Using a Parallel Genetic Algorithm to Evolve a Better Static Evaluation Function for Hawaiian Checkers

Matt Coddington

December 17, 1998

Abstract

This project's main goal is to evolve a static evaluation function for Hawaiian Checkers that is better than the 'number of black moves minus number of white moves' heuristic that is currently used. The first two sections of this paper describe Hawaiian Checkers, the current static evaluation function, and the motivation for writing a parallel GA to solve this problem. Section 3 describes the GA in detail including its fitness function, crossover technique, selection method, and mutation method. It concludes by describing the parallel implementation. Section 4 reports on the results of the project, including the fitness of the final result from the GA and the increase in efficiency of the GA due to parallelism. Two appendices list the machines used in the parallel scheme and the best member of the final population.

1 Playing the Game

Hawaiian checkers is played on an 8x8 board between two players. There are two colors of pieces on the board, black and white, and each player controls one set of pieces. The game is set up by placing the black and white pieces on the board in a checkered pattern so that every square is filled (see figure 1). It will be convenient to think of board positions in x,y coordinates. Let $(1, 1)$ be the lower-left corner of the board. The first two moves of the game work as follows: On the first move, the black player removes either one of his pieces from the middle or from the corner (position $(1, 8), (8, 1), (4, 5)$, or $(5, 4)$). Then the white player may remove a white piece adjacent to the piece black removed. After these first two moves, the game continues with black and white alternating turns jumping over one anothers' pieces. A jump is either horizontal or vertical and can be a double or triple jump as long as the jump does not 'turn corners' (see figure 2). When a player can no longer jump, that player loses.

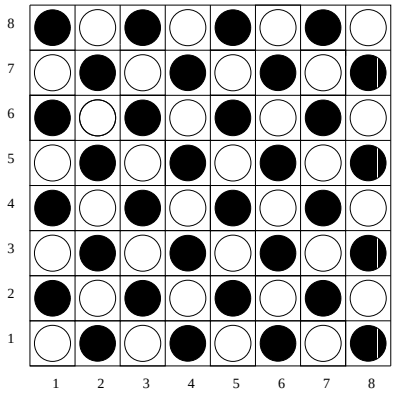

Figure 1: Initial board setup

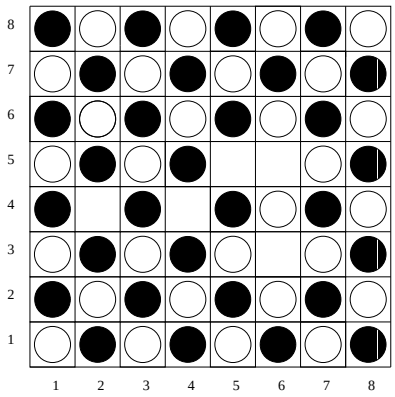

Figure 2: A sample board situation. Black can make a double jump from $(6,7) \rightarrow (6,3)$, but it is illegal for him to "turn the corner" jumping from $(8,5) \rightarrow (6,5) \rightarrow (6,3)$.

2 The Existing Strategy

Our program that plays Hawaiian Checkers currently uses the minimax algorithm with alpha-beta pruning to decide its moves [Nil98, pp 197–206]. The minimax technique is effective for this sort of game playing because it assumes that both players are trying to maximize their own position or minimize their opponent's. It does this by constructing a tree of the possible game states up to n moves ahead[1]. It gives every leaf node in the tree a value and then parses back up the tree, maximizing this value on the current player's turn and minimizing on the opponent's turn. The static evaluation function (from now on referred to as a "strategy") is the function that evaluates these leaf nodes. In the current program, the strategy is to subtract the number of moves white has from the number of moves black has. This value is maximized on black's turn and minimized on white's turn and resulted in a game that could beat inexperienced human players almost all of the time when looking four moves ahead. The goal of this project is to evolve, using a genetic algorithm (GA), a new strategy that can beat the black moves minus white moves strategy. In designing such a strategy, the fitness function of the GA needs to be parallelized so that it can finish in a reasonable amount of time (see 3.6).

