Using the Island Model Genetic Algorithm to Optimise a Draughts Board Evaluation Function

by

Brian M. Foley

A Thesis submitted to
the University of Dublin
for the degree of

Master in Science

Department of Mathematics
University of Dublin
Trinity College

29 August, 2003

Declaration

Signature of Author ....

Brian M. Foley

29 August, 2003

Acknowledgements

I would like to thank my supervisor Dermot Frost, for giving me the freedom to work independently but still being available to give me advice and ideas when I needed help. I would also like to thank Richard Morrin and Sean Healy for sharing many tips and tricks, without which my time in HPC would have been a lot harder.

Abstract

Draughts is a two-person, deterministic, zero-sum, perfect information game. This report shows how an Artificial Intelligence program was constructed using a look-ahead tree with minimax search and alpha beta pruning. Furthermore, we show how the board evaluation function for the look-ahead tree can be optimised by using a distributed genetic algorithm. Significant time was spent optimising the look-ahead procedure to enable the genetic algorithm to run in a relatively short period of time. A serial version of the code was also constructed to allow interactive play against the computer, where the computer could be fed the results of the genetic algorithm's optimisation.

Contents

List of Figures

Chapter 1

Introduction

For the purposes of this project, draughts will be defined as the conventional 8×8 game of draughts as set out in the Official Rules of Draughts[5]. The playing board is shown in Figure 1.1. If you are familiar with draughts, the only rule of note that is different to "common" draughts is that jumps are forced (*i.e.* there is no "huffing").

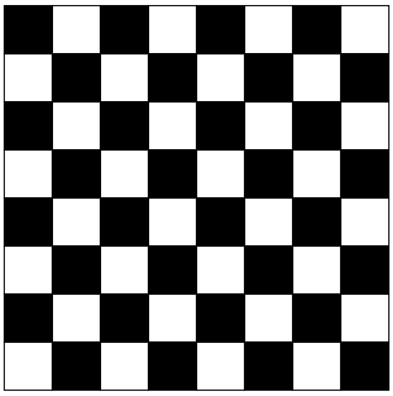


Figure 1.1: Draughts is played on an 8×8 grid.

Draughts is a two-person, deterministic, zero-sum, perfect information game. These terms are defined as:

Two-person: there are two opposing players playing the game.

Deterministic: there is no inherent chance associated with the outcome of the game, *i.e.* the outcome of each move, or transition from one state in the game to another, is known beforehand.

Zero-sum: what is good for one person is bad for the other person.

Perfect information: all information is available to both players at all times.

Games of this type are best solved using a look-ahead search tree to search for the best possible move. The look-ahead uses a board evaluation function to evaluate the relative benefit of each board position in the tree. The board evaluation function needs to be tuned to provide the best possible performance. In this case the board evaluation function was optimised by a genetic algorithm.

We will follow the lead of Samuel[1] and Schaeffer[2] in the design of the draughts program, and will reference the work of Chisholm[4] in the optimisation of the evaluation function. The main emphasis of this project is to develop the Artificial Intelligence aspects of the program, hence we avoid using opening or closing books, even though Chinook's opening books are publicly available[8]. Opening books, closing books, anti-books *etc.* are all ways of improving the performance of the draughts program by pre-calculating numerous board positions and storing them in a database. This was thought to detract from the main aim of improving the evaluation function as the books are conventionally used to replace the evaluation function in the opening and closing stages of play.

Chapter 2

Theory

2.1 Search Trees

The basic concept of search trees is to build a tree of all possible configurations that could result from the current board configuration. You can then assign a weight to each of those possible configurations and choose a route that will lead you to the most beneficial configuration.

A simple example is to introduce the game of tic-tac-toe (also called naughts and crosses). A partial search tree is shown in Figure 2.1. The number of possible board configurations is so small that you could search the entire tree. There is an upper limit of 9! (362880) for tic-tac-toe, but in reality that could be greatly reduced by taking symmetry and finishing positions into account. This search procedure is simply scaled-up to accommodate the greater complexity of draughts. The average branching factor in draughts is about 5, as shown in Figure 4.4, while searches routinely construct trees with about 100,000 terminating nodes, as shown in Figure 4.5.

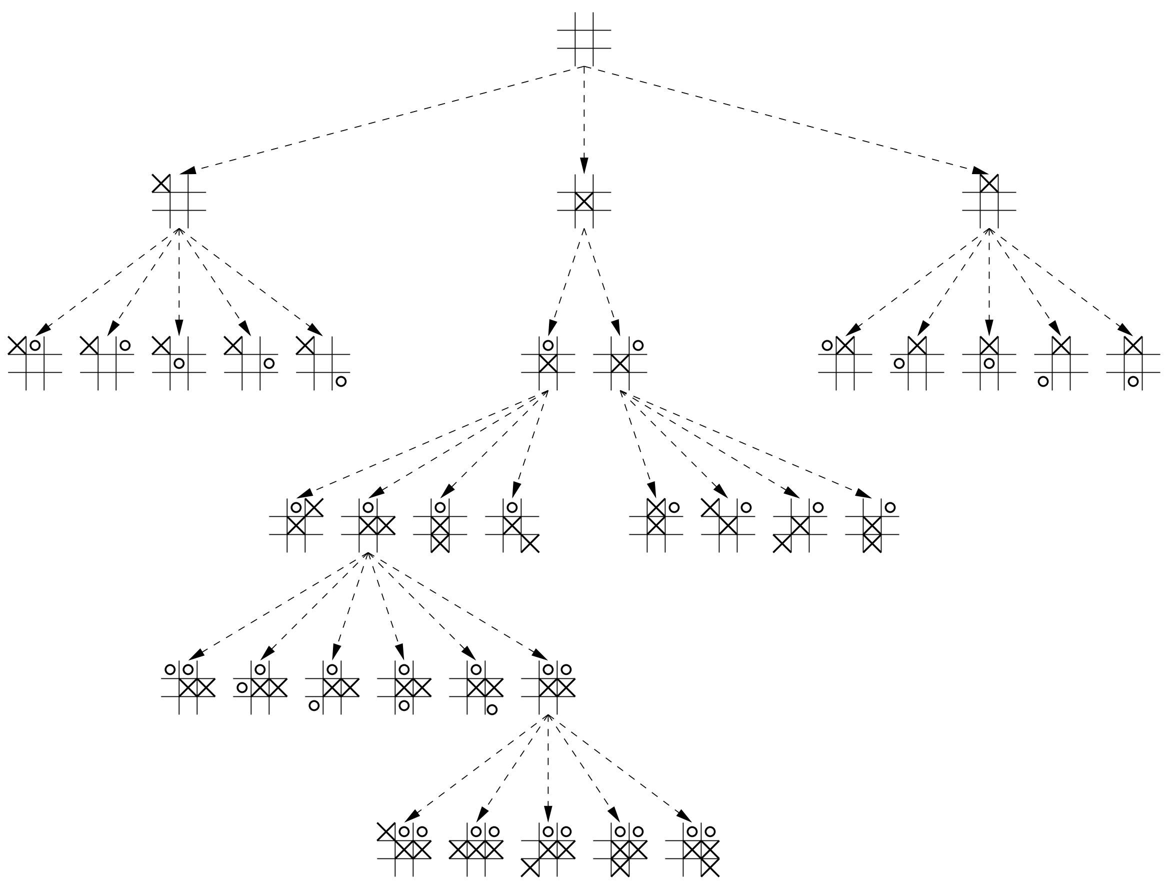


Figure 2.1: An example search tree for the game of tic-tac-toe. The first three levels are shown completely, while the next three levels are only partially shown to save space. Symmetrically identical solutions within the same branch are ignored.

2.2 Minimax Search Strategy

The minimax search strategy is used when the evaluation function is symmetric. For example, a good position for player A may be assigned a score of +100, while a good position for player B may be assigned a score of −100. For this project the evaluation function was carefully constructed to be symmetric, although it should be noted that the evaluation function does not have to be symmetric about zero. Therefore player A would always try to force the game down a path that has a positive score, whereas player B would try to force the game down a path that had a negative score.

In a two player game, player A and player B will take alternative turns, each trying to maximise and then minimise the path that the game takes. This is the origin of

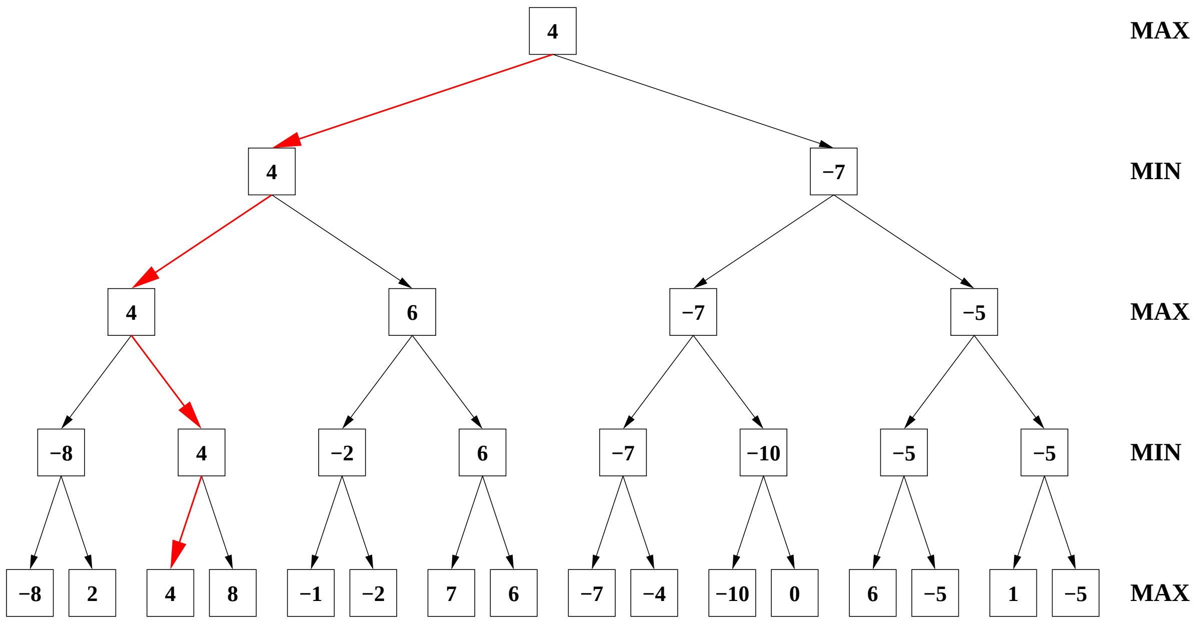


Figure 2.2: A simple minimax scheme for an imaginary game. Possible scores range between −10 and 10. The path that will be followed is shown in bold.

the term **Minimax**. A brief example is shown in Figure 2.2. Note that only the end-nodes of the tree are evaluated with the evaluation function, and then those numbers are propagated up the tree, according to whether it is on a Max node or a Min node.

