure is best suited to the problem at hand, without being limited to simple linear combinations of features. It is possible to incorporate any features desired into this framework easily either by including them in the non-terminal and terminal set or by including some number of seed individuals in the initial population.

We devised a testing procedure to evaluate the quality of the results of the evolutionary process. This testing mechanism is independent of the fitness function. Using this mechanism, we were able to show that for the game of Senet, players were evolved which play the game convincingly well. For the game of Othello, the evolved players were able to defeat random players a majority of the time, although against more sophisticated players they encountered some trouble. These results did show that the GP formulation for evolving board evaluation functions can be used effectively for more than one game, which we believe to be evidence of its general applicability.

Every individual in our formulation is a feasible player of the game for which it is being used. Formulating each individual as a board evaluation function enables us to avoid the issue of infeasible individuals completely. A default behavior was devised to deal with the condition of multiple "best" board configurations.

We experimented with population seeding, adding a small number of handcrafted individuals to our base population. In the case of Senet, the result of this was the evolution of individuals who outperformed both the seed individuals and the individuals evolved from purely random initial populations. In the case of Othello, some performance improvement over the individuals evolved from purely random initial populations was evident, while improvement over the seed individuals was conspicuous by its absence.

We observed certain population dynamics of interest as evolution progressed. In the cases of both Senet and Othello, after a point the quality of the winning individuals of each generation began to stagnate. In both cases, relatively low quality individuals would sometimes win the fitness tournament for a generation, only to return to obscurity shortly thereafter. In a few cases, particularly with Othello, relatively high quality individuals would win the fitness tournament and then apparently fade away, with the winners of subsequent generations displaying inferior performance.

5.2 Future Work

There are a number of possible future extensions of this work under consideration. A nice feature of Samuel's work is that it can learn from directly playing a human player. A disadvantage of this work is that once the evolutionary process completes, the players are unchanging. As a result, after playing a given player a number of times, it can become predictable and easier to defeat. Devising a mechanism to enable the evolved players to improve as a result of play against humans would help circumvent this problem and make the evolved players more challenging opponents.

An interesting extension to this work would be to experiment with better ways of better preserving higher quality players while still maintaining the good properties of the fitness function. A possible cause of this problem may be the disruptive nature of the genetic operators we implemented. The mutation rate is relatively high, and most of each new population is created by crossover rather than reproduction. Consequently, the selection mechanism may give too low a chance for good individuals to reproduce themselves.

It is also possible that the tournament structure for fitness evaluation may need some modification. For instance, perhaps more than four games should be played at each level of the tournament. Another possibility to consider is the issue of what the system should do when more than one board position gets the best evaluation number. Currently, each of those positions has the same probability of getting selected in that circumstance. It may be worthwhile to experiment with a deterministic strategy for handling this problem. The non-deterministic approach appeared to work well with Senet, but that game has an element of chance which is absent from Othello. In general, more experimentation could probably provide more insight in improving the results obtained from this general approach.

As was discussed in the results section, varying the parameters of the formulation might be a route to better results, particularly for Othello. Altering the default behavior invoked in the case of multiple board evaluations returning the same number is one possibility. Altering the tournament framework is another. In general, finding ways of accelerating the speed of the evolutionary process would be very helpful in giving us the opportunity to run more experiments with a wider range of parameters.

Another possible extension would be to improve the non-terminal and terminal set used in the Othello formulation. The only operator currently in place to evaluate stability works by checking to see how many corners are occupied, as corners are always guaranteed stable. Incorporation of additional stability operators could well improve performance, as the Othello literature makes it clear that this is a strategic consideration of paramount importance.

A

The Rules of Senet

The original rules of Senet are presently unknown, but a good approximation of what they were like is not impossible to find. Versions of the game have been found with anywhere from 5 to 10 pieces per side, although 7 pieces per side appears most common, and is the number used for each side in this implementation. For this implementation of Senet, the rules were based upon the rules developed by Timothy Kendall for a version of Senet he published as a boardgame [11]. Here is a summary of Kendall's rules, paraphrased directly from his rule booklet:

Set-up The 7 spools and the 7 cones are set up alternating with each other in the first 14 squares of the board. The pieces advance in an S-shaped course along the board, going left-to-right in the first row, right-to-left in the second row, and left-to-right again in the third row. A picture of the Senet board at the start of the game can be seen in figure A.1.