3 Methods

3.1 Notation and Representation of Strategies as Strings

Members of the population in a GA are often represented as strings [Nil98]. Deciding how to represent strategies in the solution space as strings was the first step in the project. Many ingenious string/solution representations are shown in [Mit96]. The goal was to come up with a representation that didn't have too small of a search space or introduce too much programmer bias, but also one that could be encoded in a reasonable amount of memory and decoded into its phenotype (the actual strategy: a `C++` object in my case) in a relatively short time. In addition, it should lend itself to some meaningful crossover technique such that crossover between two fit members of the population usually results in another fit member. I believe the representation described below encompasses all of these goals. First, some notation needs to be introduced:

Let $\mathcal{N}$ represent the size of the board in rows (thus, there are $\mathcal{N}^2$ squares on the board). Each square on the board is represented by $S_{yx} \in \{EMPTY, WHITE, BLACK\}$. We associate each member of the preceding set with a number, specifically $EMPTY = 0$, $BLACK = 1$, and $WHITE = 2$. Using this representation, we already have a way to give a board a value simply by adding the board positions:

$$\sum_{i=1}^{\mathcal{N}} \sum_{j=1}^{\mathcal{N}} S_{ij} \tag{1}$$

[1]In our implementation of minimax, $n \geq 5$ resulted in evaluation times too high for interactive gaming. The GA uses $n = 3$ as a compromise between speed and quality of evaluation.

However, such a value is fairly meaningless as far as a strategy is concerned in that it just decreases over time as pieces are jumped and removed. A more strategic evaluation might sum together every square whose indices sum to an even number (the squares who can contain only $WHITE$ or $EMPTY$). This would, in effect, be counting the number of white pieces on the board. An even more complicated evaluation might sum together various squares and then subtract from that sum the sum of other squares, and so on. There is an infinitely large search space for this sort of strategy, and thus a GA is well suited for the job. Each member of the GA's population (also to as "chromosomes") consists of a string of objects (or "alleles" in genetic terminology) that can be defined as follows:

$$(R_0\ O_0\ (R_1\ O_1\ O_2)) \tag{2}$$

where $R_0 \in \{+, -, N\}$ (N represents a 'do nothing' or 'ignore' operator), $R_1 \in \{+, -\}$, and $O_i \in S_{yx}$. For example, $(N\ [0,0]\ (+\ [1,2]\ [4,3]))$ means "add together the values of board spaces $[1,2]$ and $[4,3]$." A chromosome is implemented as a linked list of these allele objects. Using a linked list allows the length of the chromosome to increase or decrease which gives it the ability to explore more of the search space. A specific board is evaluated by summing the results of all alleles in a particular chromosome.

3.2 Fitness Generation

Letting $\mathcal{P}$ represent the size of the population, the fitness of a chromosome is generated by having it play two games against each of the other $\mathcal{P}-1$ members of the population: one game as the white player and one game as the black player. In general, a round-robin tournament such as this will have $\frac{\mathcal{P}(\mathcal{P}-1)}{2}$ games, but since we are playing two games per matchup, we end up playing a total of $\mathcal{P}(\mathcal{P}-1)$ games to determine the population's total fitness. Each chromosome gets one point of fitness per win and no points of fitness per loss, so the maximum fitness one can receive (winning all games) is $2(\mathcal{P}-1)$. Parallelism was needed in this section of the code since each game took approximately 1.75 seconds to play on a serial machine[2].

3.3 Selection

Selection is done using a simple 'roulette wheel' selection algorithm. It selects a member of the population as a parent based on its fitness relative to the population's total fitness. Parents can be chosen multiple times in each generation, and identical parents can be chosen as well (effectively resulting in an exact copy of that parent after crossover).

3.4 Crossover

Crossover is between two parents, both selected using the selection function described in section 3.3. If both parents have the same number of alleles, the child has that number of alleles. The locus of crossover (the allele at which crossover takes place) is chosen randomly. If the parents are of different lengths, the locus of crossover is chosen randomly in the shorter parents' length. There is a 50% chance of the child being the shorter length and 50% of it being the longer length.

3.5 Mutation

There are four different types of mutation that can occur, divided into two separate mutation groups. The first mutation group, "little mutations" occur more often than the second group, "big mutations". Little mutations include the changing of a random operator or operand in an allele, while big mutations involve adding or deleting an entire allele from a chromosome. Little mutations do not change the chromosome as radically as the big mutations, so they are usually set to happen more often[3].

[2]Serial timings were taken on an Sparc Ultra10 with 128Meg RAM. At 1.75 seconds/game, calculating the fitness of a single generation for a 100 member population would take 4.8 hours!

[3]In most of my runs, the big mutation probability was set to 1% and the little mutation probability was set to 4%.

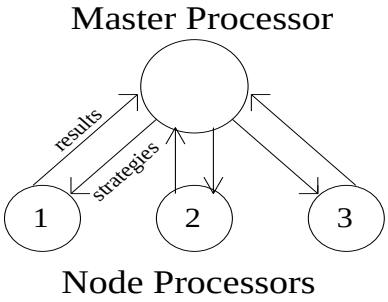

Figure 3: The parallel implementation scheme. Strategies are passed from the master processor to the nodes, where the games between strategies are resolved. Then the nodes send the results back to the master and await new strategies.