2.3 Alpha Beta Pruning

Figure 2.3 shows a modified version of Figure 2.2. Alpha Beta pruning can be summarised by saying:

> If you know a particular route is bad, do not waste time finding out how utterly terrible it could be.

Figure 2.3 assumes that you are always evaluating from left to right. Note that once you know that a particular branch is useless for your purposes, you may discard the rest of the branch.

In Figure 2.3, node 12 has been evaluated to -10, this means that without evaluating node 13 we know that node 7 is at least ≤ -10 (*i.e.* the minimum of -10 and some unknown quantity is ≤ -10). Thus to maximise node 4, we have a choice between -7

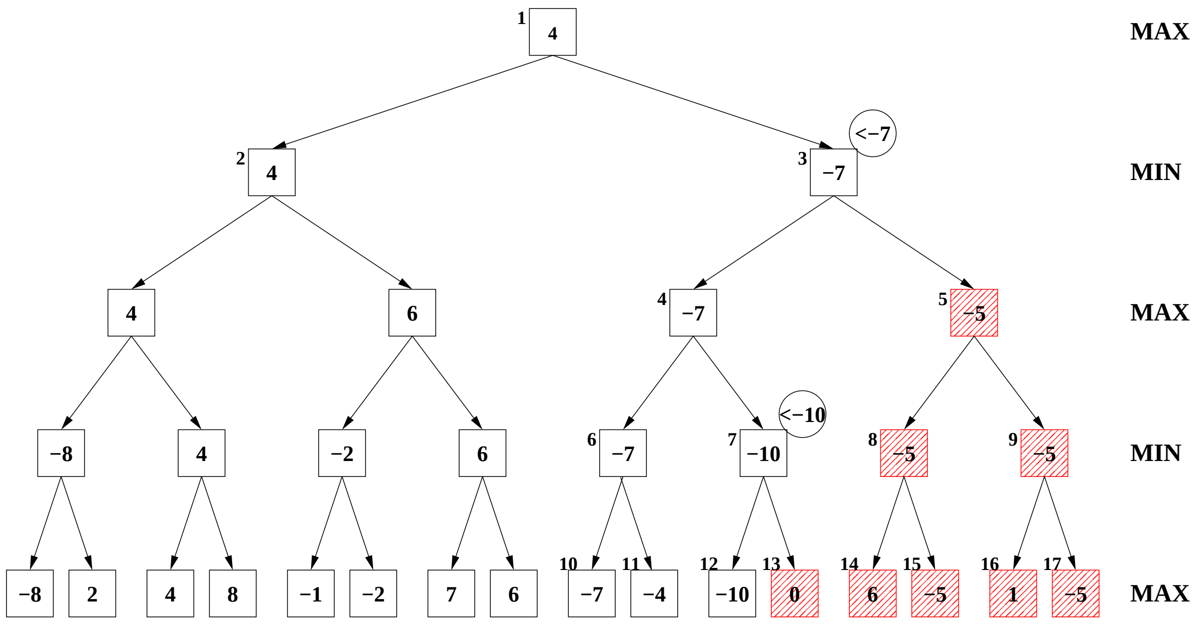


Figure 2.3: An example of alpha beta pruning of a search tree. Note that 5 out of 16 end points do not have to be evaluated, representing a saving of about 31% in calculation time.

and ≤ -10, which is always going to evaluate to -7 no matter what node 13 happens to be, so there is no need to evaluate node 13 at all.

Similarly, it is possible to eliminate all nodes below node 5. Node 3 is a minimising node, therefore it is ≤ -7 because one of its children is -7; this means that the maximising node 1 has a choice between 4 and ≤ -7. The answer to this is always going to be 4, no matter what the nodes under node 5 evaluate to.

2.4 Evaluation Function

The evaluation function for a terminal node is a linear combination of a series of tests. It takes the form:

$$f = a_1x_1 + a_2x_2 + a_3x_3 + \ldots + a_nx_n \tag{2.1}$$

In this case x_n are constructed from general knowledge about the game of draughts (as described by Samuel[1] and Schaeffer[2]) and a_n are what we need to tune to get a "good" evaluation function.

The individual x_n are described here:

Piece Advantage (PA): measures the numerical advantage of your pieces over your opponents. Kings and Men are weighted with a factor of 3 and 2 respectively.

Advancement (ADV): men are given greater weightings as they advance towards the king row.

Balance (BAL): board configurations that have a good left-right balance of the pieces are favoured.

Back Row (BR): keeping men on the back row is rewarded, so that the opponent cannot promote any men to kings.

Man Centrality (MC): having men in the center of the board is good. The center of the board is defined as squares $10, 11, 14, 15, 18, 19, 22, 23$ in Figure 3.1.

King Centrality (KC): having kings in the center of the board is good.

Total Mobility (TM): this is a measure of the total number of moves that are possible from the current position.

King Trap (KT): if a king is trapped in a corner, that is bad.

Free King (FK): if a king has four adjacent empty squares, that is good.

There are actually 36 terms in Equation 2.1. The nine terms enumerated above are grouped together and apply to four distinct stages of play. There need to be different stages of play, due to the changing strategies that are necessary as the board evolves. For example, the strategy one would adopt when there are two black kings and one white king on the board is drastically different from the strategy one would adopt in the initial moves of a game. These stages of play are determined by the number of pieces remaining on the board and are described by the following rules:

Stage	Number of pieces remaining
1	20 - 24
2	14 - 19
3	10 - 13
4	0 - 9

2.5 Iterative Deepening

Often, there is a time limit per-move in draughts competitions, thus a method for limiting the time that a calculation takes is essential. Iterative deepening is a method whereby the tree is evaluated to depth 2, then depth 3, then depth 4 *etc.* Due to results presented in Figure 4.5 we neglect the time to calculate to depth **n**-1 when we are in the process of calculating depth **n**. This simplifies the coding of the Iterative Deepening procedure; we simply construct successively larger and larger trees, noting the result of each tree. When the time limit is reached, the most recent result is used.

2.6 Search Extension

Search extensions (also called "The Method of Hot Pursuit") are used to minimise the horizon effect. A horizon effect can be caused when a large change in the board configuration occurs one move after the depth to which we have searched. A move where one piece jumps an opponent will result in a large change in the board score. To avoid not seeing such moves when we search to a fixed depth, **n**, we will only terminate the search when we have reached a depth of **n** *and* there are no possible jumps on the board. It is quite common for search extensions to extend a particular branch of the tree by two or more levels as the game progresses in a "tit-for-tat" manner. The result of this procedure should be a tree whose nodes are all relatively stable.

2.7 Genetic Algorithm

The aim of this project was to try to evolve a set of weights for the polynomial in Equation 2.1 with the weights themselves forming the genetic material processed by the genetic algorithm. The fitness function used in this genetic algorithm was the number of wins achieved by each player in a round-robin tournament. The basic genetic algorithm is shown in the following pseudo-code:

- Randomly Select Initial Populations
- For Each Generation:
  - Conduct Round-Robin Tournament
  - Set fitness of individual to number of games won
  - Sort by fitness
  - Perform crossover
  - Perform mutation
  - Migrate Individuals (if applicable)
  - Report statistics
- `MPI_Gather()` the fittest individuals to one node
- Final Round-Robin tournament to decide the "fittest of the fittest"

All random events were controlled by functions from the `GSL` libraries. In particular, the selection of individuals for crossover was controlled by the "Roulette Wheel" algorithm. This algorithm means that it is likely that a relatively fit individual will be selected to form offspring for the next generation, but there is a small probability that an unfit individual will be selected instead. This method is used to prevent convergence to local minima and to give "unusual" genetic material a chance (however small) to propagate. The algorithm is as follows:

- assign individuals intervals on the interval $[0, 1)$ whose length is proportional to their fitness
- generate a random number in $[0, 1)$
- choose the individual whose interval contains that random number

2.8 Island Model

The island model genetic algorithm is different from the standard parallel genetic algorithm. In parallel genetic algorithms, there is a single large population spread across many processors, while in the island model each processor is given a subpopulation of individuals. The processors evolve their populations using a serial genetic algorithm. Periodically a processor may migrate a number of its individuals to another population. The amount of communication involved in the island model is very small. One of its main advantages for draughts is that it helps to avoid premature convergence to a particular local maximum. Each processor begins to equilibrate to some stable solution between migration events and is then infused with new genetic material, which will either move the population to a new maximum, or have no long-term effect.