Objective To move all of one's pieces off the board before one's opponent succeeds in doing so.

Determining the Moves There are 4 throw-sticks used in the game, each one of them bearing a marked and an unmarked side. There are 5 possible resulting configurations, which have the following results: (1 = marked side up, 0 = unmarked side up)

1110: Move one house and get an extra throw.

1100: Move two houses. End of turn.

1000: Move three houses. End of turn.

0000: Move four houses and get an extra throw.

1111: Move five houses and get an extra throw.

A player should not move until all throws are completed. When the throwing is done, the throws may be used in any order or combination desired, one at a time. Once a throw is used, it is gone.

The opening move The players alternate throwing the sticks, handing them over to the other player upon a throw of 2 or 3. The first player to throw a 1 has the cones as his pieces and moves first. His opponent has the spools as his pieces and begins playing on the next turn.

Moving, Hitting, and Defending Since no two pieces may simultaneously occupy the same house, a player may only move a piece on his turn to a vacant house or a house occupied by an undefended enemy piece. A defended piece is a piece which has at least one of its friends in a house either immediately preceding or following it. When an undefended piece is hit, it trades places with the opposing piece which attacked it.

Forced Moves If a player cannot use a throw to move a piece forward, then he must move the piece backward the appropriate number of houses. If this causes the piece to land on an opposing piece, defended or not, the opposing piece gets to move forward at the original piece's expense. If there are no possible backward moves, the remaining throws are forfeit.

Endplays To bear pieces off the board, the player must deal with special rules governing movement through houses 26 to 30. To move a piece beyond house 26 (the "Beautiful

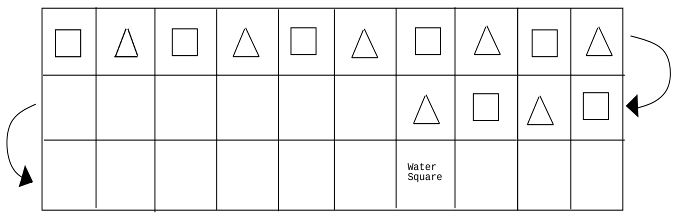

Figure A.1: The Senet Board

House"), it must first land there by exact count. It may be subsequently carried forward as follows: On a throw of:

5: Bear off at once.
4: Move to house 30. Bear off on any future throw.
3: Move to house 29. Bear off on a future throw of 2.
2: Move to house 28. Bear off on a future throw of 3.

None of a player's pieces can be borne off the board until the player has completely vacated the first ten squares. Once a piece makes it into one of the above houses, it is no longer required to travel backwards as described earlier.

The Water House This house is a pitfall that should be avoided if possible. A piece can land here either by moving 1 from house 26 (usually avoided, since a throw of 1 always enables another throw) or by getting bumped there by an opponent going from house 26 to one of the end houses. When a player has a piece in this house, he loses a turn. If he throws a 4 on his next turn, he may bear the drowned piece off the board. Otherwise, the drowned piece is transported to house 15, the "House of Repeating Life", or the nearest vacant house preceding it. In addition, while a player's piece is drowned, none of his other pieces are considered defended.

As previously mentioned, the above rules are based upon Kendall's published version of Senet. A few rules were very slightly changed to simplify some of the programming. The above set of rules is what was implemented.

B

The Rules of Othello

Othello is a two player game of strategy which bears some relation to the Go family of games, in that the emphasis is on capturing territory through the process of surrounding opponent's pieces. It is played on an 8×8 board, with a set of discs which are black on one side and white on the other side. In the initial board configuration, white owns the two central squares on the main diagonal and black owns the two central squares on the minor diagonal. This is shown in figure B.1. Players take turns moving until neither side has a legal move, with black playing first. When the game ends, the player possessing the largest number of discs is declared the winner.

