# CS221 Othello Project Report

## Lap Fung the Tortoise

Alvin Cheung
akcheung@stanford.edu

Alwin Chi
achi@stanford.edu

Jimmy Pang
hcpang@stanford.edu

November 28 2001

## 1 Overview

The construction of Lap Fung the Tortoise consists of three major areas, the search algorithm, the evaluation function, and the training algorithm. In addition to standard techniques like alpha-beta pruning, various tricks and extensions are employed. Notable features include the opening book, transposition tables, an adaptive time-management scheme, a parallel training scheme, and a 14-ply end-game search. The result is an Othello program that beats its authors as well as their friends.

## 2 The Search Algorithm

We used standard alpha-beta pruning as the search algorithm, and added two major extensions:

- The Transposition Table – using information from previous searches:

  The transposition table is a hash table that stores board positions encountered in a search together with the corresponding best moves found. During a search, the board configuration encountered is first looked up from the table. If we have seen that position in previous searches, we can retrieve it from the table and use the 'best-move' stored in the table as the next successor of the search. Likewise, when we determine the 'best move' for a particular position, we will store the information into the transposition table for future use.

  We encountered two major issues in the table implementation:

  Firstly, we have a fix-sized hash table, which can get filled up over time. What should we do if the table gets full? The solution is to use a Least-Recently-Used (LRU) entry eviction scheme for the hash table. Since the search continually encounters new positions, positions before the current move are obsolete and can be removed from the table. This is naturally done using LRU. The entry we evict is based on how recently 'visited' the position is - the longer since it has been visited, the more likely it is to be evicted. With this strategy, we don't need to dedicate time to flush the table, since the obsolete positions will be automatically overwritten by newly encountered positions under LRU. Our hash table has 262144 buckets and each bucket contains 4 entries, allowing it to theoretically 'remember' over 2 million positions at any given time. LRU eviction on the 4 entries in every bucket is implemented using a 'second chance' clock algorithm.[1]

Secondly, we need to hash the whole board into the table. This could be an expensive operation if the board were stored as a 10 by 10 array. Our solution to this is to use two bitmaps (one for white pieces, one for red pieces) to store the board, reducing the operation to a 256-bit hash per board position. Also, we used a hash function based on bit-mixing [2] to strike a balance between evenness and speed in hashing.

- The Killer Tables – for move ordering:

  With alpha-beta pruning, we can reduce the branching factor a lot if we expand the best successors first. To achieve that, we used the transposition table (described above) for choosing the best node to expand first. However, since the transposition table is limited in size, we may not get the ordering for every node.

  The handout suggested that we use our evaluation function to order the moves. This is not a good idea for two reasons:

  First, the evaluation function may be expensive to compute. Sorting the moves at every node is even more expensive - the benefit from this ordering cannot compensate the cost.

  Second, the evaluation function may not be accurate in determining the quality of the moves (that's the whole point of doing a search). Using the information from the transposition table (which is the result of a search) is much more accurate than using the evaluation function.

  In addition to the transposition table, we maintained a set of dynamically-updated tables for move ordering. These are known as the Killer Tables, a name given by their original designer.[3]

  The killer table contains 100 entries for each color. The entry $x$ in each table for color $c$ represents responses for $c$ to the move $x$ by the opponent. This entry contains not one move, but an array of all the empty squares. They are heuristically ordered from the best response to the worst by $c$ to the move $x$.

  The killer tables are initialized before each game in a heuristic fashion. For example, the corners are placed first, then the middle squares, and lastly the x-squares.

  Since the optimal move for each position varies as the game proceeds, we implemented the following algorithm for dynamically re-ordering moves in the Killer Tables:

  Whenever a node is chosen to be the best successor in the alpha-beta search, the entry in the Killer Table for that color and the previous move is updated - it is moved to the front of the array (using `memchr` and `memmove` functions in C, this can be done very rapidly). In the long run the Killer Tables maintain a move-ordering that approximates the quality of the moves.

  Also, whenever a player makes a move, that move is removed permanently from the Killer Tables. We can do this because in Othello any square on the board can only be played once in a game. Once it is played, it is never a legal move any more (since it will always be occupied afterwards).

  The Killer Tables significantly improved the search speed, as it drastically reduced the branching factor. Also, removing illegal moves as the game proceeds significantly speeds up the search for the final stages of the game, when most of the squares are played and hence removed from the Killer Tables. With fewer squares to be checked for mobility, the search speed is greatly improved.