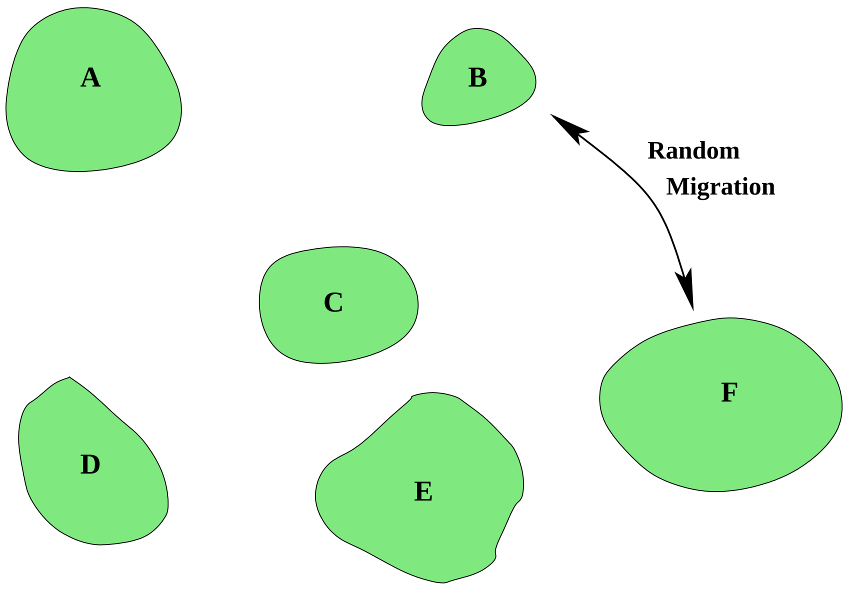


Figure 2.4: This shows the inspiration for the island model genetic algorithm. Shown are several islands with distinct subpopulations. These subpopulations evolve separately, but are subject to random migrations of individuals between islands.

As with the basic concepts of genetic algorithms, the island model is designed to mimic nature. With reference to Figure 2.4, you can compare the island model to a number of isolated islands in an ocean. Each island has its own unique habitat, and the life on that island evolves relative to the particular constraints that the habitat imposes.

This is analogous to the subpopulations evolving independently on different processors. Furthermore, every once in a while, sea or air currents may carry animals or plants between islands in our hypothetical ocean. This is analogous to the migrations that occur in the island model and just as occurs in nature, the newly arrived specimen may either die out quickly, achieve total dominance on the island or live peacefully with the original population.

Chapter 3

Algorithms and Code Design

3.1 Structures

Each individual board position is represented by the following structure:

```
typedef struct s_board{
    unsigned int bm;
    unsigned int bk;
    unsigned int rm;
    unsigned int rk;
} t_board;
```

Each unsigned integer represents the board positions of the black men, black kings, red men and red kings respectively. There are 32 bits in each unsigned integer, which is exactly the number of playable squares on the draughts board (see Figure 3.1). A bit set to zero represents an empty square, while a bit set to one represents a square occupied by that type of piece.

This type of structure does limit the portability of the code somewhat, but since both the computational cluster (`iitac.maths.tcd.ie`) and the development environment (`maths.tcd.ie`) were composed of 32 bit machines which had no problems with this type of code, this restricted portability was accepted as a solution.

Each node on the board tree takes the form:

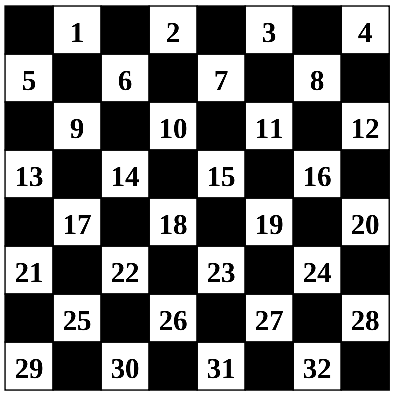


Figure 3.1: Conventional numbering scheme for a draughts board.

```
typedef struct s_node{
    struct s_node *left;
    struct s_node *right;
    struct s_node *next;
    struct s_board board;
} t_node;
```

Thus, the tree structure is built up of nodes as shown in Figure 3.2 for programming convenience, rather than the more common tree structure shown in Figure 2.2. With the depth of tree searches required for an adequate draughts playing program, the number of nodes per tree grows very large. To prevent this from filling the memory of the machine, each branch of the tree is `free()`'ed as soon as it has been evaluated.

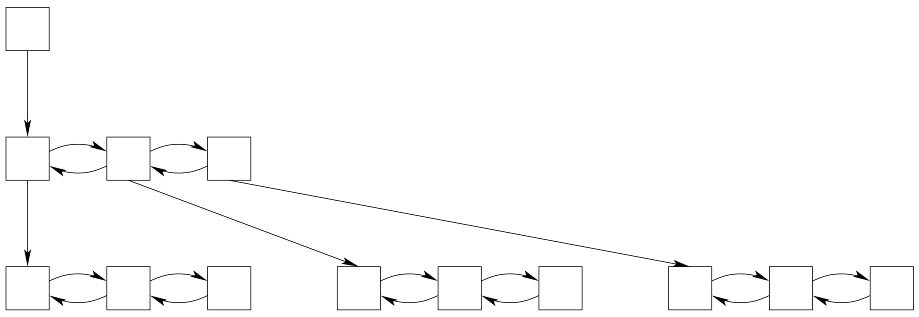


Figure 3.2: Structure used to represent the tree of nodes.

3.2 Special optimisations

3.2.1 Calculating Possible Moves

To increase the speed of the program, most operations are bitwise. This allows the calculation of possible moves to be “in parallel”. There are only three possible ways for pieces to move if they are travelling in the up direction, they can move 3, 4 or 5 places up depending on the square that they started on. See Figure 3.3 for a graphical representation of this.

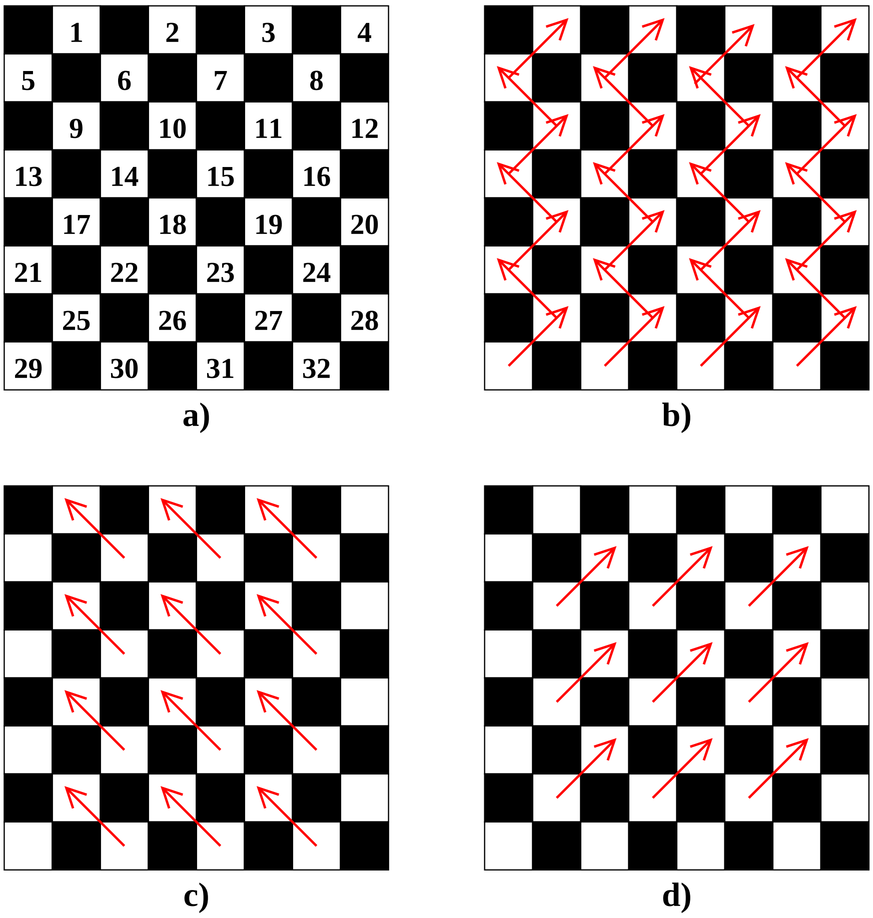


Figure 3.3: Explanation of the “up3”, “up4” and “up5” moves. Figure **a)** is a reminder of the numbering scheme shown in Figure 3.1. Figure **b)** shows how pieces may move up four places on the numbering scheme. Figures **c)** and **d)** show how pieces may move up 5 and 3 places, respectively.

With reference to Figure 3.3, you can see that squares 5 to 32 can all move to a square that is numerically 4 less than itself. We can calculate which of those squares have pieces and can move in that direction with a simple algorithm. First we define a mask called `up4` to mask the squares 5 to 32. We also create `empty` and `upward` to represent the squares that are empty and the squares that contain pieces that are legally allowed to move upwards. A graphical representation of this is shown in Figure 3.4.

```
unsigned int up4 = 0xfffffff0;
empty = ~ (board.rm | board.rk | board.bm | board.bk);
```

Then we apply the `up4` mask to `upward`, shift that integer four places downwards (so that the pieces are now over the destination squares), and `and` that with `empty` to check if the destination squares are empty. Any bit that is non-zero in the resulting integer represents a valid move.

```
valid_moves = empty & ((up4 & upward) >> 4);
```

A similar approach can be taken to calculate possible moves in the down direction, and to calculate jumps. Double and triple jumps *etc.* are accommodated with a recursive call to a similar routine.