A move is made by placing a disc on the board, with the player's color facing up. In order for the move to be legal, the square must be empty prior to making the move and placing the disc must capture at least one of the opponent's discs. Enemy discs are captured by bracketing them between the disc being played and an existing disc belonging to the player. The player's discs must be immediately adjacent to the enemy discs on both sides. Bracketing can occur in a straight line in any of the eight directions and consists of an arrangement starting with one of the player's discs, immediately followed by one or more of the opponent's discs, followed by the disc now being played. All of the bracketed discs are captured from the enemy player's color to the current player's color. A placement

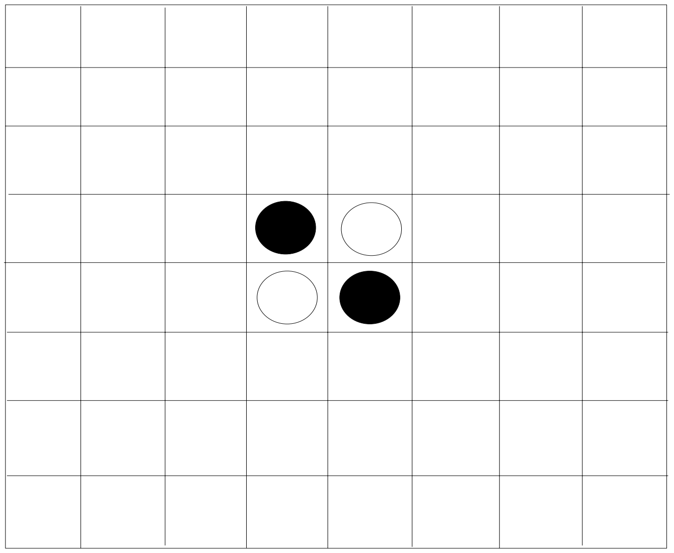

Figure B.1: The Othello Board

of a disc can result in flips occurring in multiple directions.

When a player has no legal moves available, the turn is forfeited and the other player gets to move. When neither player has a legal move, the game ends and the score is totalled. Normally, this occurs when the board has been completely filled with pieces, although this is not necessarily the case.

C

Computerized Othello Playing

This appendix is based on the work described in two major papers discussing computerized Othello players, namely Rosenbloom's program Iago [18] and the program of Lee and Mahajan, Bill [15].

C.1 Basic Othello Strategy

C.1.1 Trivial Othello Strategies

The most obvious strategy is to try to greedily accomplish the top-level Othello goal of having more pieces than one's opponent. This seemingly intuitive strategy works pretty badly in practice. Taking squares without regard to their strategic value often spells doom in Othello. This leads to the second simple strategy, that of weighting the squares depending on their typical strategic value. Its failings are twofold: there are more important reasons for taking or avoiding squares than just their location, and the value of a location may change during the course of play.

C.1.2 Mobility and Stability

Certain strategic considerations have been shown to be of immense value when utilized in computer Othello players. Two of the most studied strategic considerations are *mobility* and *stability* [18] [15]. For example, a failing encountered by both of the above strategies is that they tend to sacrifice mobility for the sake of more pieces or possessing a certain location. Mobility is an important strategic concern in Othello, since a lack of flexibility in choice of moves tends to reduce the possible number of good moves a player can make, which tends to result in the deterioration of the player's position. In the extreme case a player with no mobility is unable to move and can readily be abused by its opponent. There is more to mobility than maximizing the difference in the current number of moves available to the players, in that having a large number of bad moves available is not necessarily any better than having a small number of bad moves from which to choose. Additionally, maximizing long-term mobility advantage may involve examination of board features other than mobility.

As the game closes, maximizing the disc differential becomes a paramount concern. However, flipping a disc over does little good if an opponent can soon thereafter flip it right back. Hence the importance of analyzing the *stability* of discs. A disc which is completely stable can never be recaptured by an opponent.

To prove that a disc is stable, it must be shown that the disc can never be bracketed on one side by an opposing piece and on the other side by a blank square (along the same line). An example of a stable piece is any occupied corner square. They are consequently very important not only because of their own stability but additionally because they can form an anchor by which other discs may become stabilized. This phenomenon is easiest to see with the non-corner edge squares. The only line of possible instability is the edge itself, so to remove this instability all that is needed is to have a stable disc of the same color immediately adjacent along the edge. Internal discs are very difficult to stabilize until late in the game, since there are four sources of potential instability for each disc: horizontal, vertical, and two diagonal.

C.2 Computing Othello Board Evaluation Functions

C.2.1 Computing Mobility

There are two ways of considering a player's mobility for a particular board configuration. *Current mobility* considers which moves are legal in the present configuration. *Potential mobility* attempts to discern moves that may later become legal but are not yet so.