[1] Silberschatz p.310
[2] Jenkins, taken from <`http://burtleburtle.net/bob/hash/evahash.html`>
[3] see Kai-Fu Lee's paper on 'BILL'

An experiment was done with the program playing with and without the killer tables and transposition tables. It turned out that the program with both tables searches only around 60% of the nodes of the program without the tables at depth 7. (Although the program with tables ran a little slower due to the overheads of the tables, the overall result was still a significant speedup) This result further illustrates the merits of these extensions.

## 3 The Evaluation Function

Our evaluation function is a weighted sum of the following features:

1. Piece differential
2. Mobility differential
3. Potential mobility differential
4. Corners differential
5. X-Squares differential
6. Wipe-out avoidance [4]

These features are chosen based on a research on the strongest Othello-playing programs at the world-champion level.[5] The features we chose are the most widely used features in those programs.

- Piece differential

  Definition: number of our pieces - number of opponent's pieces.

  This feature enables us to minimize the number of pieces at the early stages of the game (a strategy well-known to othello experts), and also contribute to the wipe-out avoidance procedure described below. This is the only feature considered during the end-game search.

- Mobility differential

  Definition: number of legal moves for our side - number of legal moves for our opponent.
  This feature enables us to maximize our number of moves relative to our opponents (also a well-known Othello strategy).

- Potential mobility differential

  Definition: number of empty squares adjacent to an opponent piece that is not stable - number of empty squares adjacent to our stable pieces.

  This feature measures the number of potential moves one has. It is closely related to the number of frontier discs one has. The greater the potential mobility differential, the more centered are the pieces positioned. Maximizing this feature tends to result in pieces being played in the center region of the board, where they are least likely to become frontier disks or walls (which are bad since frontier disks tend to increase the mobility of the opponent).

[4]This is not really a 'feature' in the classical sense but rather a mechanism for ensuring it will not be the case that all of our discs are captured in the middle of the game.

[5]see "The inner workings of strong Othello programs" at <`http://www.nada.kth.se/~gunnar/howto.html`> for details.

- Corners differential

  Definition: The board is divided into 4 quadrants, with 3 corners in each quadrant. The corners evaluation is the sum of the square of the number of corners occupied in each quadrant. The corners differential simply takes the difference between the corner evaluations of both sides.

  We take the square of the number of corners in the same quadrant having the notion in mind that it is more preferable to occupy all the corners in the same quadrant than occupy corners in different quadrants.

- X-Squares differential

  Definition: The X-Squares are the squares adjacent to an unoccupied corner. (see diagram below). They are considered to be undesirable moves by othello experts, and that's why they are named X-Squares. The X-Squares evaluation, like the corners evaluation, takes the sum of the squares of the number of X-squares occupied in each quadrant. The X-Squares differential is a difference in the X-Squares evaluation of both sides.

  This feature is given a negative weight, since it is generally not very wise to place a disc next to an unoccupied corner, as doing so usually enables the opponent to win the corner next to it.

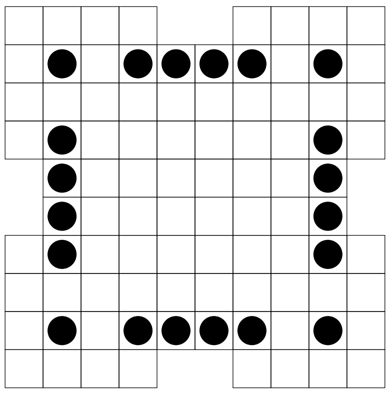


Figure 1: Illustration of x-squares

- Wipe-out avoidance

  In the case where one side loses all its discs. This is essentially a loss, but from the evaluation function, it might appear highly desirable, from the mobility point of view. Therefore, when we evaluate a board position, we must make sure that we will not evaluate along a wiped-out deadend. The way we do this is by returning a large penalty for getting wiped out (1000 $\times$ number of pieces of the winning side in our implementation). This ensures that the program won't seek to reduce its piece count to zero.

## 4 Data Structures for Calculating Mobility and Potential Mobility

Since the given board data structure is very inefficient in determining whether a square is a legal move or not, we devised our own data structure for evaluating mobility and potential mobility efficiently.

## 4.1 The BitBoards

We used two bitmaps [6] to store the board, one for each color. The BitBoards we used have the advantage of being 'hash-friendly' as mentioned above. Now we will see how they enable us to efficiently compute mobility and potential mobility.