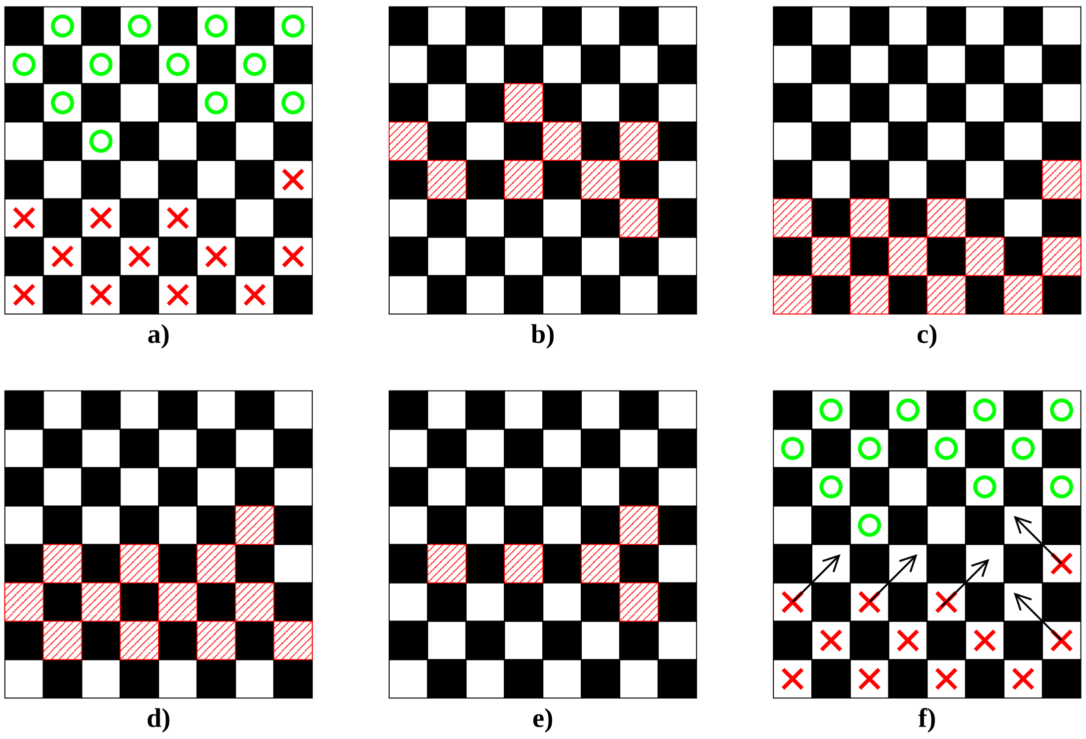


Figure 3.4: This is the procedure used to calculate possible moves for the "up4" case. We begin with the board configuration **a)**, where it is the turn of the red crosses to make a move. Figure **b)** highlights all the empty squares on the board. Figure **c)** shows all pieces that are legally allowed to move upwards. Figure **d)** shows those squares shown in figure **c)** shifted downwards four places according to the numbering scheme in Figure 3.1, this represents the locations to which the pieces in **c)** could move for the "up4" case if each piece was alone on the board. Figure **e)** shows the logical `and` of figures **b)** and **d)**, which represents the actual destinations that are possible (reversing the assumption that each piece is alone on the board from figure **d)**). Finally figure **f)** shows the final product; there are five possible moves available to the red crosses in the "up4" direction. To get the full complement of moves available, the above procedure will be followed for "up3" and "up5" and the results combined.

3.2.2 Profiling code

The performance of the code was a cause for some concern. For example, consider a genetic algorithm with 100 generations, each generation contains 32 x 31 games (if the population size is 32), each game contains about 80 moves, each move involves a tree search of about 100,000 nodes, each node must be evaluated by the `static_evaluator()`. This gives 8×10^{11} calls to `static_evaluator()`, which in turn is based on `count_pieces()`, a function to count the number of bits in unsigned integers. It was found that this counting of bits was the main slowdown, and two algorithms were considered to try to

% time	seconds		calls	ms/call		name
	cumul.	self		self	total	
33.8	0.93	0.93	17087802	0.00	0.00	`count_pieces()`
12.1	1.26	0.33	301814	0.00	0.00	`static_evaluator()`
10.1	1.54	0.28	812566	0.00	0.00	`count_moves()`
9.8	1.80	0.27	259006	0.00	0.00	`calc_jumps()`
8.7	2.04	0.24	1693	0.14	1.61	`minimax_ab()`
6.6	2.22	0.18	432475	0.00	0.00	`count_jumps()`
5.1	2.36	0.14	900754	0.00	0.00	`swap_pieces()`
5.0	2.50	0.14	865656	0.00	0.00	`make_kings()`
4.2	2.61	0.12	100359	0.00	0.00	`calc_moves()`
1.4	2.65	0.04	223332	0.00	0.00	`delete_piece()`
1.3	2.69	0.04	560270	0.00	0.00	`deep_enough()`
1.1	2.72	0.03	259006	0.00	0.00	`calc_all()`
0.7	2.74	0.02	158131	0.00	0.00	`free_restof_branch()`
0.1	2.74	0.00	24	0.12	114.18	`draughts()`
0.0	2.74	0.00	1714	0.00	1.60	`computer_move()`
0.0	2.74	0.00	1714	0.00	0.00	`count_all()`
0.0	2.74	0.00	1693	0.00	0.00	`second()`
0.0	2.74	0.00	1	0.00	0.00	`ParseCommand()`
0.0	2.74	0.00	1	0.00	2740.23	`genetic_algorithm()`

Table 3.1: This is a profile generated by the command `gprof` for a single game of draughts. Note that the majority of execution time is spent in the function `count_pieces()`. Two methods of optimising this function are shown in Figure 3.5 and Figure 3.6.

increase speed.

The two approaches to bit-counting are shown in Figure 3.5 and Figure 3.6. The first method works by `and`'ing an integer with that integer minus one, this will set the least significant `1` bit to `0`. Therefore, counting the number of bits is a simple matter of looping over this code until the integer is zero, while incrementing a counter. The second method works by splitting the 32 bit integer into four sets of eight bits. There are only 256 different bit configurations for eight bits, the results of which are stored in a table in the code.

```
int count_pieces(unsigned int piece){
    unsigned int count = 0;

    while(piece != 0){
        piece &= piece−1;
        count++;
    }

    return(count);
}
```

Figure 3.5: Bit-counting using loops

```
int count_pieces(unsigned int piece){
    static char table[256] = {
        0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8 };
    unsigned int count = 0;

    count = table[piece & 0xFF];
    piece >>= 8;
    count += table[piece & 0xFF];
    piece >>= 8;
    count += table[piece & 0xFF];
    piece >>= 8;
    count += table[piece & 0xFF];

    return(count);
}
```

Figure 3.6: Bit-counting using a lookup table

3.3 Parallelisation

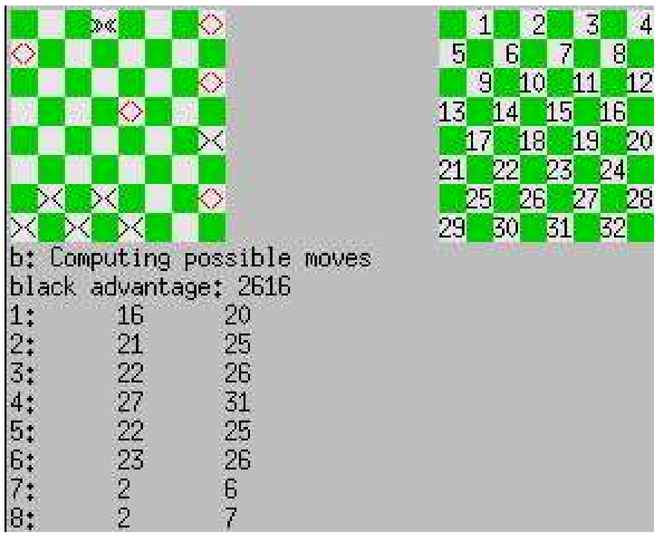
The method of parallelisation used was the Island Model as explained in section 2.8. This method of parallelisation is not communication intensive as migration will only occur after several generations have evolved. It is important to use non-blocking sends, so that the processors can continue somewhat independently. Consider several round-robin tournaments, each on a different island. It is clear that with individuals of different fitness, the time taken to play games could vary quite a lot. For this reason, each island could arrive at the "migration time" at different physical times. With non-blocking sends, the amount of time spent waiting for your neighbours to catch up is minimised.

The migration occurs in a cyclical fashion, with processor 1 sending to processor 2, 2 sending to 3 *etc.* The selection of which individual to migrate was based on the "Roulette Wheel Algorithm" as discussed in section 2.7.

Chapter 4

Testing & Performance

4.1 Draughts Engine



```
b: Computing possible moves
black advantage: 2616
1:       16      20
2:       21      25
3:       22      26
4:       27      31
5:       22      25
6:       23      26
7:       2       6
8:       2       7
```

Figure 4.1: This is the output of the computer *vs.* human mode of the program. The actual board of play is in the top lefthand corner. A board showing the square numbering is shown in the top righthand corner for convenience. At the bottom of the screen all possible moves by the black player are shown, as it is blacks turn to move.

One of the most difficult aspects of this project was testing the validity of the draughts engine. That is, it was imperative that the rules of draughts had been programmed correctly. To test this, the game was played in two-player mode, where the developer could control the moves of both players. Several games were played, at each step the board was carefully analysed to check that the suggested moves (as shown in Figure 4.1) were consistent with the board position and the rules of draughts. Luckily the rules of draughts are quite straight-forward, so once a logical method was adopted and some coding issues were overcome, the draughts engine could successfully operate.