C.2.1.1 Current Mobility

One approach to computing current mobility is to simply count all legal moves currently available. One problem with this is that no distinction is made between "acceptable" and "unacceptable" moves. This is a problem in that having seven possible moves all of which yield a corner to the opponent is considered "better" by an algorithm not making this distinction than having three moves available with less disastrous consequences. Rosenbloom [18] discusses three possible approaches for dealing with this problem. One approach is to search the game tree below each move. This is essentially just adding another search level and if we assume that the search is already as deep as possible, this approach is necessarily too expensive. A second alternative is doing an a priori classification of whether it is "acceptable" to move to a particular square. The alternative preferred by Rosenbloom is to retain the simple count of the total number of legal moves and add a second measure which eliminates moves which result in immediate surrender of a corner.

In Bill [15], the computation of current mobility incorporates several additional criteria for determining which moves are "acceptable" or "good". These criteria include considering whether a corner is captured or surrendered, how many discs get captured (and in how many directions), and whether those discs being captured are internal or on the frontier. First, the total number of legal moves available in a position is counted, and then penalties are assessed based on whether some of those moves adversely affect the aforementioned criteria. Both move counting and move penalty are calculated by a series of table lookups. A board is represented for this purpose as 38 numbers, where each number is an index into a table that represents a horizontal, vertical, or diagonal line on the board.

C.2.1.2 Potential Mobility

In Iago [18], the key idea for computing potential mobility is to find a string of the opponent's discs with an empty square at one end. This is useful because if there is a way for the current player to place a disc at one end of this string of discs, then a new move will be available on the player's next turn. Iago's board evaluation function uses three measures to attempt to determine the potential mobility of a given board configuration. It adds together the number of opponent discs next to empty squares, the number of empty squares adjacent to opponent discs, and the sum of the number of empty squares adjacent to each of the opponent's discs. The combination of these measures gives Iago a pretty good idea of the potential mobility of the board configuration being examined.

In Bill [15], potential mobility is computed in a manner similar to how current mobility is computed. Tables are generated for each of the 38 lines, with bonuses given for each enemy internal disc which is next to an empty space. The size of the bonus is naturally dependent upon the desirability of the potential move, using criteria similar to those used in the current-mobility computation.

The *sequence penalty* is an additional computation used to supplement Bill's mobility estimation. It is based on the observation that it is in general not a good practice to have long sequences of one's own discs. They often hinder one's mobility. Bill penalizes long strings of discs for this part of its evaluation function. The penalties are pre-computed for each configuration of each of the 38 possible lines, as with the other mobility features of Bill discussed above.

C.2.2 Computing Stability

The simplest method for computing stability is to determine how many corner squares are occupied, as each of these squares is automatically stable once a disc is placed thereupon. Other simple methods of computing stability involve determining which discs on an edge are adjacent to discs of the same color which ultimately are adjacent to discs occupying corners. Because of the large number of possibilities which begin to emerge, Iago and Bill rely upon tables for computing stability.

To compute edge stability, Iago relies upon a precomputed table of all possible edge configurations. It then evaluates each edge by looking it up in the table, which will contain the stability value of that particular configuration. With a total of 6561 (3^8) possible edge configurations, a lookup table is not an unreasonable approach. This table gets precomputed using an iterative algorithm. The table is initialized with a set of static values depending on the location and stability of each disc in each configuration. Each iteration of this algorithm starts with a completely filled configuration and works backwards to intermediate positions by removing discs. The intermediate positions are evaluated by computing the expected value of the position. For each empty square, the value of playing there is multiplied by the probability of being able to legally make that move to determine the overall value of that configuration. An iteration is complete when a value has been assigned to the empty configuration. The table resulting from this process is used by finding the entry for the current edge position in the table, and returning the value corresponding to that entry to the board evaluation function.

Computing internal stability is rather difficult, so Iago uses a somewhat ad-hoc algorithm that checks for a subclass of stable internal squares. The algorithm essentially checks to see if there is a friendly stable disc in each of the four potential directions from which it could potentially be captured. The author notes that the utility of this algorithm is marginal, in that this particular subclass of stable internal squares is not large.