3.6 Implementing Parallelism

GA's lend themselves to parallelism because they work independently with an entire population of chromosomes during each generation. Parallelism was introduced in this project solely in the fitness generation phase of the GA [CP97]. This phase accounts for an overwhelming majority of the processing time compared to selection, crossover, and mutation[4].

The PVM [Ge+94] libraries were used as the parallel interface. PVM excels at message passing between machines in a networked, heterogeneous environment[5] so it was ideal for this project. The general parallel scheme was to distribute chromosomes to different machines and have them play games between those chromosomes, returning the result to a master process, which would then evaluate the result and send another group of chromosomes to the node machines (see figure 3).

All of the message passing is accomplished by passing arrays of integers. The master process sends the node an array containing both strategies encoded as integer arrays, and then the node sends back a five-integer array containing the following information: the index of player1 in the master's population array, the index of player2 in the master's population array, how many games player1 won, how many games player2 won, and the task ID of the node process itself. The master uses the task ID to determine who to send the next two strategies to. Thus, when it receives a response from any node, the master unpacks the array from the message and can immediately send another two strategies back to that particular node, keeping all nodes busy at all times. This scheme works much faster than a scheme that waits for all nodes to return before sending out another 'batch' of strategies since the node machines are all processing at slightly different speeds depending on their current load and their architecture.

4 Results and Discussion

4.1 Results Evolving a Better Static Evaluation Function

The generated algorithm beat a random move generator 17 out of 20 times (10 out of 10 times as black and 7 out of 10 times as white). It also beat me 5 out of 5 games and even beat the black minus white strategy as black, but not as white[6]. It is interesting to note that the black minus white strategy often chooses more efficient ways to win, beating the random move generator in less than 20 turns, while the evolved strategy usually takes 25 turns to beat anyone. This may be because the evolved strategy has come to value certain positions on the board and spends the game getting into those positions so that it can then execute a series of moves to win in the end.

The final strategy evolved by the GA is shown in Appendix B. I expected it to evolve such that it was adding the values of squares in specific rows and columns since those are the only valid places to jump. This did not happen,

[4]Fitness generation timings can be measured in hours on serial machines for decent sized populations, while selection, crossover, and mutation timings can be measured in fractions of seconds.

[5]The sun lab is a fairly homogeneous (although networked) environment, except for the fact that Allspice is of different base architecture than the clients (as far as PVM was concerned) since it has a different base instruction set to take advantage of its dual processors. PVM makes message passing between different architectures transparent to the programmer.

[6]In all of these games, the evolved strategy had a terminal depth of 3, as did the black minus white strategy.

however. Looking at the strategy, I cannot see any rhyme or reason to its method, but apparently it works well. This is the beauty of genetic algorithms: developing unique solutions that would not otherwise have been thought of.

4.2 Increased Efficiency from Parallelism

As was mentioned before, the game-playing algorithm took approximately 1.75 seconds to run on the fastest serial machine available[7]. The parallel algorithm was able to complete 52 generations in 10.17 hours with a population size of 50. This is equivalent to doing 3.48 games/second or a speedup of 6.1 times the original performance. It was run on 13 machines, but many of those machines were much slower than the benchmark serial machine. Because of the diversity in processor speed and minor differences in architecture it is difficult, if not impossible, to measure the overhead due to message-passing. However, we can take a weighted average of processor speeds and account for that in the speedup. This weighted average comes to 253.8MHz (for machine data see Appendix A). A perfect speedup would have been one of 13 times, since 13 machines were running in parallel. Thus, there is an unexplained slowdown of 53% in the parallel version. Taking into account the processor speeds, we can explain a little more than 15% of this slowdown. The rest is due to network latency, the added overhead of sending and receiving messages, and the presence of other users' processes on the various machines running in parallel (the benchmark tests were done on a machine with no other users at the time). Although the results were less impressive than those in [AP98] and below the theoretical expected efficiencies in [CP97], I believe the parallel implementation was still a success since it was able to reduce the run time of the algorithm from one month to five days.

4.3 Discussion

Many variables in this GA could have been changed to try to improve the results. I came up with the mutation and crossover probabilities through trial and error, with the goal being to keep a high diversity within the population while still having it increase in average fitness every generation. It is difficult to measure an average fitness when the fitness function is a round-robin tournament, since every member of one population might be able to beat every member in the previous population, but their total fitness is the same (the total fitness is the same for every generation because they get one point per game won, and there are a constant number of games played per generation). I tested a few members from later generations against members from previous ones, and they won 90% of the time.