4.2 Efficiency

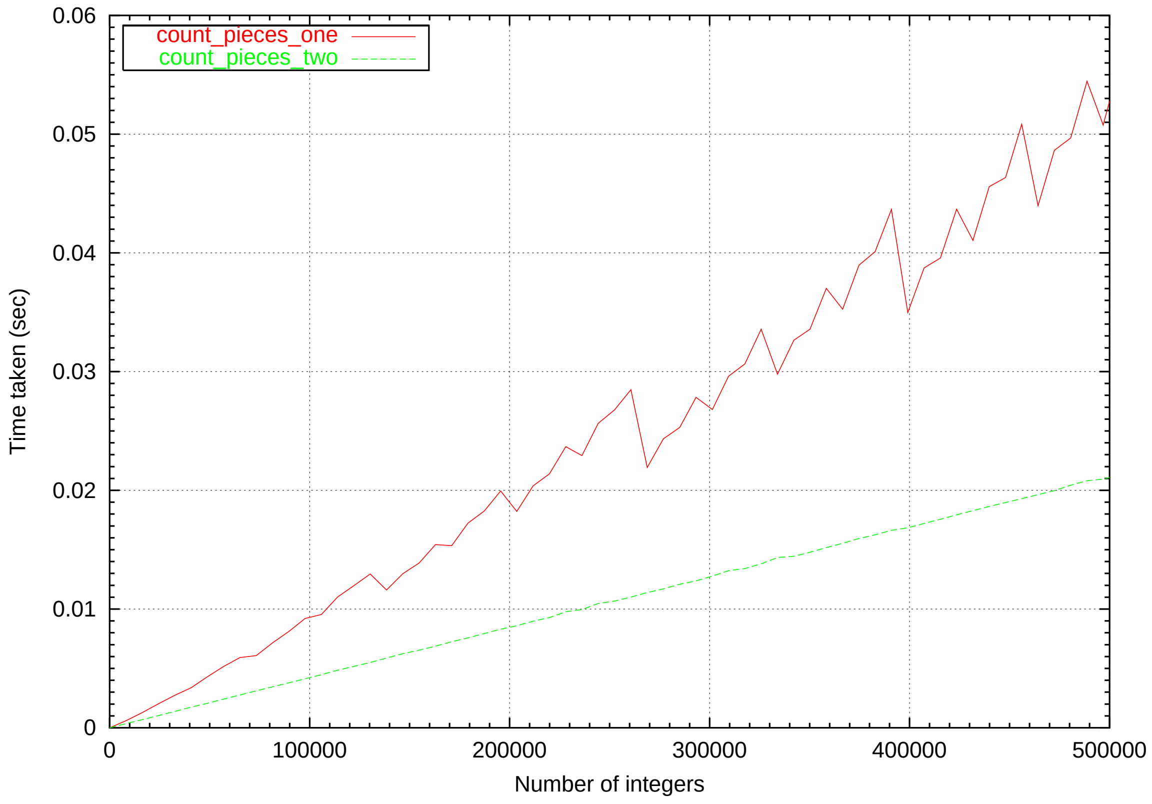


Figure 4.2: Benchmark of the bit-counting algorithms. Note that the number of operations in the first method is dependent on the number of bits in the number, thus there is some variation in the timing. The second method is completely linear, however, with the number of operations staying the same no matter what number is tested.

The efficency of the code was improved in two ways. The first way was to improve the most commonly used function, `count_pieces()`. The theory behind this function is discussed in subsection 3.2.2 and the results are shown in Figure 4.2. Note that the time required to count the bits in a series of numbers is reduced almost in half by adopting the second, more efficient algorithm.

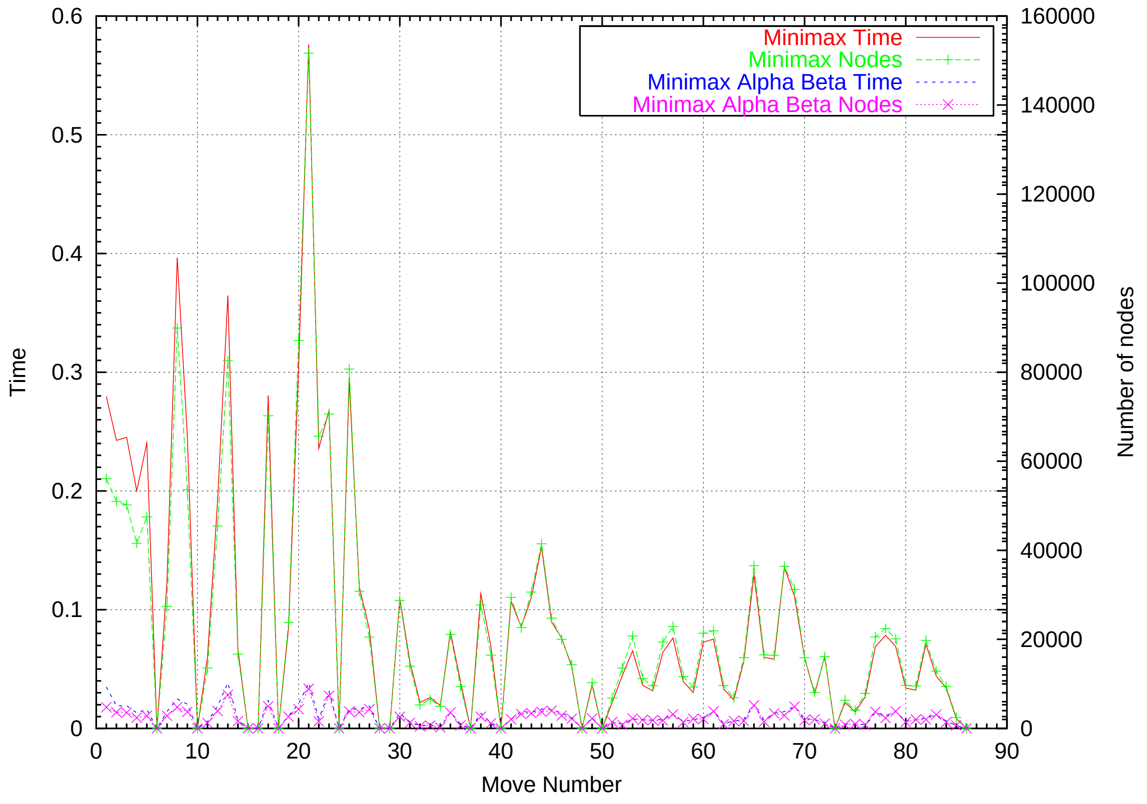


Figure 4.3: The increase in efficiency of using alpha beta pruning is shown here. Note that time taken and nodes counted are always proportional (as expected). Alpha beta pruning increases the efficiency of the search algorithm by about a factor of ten. This particular graph shows a depth limit of 6. Note that the tree is reduced in size as the game progresses, this reflects the reduction in the branching factor shown in Figure 4.4.

While increasing the efficency of the `count_pieces()` code was a valuable exercise, it was even more valuable to reduce the number of times that `count_pieces()` needed to be called. This was accomplished using the method of "Alpha Beta Pruning" as discussed in section 2.3. The results of pruning are quite dramatic and are shown in Figure 4.3. The time taken to search a tree to a given depth using pruning is an order of magnitude less than the time taken without pruning.

4.3 Branching Factor

The branching factor is an interesting statistic to observe during the game of play. As pieces are removed from the board, the branching factor will initially increase because

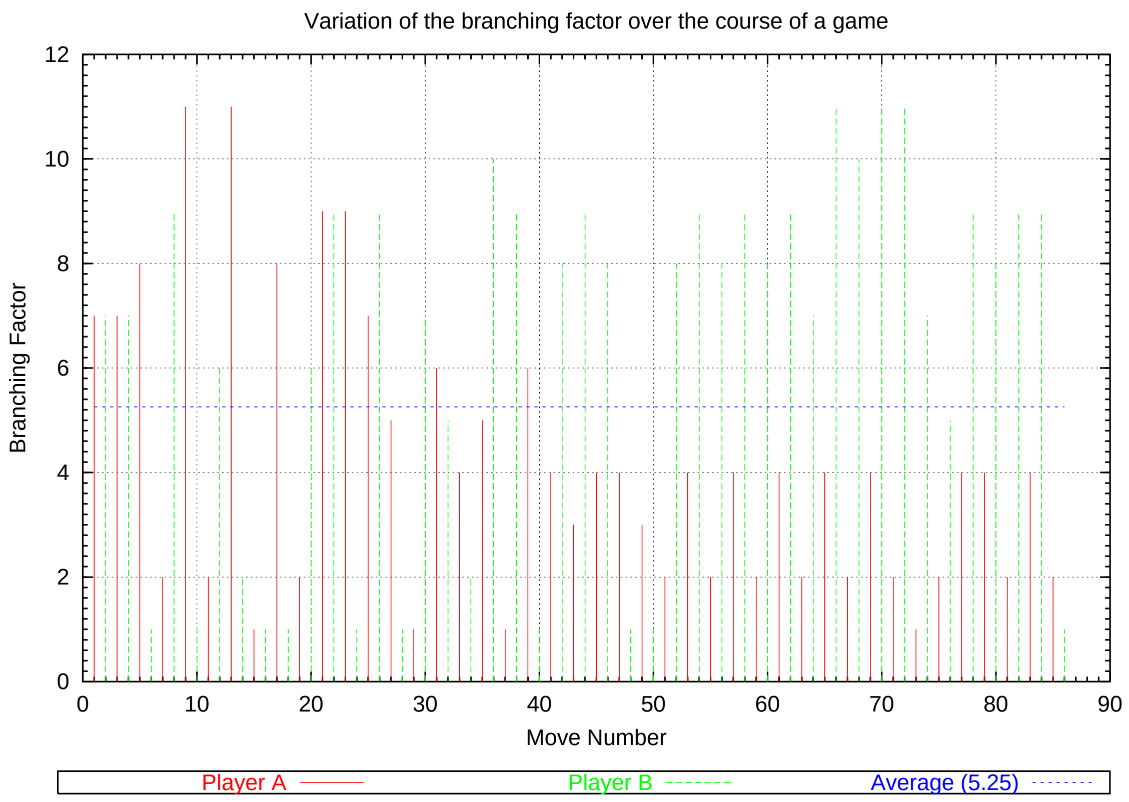


Figure 4.4: The branching factor varies greatly during the course of the game, it can be as low as one when a particular move is forced, or sometimes more than 10 when the board is very sparse, or when there is a lot of kings on the board. Note that the definition of a win in draughts is when the other person cannot move, *i.e.* they have a branching factor of 0. Also note the dramatic reduction of player A's branching factor towards the end of the game, this occurs both due to his loss of pieces and loss of control of the playing board.