The edge evaluation process used by Bill is reminiscent of Iago's, except that the authors of Bill decided to include the "X squares" as part of the edge in their lookup table. The "X squares" are the four squares on the Othello board diagonally adjacent to the corner squares. Each edge in Bill's lookup table contains 10 squares: the eight squares which comprise the edge itself, and the "X squares" corresponding to each corner in the edge. Consequently, Bill's edge lookup table contains 59041 (3^{10}) entries. Including the "X squares" compensates for weaknesses in Iago's edge evaluation function resulting from the fact that these squares are an excellent staging ground for corner attacks. To compute the edge table, an algorithm resembling that of Iago's is used, which recursively searches every possible edge position to determine its value, using probabilistic weights in a manner similar to Iago. Bill does not bother with any computations of internal stability, given the observation in Iago that it was of little real value and also given the more general observation that the key to achieving stable internal discs is to stabilize the edges.

C.2.3 Complete Board Evaluation Functions

Based on the considerations of mobility and stability, Rosenbloom formulated a single board evaluation function for his Iago program. It is a linear polynomial constructed from four terms representing edge stability, internal stability, current mobility and potential mobility each multiplied by various coefficients which can change depending on the number of the current turn. The edge stability coefficient is particularly large. The coefficients in Iago were selected by first assigning values based on opinions of the relative importance of the terms. A small set of variations on these coefficient values was evaluated by having the different versions of Iago play against each other. The best variation became the official Iago evaluation function.

Bill's evaluation function is composed of terms representing edge stability, current mobility, potential mobility, and the "sequence penalty". As described above, computation of all of these features is done by table-based algorithms for the sake of speed.

To combine these four features, Bayesian learning [6] is used to combine the features "optimally" into a quadratic polynomial. This technique is often used in pattern recognition to classify an image based on features extracted from it. In the case of Othello, a board position is classified as a "win" or "loss" based on features extracted from the board. The algorithm for training has four steps. First, a large database of training positions was generated. This was achieved using an earlier version of Bill. Second, these positions were labeled as winning or losing, depending upon whether the player who won or the player who lost made the move which generated the position being examined. Third, a discriminant function was computed from the labeled data. This function attempts to recognize feature patterns that represent winning or losing positions. Given the feature vector for a position, it assigns a probability that the position is a winning one. Finally, different classifiers are built for different stages of the game.

The training positions were generated by an earlier version of the Bill program, Bill 1.0. Bill 1.0's evaluation function combined the features given above linearly, tuning the coefficients by playing against Iago. Bill 1.0 was the winner of several computer Othello tournaments, and was thus considered an expert Othello player by its authors. Using Bill 1.0 to generate training positions thus guaranteed that all of the generated positions come from expert games, according to the authors. For training purposes, the game of Othello was divided into stages by observing that there are nearly always 60 moves per game. Consequently, 60 discriminant functions were generated, with each one being generated from training data with plus or minus two discs. This coalescing of adjacent data makes the discriminant function slow-varying. To compute the discriminant function itself, the feature vector gets extracted from each position. The mean feature vector and covariance matrix are then computed for the win and loss classes. The quadratic discriminant function is a polynomial combination of the feature vector (containing the four features Bill uses which were discussed above), the covariance matrix, and the mean feature vectors for wins and losses.

C.3 Conclusion

Bill and Iago have both performed impressively. Iago was victorious in the Santa Cruz Open Machine Othello Tournament [8] without losing a single game against an international field of computer Othello programs. Jonathan Cerf, who was at the time the current Othello world champion, stated in a review of the tournament [5] that "In my opinion the top programs from Santa Cruz are now equal (if not superior) to the best human players." In that same article, he goes on to state:

> I understand Paul Rosenbloom is interested in arranging a match [for Iago] against me. Unfortunately, my schedule is very full, and I'm going to see that it remains that way for the foreseeable future.

Bill 3.0, which is the version of Bill described above, defeated Brian Rose, who at the time was the highest rated American Othello player, by a score of 56-8. Against Iago, Bill 3.0 did not lose a single game. Lee and Mahajan do not state the total number of games played between the two programs.

Given the performance of Iago and Bill, and given the features of Othello emphasized in their board evaluation functions, it would seem that the principal strategic considerations to be considered in Othello are stability and mobility. The use of tables facilitates accessing large amounts of precomputed useful strategic information quickly enough to be useful in play.