## 4.2 The AdjacencyTable

This is an array of 100 BitBoard structs. It stores all the positions that are adjacent to the square to be looked up. To be more concrete, let's consider following example:

For instance, if we want to get all the squares adjacent to (5, 5), we would look at `adjacencyTable[coordToMove(5, 5)]`, which would be a BitBoard struct with the following configuration:

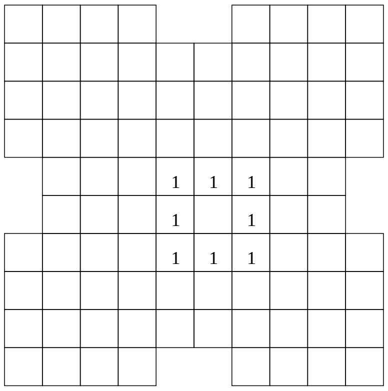


Figure 2: An example of an entry in the adjacency table

All the squares that are adjacent to (5, 5) will be marked with bit 1 on the bitboard. The adjacencyTable is extremely useful when we are trying to determine potential mobility or finding out all possible moves given a board position. The way we do it is as follows:

Suppose we want to find the possible moves for black. We know that all these moves must be squares adjacent to white squares. So we simply take the union of the bitboards returned by looking up the adjacencyTable for all the white squares, minus the current bitboard for black and white. This gives a board with all the "potential moves" by black marked as 1.

This provides a cheap and efficient way to find both the possible moves (which is then a search on the potential moves, instead of every square on the board), and the potential mobility, which is a very important feature of the evaluation function. And we get both for a single iteration through the white pieces!!

Note that since the adjacencyTable is designed for finding potential moves, some squares are not marked even though it is strictly speaking adjacent to the square to be looked up. For instance, if the square to be looked up is the corner square, the corresponding board in the adjacency table is an empty board, since no square can 'potentially' flip a corner.

Also, since the adjacencyTable depends on the board configuration (like where the X squares are), we generate them in run-time before the game starts as part of the game initialization process.

[6] for details see <`http://eclipse.sourceforge.net/othello_code.html`>

### 4.3 The DirectionTable

Since we are storing the board as a bitmap, we lose the adjacency information of the squares. For instance, if we need to find the square to the northwest of square a, we need to do arithmetic calculations. This is further complicated given the illegal X squares of the board. Our solution to this problem is to precompute all the adjacency information of all the squares, and store them in a 2D array:

`directionTable[100][8]`

so we can look up the adjacent squares of square a in each of the 8 directions.

We conducted an experiment comparing the speed of the search using these data structures with the program using the provided data structures. Our test involves running the greedy search with alpha beta pruning at depth 7. Our program turns out to take only one tenth of the time required for the stock program. This illustrates the importance of these data structures. It enables our program to search deeper under given the same time.

## 5 The Opening Book

Most expert Othello playing programs have an opening book that stores pre-computed responses for the opening moves. Lap Fung the Tortoise is no exception. The opening book is made possible by having the transposition table store the recorded moves and the appropriate responses. Thus only a fast hash table lookup is required to find the response to a certain position in the opening book. Here we will discuss the generation of the opening book for the white player. The method of generation is similar for red.

A file storing the opening positions is first initialized. A program parses this file, and whenever it encounters an unmarked position, it marks the position and computes the best move using a search depth of 10. Then it writes down the best move in the file, as well as append into the file all the possible moves by red after white takes that best move. After doing so, the program simply looks for the next unmarked position and repeats the above process, until the encountered positions reach a certain depth (*i.e.* the total number of pieces on the board reaches a certain number).

Since the evaluation of one position is independent of the evaluation of another, we can take advantage of this and allow the opening book to be generated with parallel processing. The marker mentioned above is the mechanism for ensuring mutex among the different processes - each process must be evaluating a different position than others.

After running the opening book generator for 3 days on 6 machines in sweet hall (with a `+10 nice` parameter), we get the opening books for both red and white. The white opening book contains 2848 positions, which are all possible moves up to 14 pieces on the board. (6 moves for white) The red opening book contains 16821 positions, which are all possible moves up to 13 pieces on the board, plus some possible moves up to 15 pieces on the board. (4-5 moves for red) Given the huge size (262144 $\times$ 4 entries) of our hash-table, we are able to store most if not all of these positions without collision.

The result of the opening book is that the program takes no time at all to figure out the first 5-6 moves, giving a huge time advantage over other programs without such a feature. Also, since the opening book is generated using a deep search of 10 plies, the responses are generally better than those calculated in real-time, when the program is under a time-limit. This gives the program an advantage of a better starting position since most programs cannot afford a search depth of 10 at the early stage of the game under the 150 seconds time limit.