there are more squares for the remaining pieces to move to, however as more pieces are removed, the small number of pieces remaining on the board will mean that the branching factor reduces somewhat. It is interesting to compare the branching factors of the winning and losing side of a match. In Figure 4.4 note that player B has a branching factor of 1 quite a few times in the early game. These represent occasions when player B had an opportunity to jump player A, and since jumps are forced the branching factor was 1. The overall trend of player A's branching factor is downwards, while the branching factor for player B remains steady at about 8. The downward trend for player A is representative of their loss of options and loss of control as time progresses.

4.4 Effect of Search Depth

The effect of search depth on the speed of play is highly significant. Figure 4.5 shows the number of nodes that are examined for trees of depth $2, 4, 6, 8$ and 10. The horizontal lines represent the average values of the set of points for each depth. Any increase in accuracy is paid for by an exponential increase in computation time. Thus the search depth is usually set to 8, a value which represents a large look-ahead, but is not prohibitively slow.

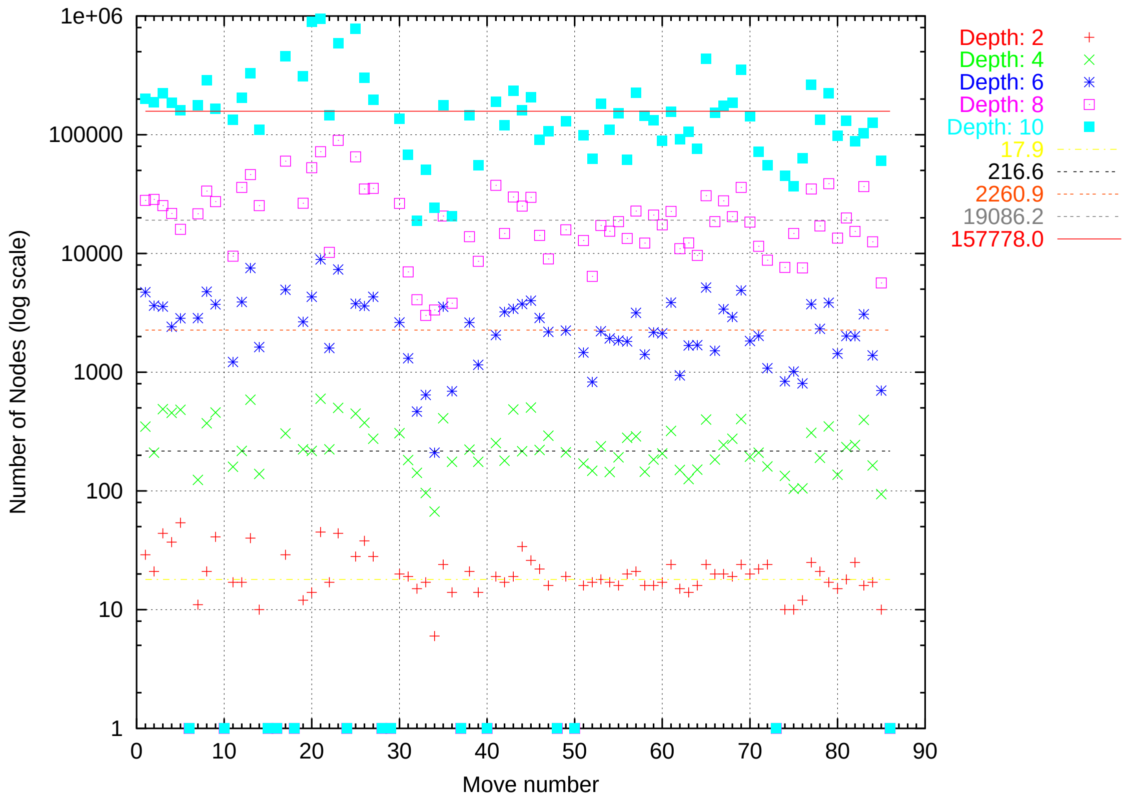


Figure 4.5: This is a comparison of the number of nodes in different depths of lookup trees. Figures were recorded during the same game of draughts. Note that the increase in the number of nodes is exponential, with an approximate fit of $33.7e^{2.1x}$. The horizontal lines represent averages for each set of points.

Chapter 5

Results

5.1 Evaluation Function Parameters

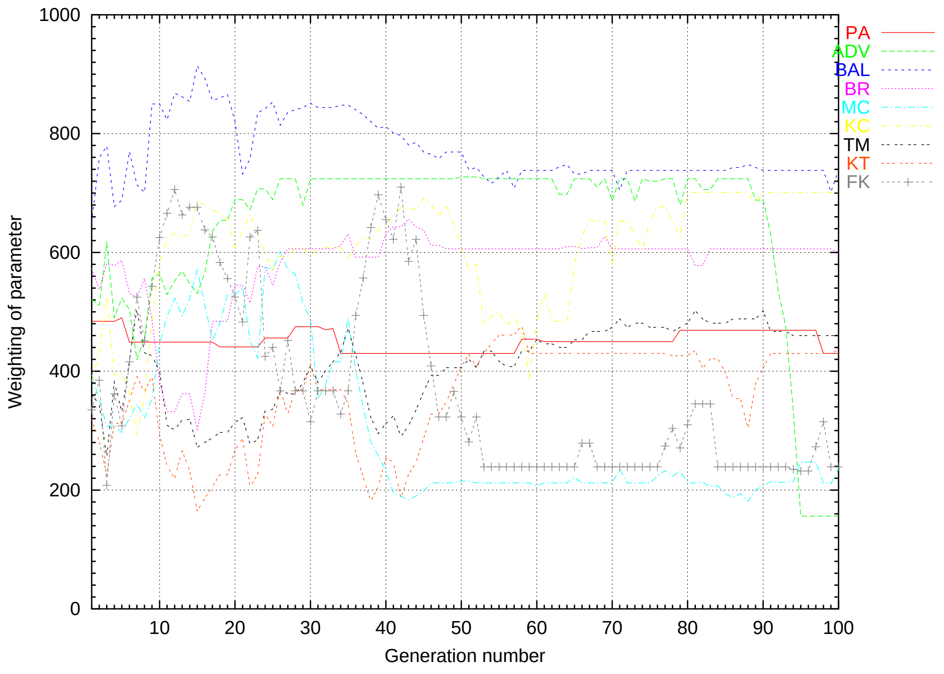


Figure 5.1: Evaluation Function Coefficients for Stage 1.

Sample output from a single-population run of the program is shown in Figures 5.1 - 5.4, which show the different parameters from Equation 2.1 split into stages, as described previously. The first thing to note about these plots is that the figures reported

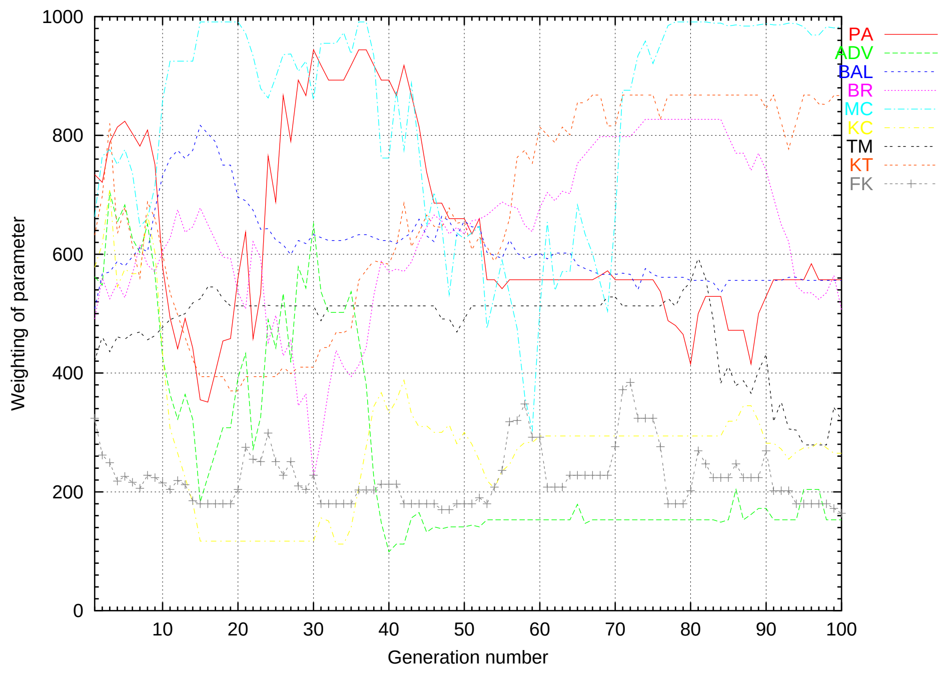


Figure 5.2: Note the large transition at about generation 50 during stage two. This indicates a transition from one maximum to another. The general confusion of this graph indicates that stage two may benefit from being made smaller.