D

Other Methods for Evolving Game Players

A number of researchers have previously applied evolutionary approaches to the problem of playing a boardgame. Angeline and Pollack [1] used genetic programming to evolve computer programs which played the game tic-tac-toe. This project did not, strictly speaking, evolve board evaluation functions, since the programs also were responsible for "physically" moving pieces on the board. This allowed the possibility that a player would fail to move a piece on its turn, which adds the potentially undesirable possibility of infeasible individuals who don't actually move on every turn, thus not playing a legal game.

Rosin and Belew [19] used a coevolving method for creating tic-tac-toe players. One population was the *host*, and the other the *parasite*. Fitness for the host population is evaluated by having each member of the host population play against some number of individuals from the parasite population. The populations then trade roles, so that the members of the (former) parasite population can have their fitness evaluated. To represent tic-tac-toe players, they represent every legal, reachable board position in the genome. Even eliminating redundant positions (due to symmetry), the genome still has 593 positions. Each position is represented as a 9 digit base 3 number, with a 0 digit representing an empty square, a 1 to indicate an opponent occupation, and a 2 indicating that this player holds the square. Each gene in the genome takes on a number of allele values equal to the number of open squares in the corresponding position. This uniquely specifies a legal move from each position.

The Rosin/Belew representation described above has the advantage of being fixed in size, thus avoiding memory consumption problems sometimes experienced using genetic programming [7]. However, using computer programs as the representation allows for much greater scalability, in that a program doesn't have to internally represent every possible board position. It is difficult to imagine using the Rosin/Belew representation for a game such as chess, for instance.

Another more conventional genetic algorithm representation is that used by Sun and Wu [23] for the game of Othello. They evolved the numerical coefficients of a linear board evaluation function. The terms of the function are features of the current state of the game. In a previous paper [22] they mention that the features used are a board position measurement, current mobility, stability, and two versions of potential mobility. This problem formulation means that the evaluation function will necessarily be a linear polynomial, which limits the flexibility of possible evaluation functions that can be evolved by this method.

In order to evaluate the fitness of individuals, each individual played one game against each of five "coaches". The coaches were pre-selected expert Othello evaluation functions. The fitness of each individual was the sum of the square advantages of the five games. After around 20 generations, the original coaches get replaced by the best individuals in the population, and as time marches on those new coaches in turn get replaced by new superior individuals from the population being evolved.

Moriarty and Miikkulainen used an approach based on the artificial evolution of neural networks to develop a board evaluation function for Othello [17]. The networks were afforded no search mechanism, and consequently were forced to rely solely on pattern recognition of the current board configuration to achieve good play. Fitness was evaluated by playing each individual against a standard opponent. The fitness value was the number of games out of ten in which the individual emerged victorious. The networks were first evolved against a player which moved randomly, and subsequently against a player utilizing three levels of alpha-beta search with a positionally based evaluation function. No preselected features of any sort were presented to the individuals.

After the evolutionary process was complete, analysis of the play of the top individuals demonstrated that they had independently developed a mobility-based strategy for Othello play. They evidenced this strategic behavior not only against the individual against which they were evolved, but also against other players. A top Othello player agreed that the individuals evolved using this scheme were fairly impressive considering that they started with random initial weights.

The principal weakness of using a neural network based representation is that if it is considered desirable to incorporate preselected game features, there is no straightforward mechanism present for doing so. The principal method available would be to insure that at least some of the individuals in the initial population have been trained in Othello play by some other means. Thus, an advantage of the genetic programming formulation is that features are easily incorporated as desired simply by placing them in the function and terminal set.

E

Sets of Terminals and Non-terminals

E.1 Non-terminals and Terminals Common to Both Games

- **Arithmetic Operators**: The arithmetic operators `plus`, `minus`, and `times` are available in both games. They each take two arguments and return the sum, difference, or product.

- **Decision Operator**: The `if` operator takes three arguments: a condition and two possible actions. If the condition reduces to 0 (i.e. false) the second action is taken, otherwise the first action is taken.

- **Logical Operators**: The `and`, `or`, and `not` operators take two arguments and return an integer representing the appropriate logical combination of those arguments.

- **Integers**: The terminal sets of both games contain the integers from 0 to the number of board locations minus 1.

E.2 Non-terminals and Terminals Specific to Senet

- **Throws Left**: The terminals `5-left?`, `4-left?`, `3-left?`, `2-left?`, and `1-left?` return how many of each sort of throw the current player has remaining.

- **Move Information**: The terminals `get-start` and `get-distance-moved` report information on the move the function just made.