To come up with ideal mutation and crossover rates, population size, and number of generations to run, one could use a "meta-GA" that uses the original GA as its fitness function. This was not an option in this experiment, however, since the GA itself took so long to run. Another way to make the final population more fit, might have been to seed the initial population with chromosomes that I knew to be fit. This adds programmer bias to the GA, possibly causing it to miss ideal solutions, but also making it more likely that it will find some sort of "best strategy" in a reasonable amount of time.

[7] A significant speedup was gained by using Sun's proprietary SparcWorks `C++` compiler instead of GNU's `g++`. Games were only being played at one game per 3.5 seconds before compiling with SparcWorks.

Appendix A
Data for Machines Used

Allspice was used as the master, and it was also serving as a node. PVM has multiprocessor support (for machines like Allspice), but I was not able to implement this support as the multiprocessor compiled binaries failed on allspice for some unknown reason.

Name	Arch/Speed	RAM	Model	IP Address
Allspice	sun4u, 2x248MHz	512M	E450	130.58.68.10
Tarragon	sun4u, 300MHz	128M	Ultra10	130.58.68.17
Sage	sun4u, 300MHz	128M	Ultra10	130.58.68.16
Parsley	sun4u, 300MHz	128M	Ultra10	130.58.68.23
Thyme	sun4u, 300MHz	128M	Ultra10	130.58.68.18
Cilantro	sun4u, 270MHz	64M	Ultra5	130.58.68.21
Oregano	sun4u, 270MHz	64M	Ultra5	130.58.68.24
Basil	sun4u, 270MHz	64M	Ultra5	130.58.68.20
Coriander	sun4u, 270MHz	64M	Ultra5	130.58.68.12
Nutmeg	sun4u, 270MHz	64M	Ultra5	130.58.68.14
Mint	sun4u, 167MHz	256M	Ultra1	130.58.68.19
Salt	sun4u, 167MHz	128M	Ultra1	130.58.68.45
Pepper	sun4u, 167MHz	128M	Ultra1	130.58.68.44

Appendix B

Final Strategy (Population Size 75, 131 Generations)

70% Crossover, 4% Little Mutation, 1% Big Mutation

```
 (+ [7,1] (+ [1,4] [4,4]))
(- [8,1] (+ [7,2] [2,3]))
(N [6,8] (- [5,5] [3,2]))
(+ [8,1] (+ [6,4] [7,7]))
(N [2,1] (- [4,7] [6,8]))
(N [1,3] (- [4,7] [6,8]))
(N [5,8] (+ [8,4] [7,2]))
(N [3,3] (+ [5,7] [3,7]))
(N [6,3] (- [2,2] [5,4]))
(- [5,5] (- [6,6] [2,8]))
(+ [3,2] (- [2,1] [6,5]))
(- [4,5] (- [3,8] [4,2]))
(+ [4,3] (+ [7,2] [3,6]))
(N [8,2] (+ [4,7] [5,2]))
(+ [7,8] (+ [6,4] [5,2]))
(- [6,5] (+ [2,6] [1,8]))
(+ [6,5] (+ [2,4] [2,3]))
(N [3,5] (- [3,1] [2,1]))
(+ [2,2] (+ [5,8] [3,1]))
(- [6,2] (+ [7,2] [1,3]))
(+ [2,2] (- [6,1] [3,4]))
(+ [5,3] (- [8,8] [8,3]))
(N [2,2] (+ [5,7] [1,7]))
(N [2,2] (+ [1,6] [6,2]))
(- [5,5] (- [1,5] [5,8]))
(+ [8,2] (+ [3,4] [7,4]))
(+ [2,2] (+ [1,7] [6,2]))
(- [3,2] (- [1,5] [7,3]))
(N [6,2] (+ [4,5] [5,8]))
(N [1,7] (+ [2,3] [6,7]))
(N [7,5] (+ [4,5] [3,4]))
(+ [6,7] (+ [4,7] [4,3]))
(- [6,5] (- [1,8] [5,8]))
```

References

[AP98] P. Adamidis and V. Petridis. On the parallelization of artificial neural networks and genetic algorithms. *International Journal of Computer Mathematics*, 67(1-2), 1998.

[CP97] Erick Cantu-Paz. Designing efficient master-slave parallel genetic algorithms. Technical report, University of Illinois at Urbana-Champaign, 1997.

[Ge+94] Al Geist, Janusz Kowalik (ed.), et al. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994. HTML version at http://www.netlib.org/pvm3/book/pvm-book.html.

[Mit96] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, 1996.

[Nil98] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis.* Morgan Kauffman, San Francisco, 1998.