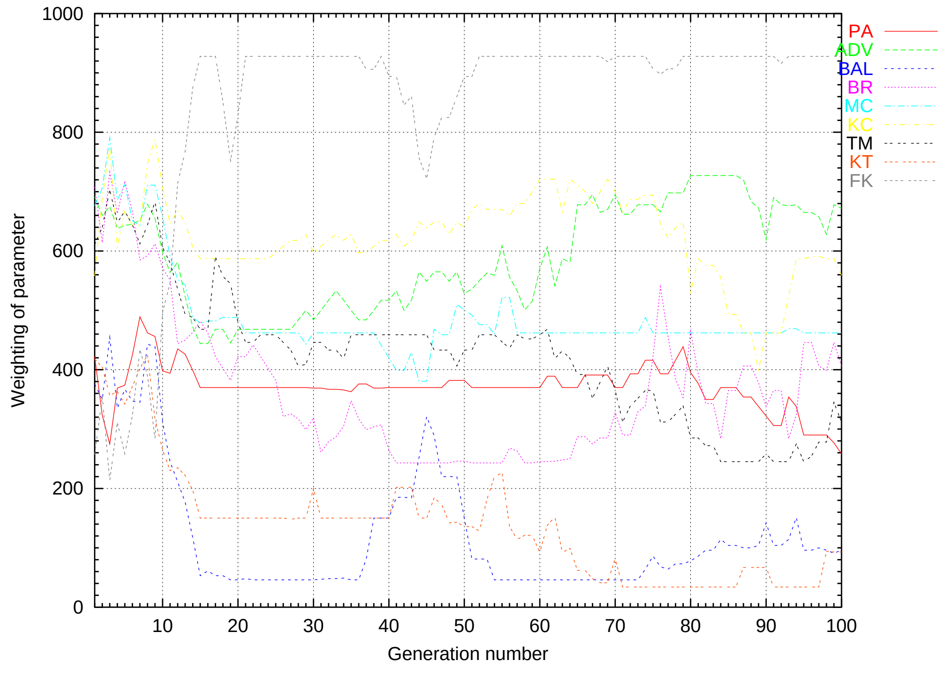


Figure 5.3: Evaluation Function Coefficients for Stage 3.

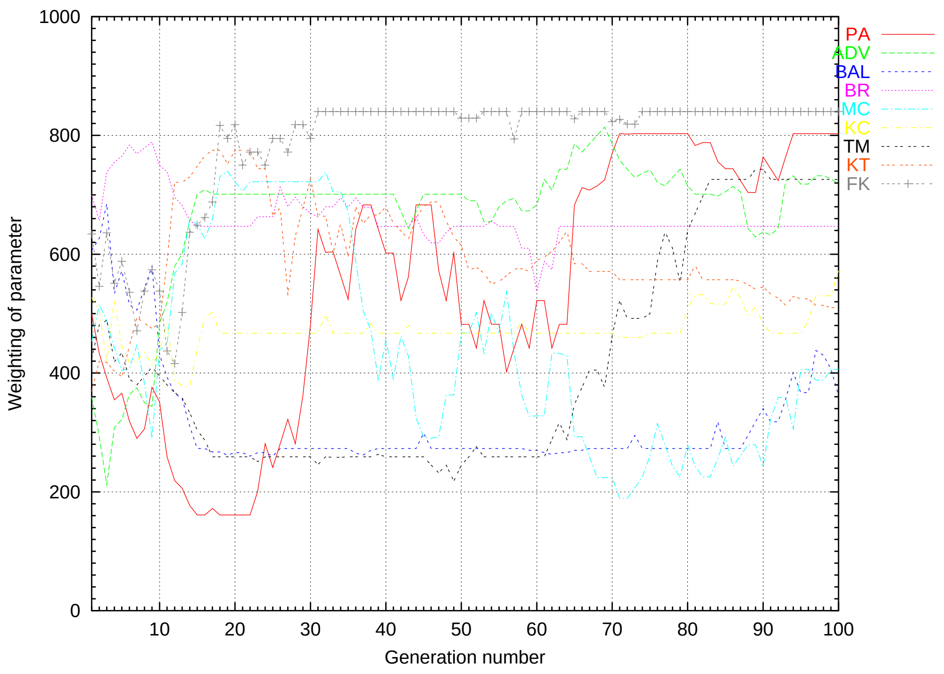


Figure 5.4: Note that in the latter stages of the game, Free Kings become very important. This is in keeping with common experience during the end-games of draughts, where controlling the board with kings is a good strategy. Kings are so much more important because they can guard four squares as opposed to men, which can only guard two squares at most.

for each parameter are actually the average of that parameter over all members of the population. These parameters can change quite quickly, if an element undergoes a beneficial mutation, or have a slight bump if an element undergoes a non-beneficial mutation.

Some of the parameters stabilise to what would be considered "intuitive" levels, while others are more surprising. Note that Piece Advantage (PA) is not dominant in all but stage 4. This is somewhat surprising as one would have thought that piece advantage would be important in all stages of play. It may suggest, however that in the early stages of play strategy is much more important than maintaining a strict numerical advantage. In stages 3 and 4 things like King Centrality (KC) are very dominant, whereas in stages 1 and 2 Balance (BAL) and Piece Advancement (ADV) are much more important.

It is also instructive to see what is not important in each of the stages. Stages 3 and 4 put a very low emphasis on Balance (BAL), because during the latter stages of the game the board becomes more symmetrical as more and more pieces are converted into kings. Similarly the importance of Free Kings (FK) in stage 1 is ranked very lowly, simply because it is very difficult to have a piece promoted to king when there are still 20 or more pieces remaining on the board.

5.2 Genotypic Similarity

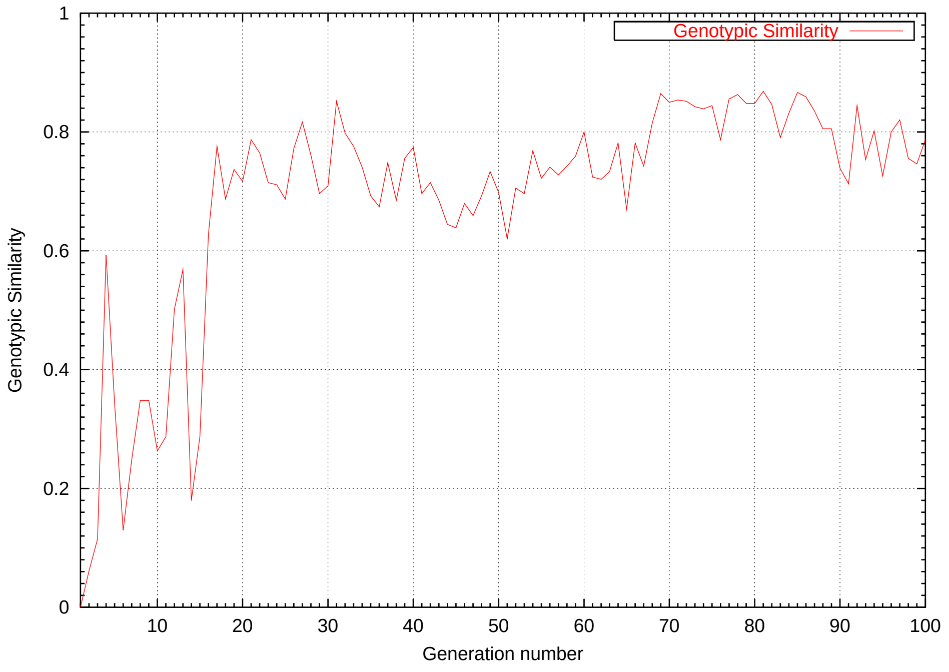


Figure 5.5: Genotypic Similarity is a measure of convergence of the individuals in a population. It is the fraction of the populations elements that are the same as the elements of the fittest individual. We can see here that the Genotypic Similarity increases rapidly during the first 20 generations, and increases gradually after that. The genotypic similarity will never approach 1 due to the effects of mutation on the population.

Figure 5.6 shows the interaction between the genetic material of three islands. Note the prominent drop to zero for nodes two and three just after 45 generations. This type of feature occurs when the newly arrived genetic material is dramatically better that the material that was on the node beforehand. This results in the newly arrived individual instantly being ranked first in the round-robin tournament against the rest of the subpopulation. The node can recover from this spike so quickly because when an individual is so dominant they get predominantly chosen for the crossover operations and thus their genetic material is quickly distributed among the sub-population, resulting in a high genotypic similarity.