- **Data on Pieces**: The terminals `self-pieces-left?` and `foe-pieces-left?` report how many pieces a player has remaining. The terminals `self-drowned?` and `foe-drowned?` report whether a player has landed a piece on the water pitfall. The non-terminals `self-nth-piece?` and `foe-nth-piece?` take one argument n and return the board location of piece n of the appropriate player.

- **Data on Board Locations**: The non-terminals `occupied?`, `friend?`, `foe?`, and `defended?` take one argument which is a board location and return a value indicating whether that predicate is true for that board location.

E.3 Non-terminals Specific to Othello

- **Comparison Operators**: The non-terminals `>`, `<`, and `=` compare their two arguments and return 1 if the test is true and 0 if it is false.

- **Data on Board Locations**: The non-terminals `empty?`, `me?`, `enemy?`, `legal-self-move?`, and `legal-enemy-move?` take one argument which is a board location and return a value indicating whether that predicate is true for that board location.

- **Data on General Game State**: The non-terminals `num-legal-moves` and `num-pieces` take one argument which if an odd number refers to the current player's opponent and if an even number the player proper. They then return the appropriate number, as their names imply.

- **Stability Data**: This terminal was absent from the first round of Othello experiments. The `num-corners-occupied` non-terminal takes one argument which if odd refers to the opponent and if even the current player. This non-terminal then returns the total number of corners occupied by the indicated player.

Bibliography

[1] Peter J. Angeline and Jordan B. Pollack. Competitive environments evolve better solutions for complex tasks. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 264–270, 1993.

[2] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, 1985.

[3] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence, Volume 1.* William Kaufmann, Inc., 1981.

[4] Hans Berliner. On the construction of evaluation functions for large domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 53–55, 1979.

[5] Jonathan Cerf. Machine vs. machine. *Othello Quarterly*, 2(4):12–16, 1980.

[6] R. Duda and P. Hart. *Pattern Classification and Scene Analysis.* Wiley, New York, 1973.

[7] Gabriel J. Ferrer and Worthy N. Martin. Using genetic programming to evolve board evaluation functions. In *Proceedings of the 1995 IEEE Conference on Evolutionary Computation*, pages 747–752, Perth, Australia, November 1995. `http://www.cs.virginia.edu/g̃jf2a/work/papers/senet.ps`.

[8] P. W. Frey. The Santa Cruz open Othello tournament for computers. *BYTE*, 6(7):26–37, 1981.

[9] J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, 1987.

[10] John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[11] Timothy Kendall. *Passing Through the Netherworld.* Kirk Game Company, 1978.

[12] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th Joint Conference on Artificial Intelligence*, pages 768–774, 1989.

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

[14] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, 1994.

[15] Kai-Fu Lee and Sanjoy Mahajan. The development of a world-class othello program. *Aritificial Intelligence*, 43:21–36, 1990.

[16] David Levy. *Computer Gamesmanship.* Simon and Schuster, Inc., 1983.

[17] David E. Moriarty and Risto Miikkulainen. Discovering complex othello strategies through evolutionary neural networks. *Connection Science*, 7(3):195–209, 1995.

[18] Paul S. Rosenbloom. A world-championship-level Othello program. *Artificial Intelligence*, 19:279–320, 1982.

[19] Christopher D. Rosin and Richard K. Belew. Methods for competitive co-evoltion: Finding opponents worth beating. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 373–380, University of Pittsburgh, July 1995.

[20] Arthur Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development 3*, pages 210–229, 1959.

[21] C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275, March 1950.

[22] Chuen-Tsai Sun, Ting-Hong Liao, Jing-Yi Lu, and Fu-May Zheng. Genetic algorithm learning in game playing with multiple coaches. In *Proceedings of the 1994 IEEE Conference on Evolutionary Computation*, pages 239–243, 1994.

[23] Chuen-Tsai Sun and Ming-Da Wu. Self-adaptive genetic algorithm learning in game playing. In *Proceedings of the 1995 IEEE Conference on Evolutionary Computation*, pages 814–818, Perth, Australia, November 1995.

[24] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[25] Astro Teller. Turing completeness in the language of genetic programming with indexed memory. In *IEEE World Congress on Computational Intelligence*, 1994.

[26] Gerald Tesauro. Temproal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, March 1995.