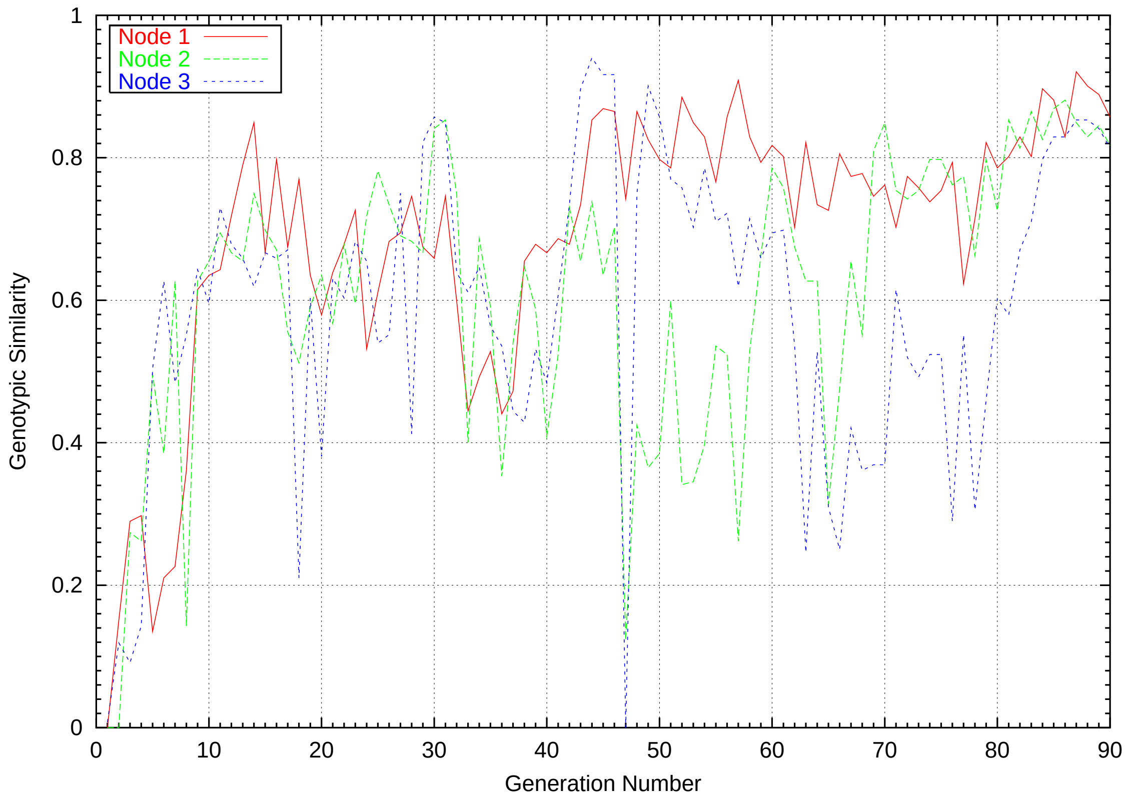


Figure 5.6: This figure shows the interaction between processors for a three processor system. A migration event was scheduled to take place every fifteen generations. Note that the system eventually stabilises after a certain number of generations.

What appears to be happening in Figure 5.6 is that node 1 is the dominant node with a better population. Shortly after several of the migrations, you can see a drop in the genotypic similarity of nodes 2 and 3, which represents a change in their genetic material. If we examine the raw data of the genetic material we see that there is some evidence that the nodes are coming to equilibrium. For example, the total mobilities (TM) for stage one were: 69, 84 and 93 for the three nodes, which are similar enough for us to think that they are related. Likewise the balance figures (BAL) for stage four were: 409, 428 and 419 respectively. Again it is clear that the genetic material of the three nodes is converging, although at a very slow rate.

5.3 Playability of Game

The playability of the final game, once we had used the genetic algorithm to determine a good set of parameters, was the final test of the program. The resulting program is about equal in ability to the authors' draughts playing ability. The program was particularly good at setting up its opponent for multiple jumps. It would usually sacrifice a piece early in a set of moves, to force its opponent to make a certain set of moves that would result in a double- or triple-jump for the computer. The number of games won by the author and by the computer are about equal, although as noted below, if the computer is forced into an end-game situation it can usually be beaten.

One area in which the program struggled was in the end-game. This is a known factor in the world of computer checkers, and one of the reasons why end-game databases are so common. The number of moves required to achieve a win when there are few pieces on the board is very large. In some classic cases[1] such as the so-called "Second Position", where there are 3 pieces against 3 pieces, the winning side must play 40 accurate moves to reach an easily winning position. Unfortunately this level of look-ahead is beyond the power of the program and its playing ability suffered as a result.

Chapter 6

Conclusion

A draughts engine was constructed and was used as the fitness function for a genetic algorithm. Several methods were employed to speed up the operation of the code. An island model genetic algorithm was employed to decrease the likelihood of getting stuck in local maxima and to increase the convergence of the genetic material to a stable state. The final program was able to play draughts against a human opponent to a reasonable level.

Bibliography

[1] AL Samuel: Some Studies in Machine Learning Using the Game of Checkers, Computers and Thought, MIT Press, 1995. 2, 6, 30

[2] Jonathan Schaeffer *et al*: A World Championship Caliber Checkers Program, Artificial Intelligence, Vol 53, pp 273-290, 1992. 2, 6

[3] Jonathan Schaeffer *et al*: Reviving the Game of Checkers, 1991.

[4] KG Chisholm and PVG Bradbeer: Machine Learning Using a Genetic Algorithm to Optimise a Draughts Program Board Evaluation Function. 2

[5] `http://www.irishdraughts.org`. 1, A-1

[6] `http://www.triplejump.net`.

[7] `http://www.jimloy.com/checkers/checkers.htm`.

[8] `http://www.cs.ualberta.ca/~chinook/`. 2

Appendix A

Official Rules of Draughts

These rules have been sourced from the North West Draughts Federation [5], which is a federation of draughts clubs and individual players across the north west of Ireland.

1. The draughts board is square in shape and is divided into 64 squares of equal size, alternately light and dark in colour (technically called green and buff).

2. The board is placed between the two players such that the bottom left-hand corner square is green.

3. The game is played on the green squares, which for the purpose of reference are assigned numbers from 1 to 32.

4. Each player starts with 12 discs, or “men”, all of equal size. One player has dark coloured men (called red) and the other has light coloured men (called white)

5. At the commencement of play the red men occupy squares 1 to 12 and the white men occupy squares 21 to 32.

6. To start the first game the players decide by the toss of a coin which colour they will play. In subsequent games the players alternate colours.

7. The first move in each game is made by the player with the red men, thereafter the moves are made by each player in turn.

8. There are fundamentally 4 types of move: the ordinary move of a man, the ordinary move of a king, the capturing move of a man, and the capturing move of a king.

9. An ordinary move of a man is its transfer diagonally forward left or right from one square to an immediately neighbouring vacant square.

10. When a man reaches the farthest row forward (the king-row or crown head) it becomes a king, and this completes the turn of play. The man is "crowned" by the opponent, who must place a man of the same colour on top of it before making his own move. (It may be necessary to borrow from another set if no captured man is available for the purpose).

11. An ordinary move of a king (crowned man) is from one square diagonally forward or backward, left or right, to an immediately neighbouring vacant square.

12. A capturing move of a man is its transfer from one square over a diagonally adjacent and forward square occupied by an opponent's piece (man or king) and on to a vacant square immediately beyond it. (A capturing move is called a "jump"). On completion of the jump the capturing piece is removed from the board.

13. A capturing move of a king is similar to that of a man, but may be in a forward or backward direction.

14. If a jump creates an immediate further capturing opportunity, the capturing move of the piece (man or king) is continued until all the moves are completed. The only exception is that if a man reaches the king-row by means of a capturing move it then becomes a king but may not make any further jumps in the same turn. At the end of the capturing sequence, all captured pieces are removed from the board, in the order in which they were jumped.

15. All capturing moves are compulsory, whether offered actively or passively. If there are two or more ways to jump, a player may select any one he/she wishes, not

necessarily that which gains the most pieces. Once started, a multiple jump must be carried through to completion.

16. Either player, on intimating his/her intention to his/her opponent, is entitled to adjust his/her own or his/her opponent's pieces on their squares at ant time during the course of the game.

17. If a player on his/her turn to move touches a piece he/she must play the piece, unless he/she has given an adjustment warning. If the piece is not legally playable, rule 19.2 applies.

18. If any part of a playable piece is played over a corner of a square on which it is stationed, the move must be completed in that direction.

19. A player making a false, improper or illegal move shall be cautioned for the first offence, and the move recalled. He/she shall forfeit the game for any subsequent false, improper or illegal move made in that game. This applies, for example, if a player:

    (a) Omits to capture or to complete a multiple capture (this supersedes the old "huff" rule).

    (b) On his/her turn to play touches an unplayable piece.

    (c) Moves a piece, either in an ordinary move or in a capturing move, on to a wrong square.

    (d) Moves an uncrowned man backwards.

    (e) When capturing, removes an opponent's piece or pieces not in a position to be captured in that move.

    (f) When capturing, inadvertently removes one or more of his/her own pieces.

    (g) Continues a capturing move through the king-row with a man not already crowned.

    (h) Moves a piece when it is not his/her turn to play.

20. If any of the pieces are accidentally displaced by the players or through any cause outside their control, the pieces are replaced without penalty and the game is continued.

21. A player who refuses to adhere to the rules shall immediately forfeit the game.

22. There are only two possible states to define: the win and the draw.

23. The game is won by the player who can make the last move, that is, no move is available to the opponent on his/her turn to play, either because all his/her pieces have been captured or his/her remaining pieces are all blocked.

24. A player also wins if his/her opponent:

    (a) Resigns at any point.

    (b) Forfeits the game by contravening the rules.

25. The game is drawn if at any stage both players agree on such a result. (This usually occurs when neither player can force a win)

26. 40- Move Rule. The game shall be declared drawn if, at any stage of the game, a player can demonstrate to the satisfaction of the referee that both of the following conditions hold:

    (a) Neither player has advanced an uncrowned man towards the king-row during the previous 40 moves.

    (b) No pieces have been removed from the board during the previous 40 moves.

    Note: For the purpose of this rule, a move shall be said to consist of one red move and one white move.