## 6 The End Game Search

Towards the end of the game the branching factor decreases drastically. Using our efficient bitmap-based data structures we are able to do a 14-ply end-game search to solve the end-game completely. Since we are interested only in the difference in the number of pieces, the only feature taken into account is the piece difference, and we don't waste time to look at other features. The end-game search is activated only when there are 14 or fewer empty squares on the board. While in most cases the end-game search finishes in 5 seconds, it is activated only when there are more than 25 seconds left as a precaution against running overtime.

## 7 The Training Algorithm

We deviate from the handout the most in our training algorithm. We did implement reinforcement learning (see section below), but we need a better way to get good weights since we divided the game into 88 stages. Using gradient descent we need many training instances to get converging weights. This process is very time consuming if done sequentially. We devised a way to do training in parallel so that we are able to obtain a lot of training instances in a limited time.

### 7.1 Original Algorithm – Reinforcement Learning

We originally trained our program using reinforcement learning. However, since we have switched to our new training algorithm, we have taken the code for reinforcement learning out from the main program to avoid confusion. In order to show that we have implemented the algorithm at some point, the code for reinforcement learning is now in the file `reinforcementLearning.c`, which is not compiled in our final program submission.

#### 7.1.1 The Algorithm

We basically followed the hints given under '**Method 2**' in the handout for reinforcement learning. For each stage in the game we recorded the weights for each of the features that we had, according to the value of the current state (*i.e.* for each state $s$ we compute the game score $V(s) = \sum_{i=1}^{n} w_i f_i(s)$, where each $w_i$ is a weight value and each $f_i(s)$ is the value of a feature, and $n$ is the total number of features.). We selected the state $s'$ for comparison using the minimax algorithm with a constant search depth (we decided to use a search depth of 4 for training). We then updated each of the weights using the update rule

$$w_i \leftarrow w_i + \gamma f_i(s)(V(s') - V(s))$$

as described in the handout.

#### 7.1.2 Program Structure

As mentioned the original code for reinforcement learning resides in `reinforcementLearning.c`. Two functions are included:

- `WriteOutWeights`, which is called after one game has ended to write out the weights for each stage to file.

- `RecordWeights`, which was is called inside `ExploreTree` after each subsequent state is found in `gameEngine.c` and implements the reinforcement learning algorithm. Since `ExploreTree` has already located the optimal subsequent state using the minimax and alpha-beta pruning algorithms via the function `FindBestMoveSmart`, the game score for the subsequent state has already been computed, and is passed into `RecordWeights`. So `RecordWeights` only need to compute the game score for the current game.

#### 7.1.3 Results

Since we switched to our final learning algorithm rather quickly after realizing the new algorithm's benefits, we did not carry out a lot of experiments with reinforcement learning. As an example of running the program, however, we have included the output weights for a trial run in the appendix, where each line represents one stage in the game and each number represents the value of a weight for a feature (from left to right the numbers represent the following features: mobility, potential mobility, piece difference, corners evaluation, x-squares evaluation, and the last feature is no longer used.). In the trial run $\gamma$ was set to be 0.01.

### 7.2 Our Algorithm – Parallel Supervised Learning

#### 7.2.1 Basic Procedure

First we divide the game into 88 stages. The stage of a given board position is given by

$$\text{number of white pieces on board} + \text{number of red pieces on board} - 4$$

This is a good approximation of the stage of the game since each player puts down a piece in every move, and we want a measure of how far the game has proceeded.

As in reinforcement learning, our goal is to find weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=0}^{n} w_i f_i(s)$ gives a good evaluation of the state $s$.

In reinforcement learning we used the minimax value of a 4-ply depth search on the state $s$ to be $v'$, and used the weight update rule

$$w_i \leftarrow w_i + \text{rate} \times (v' - V(s))$$

Our object is to make the $\sum w_i f_i(s)$ approximates $V(s)$ as closely as possible.

Since we are able to solve for the end-game completely at the final stage of the game, our training starts at those stages. We play the game up to stage 73 (15 moves from the end-game), then use the minimax result of an end-game search as the $v'$ of the stage.

Instead of using stochastic gradient descent, we simply generate a lot of training samples by having the program play against itself, and record the values of each feature and $v'$ of each state s in a file. After several thousand games are played, the training data for each of the 15 stages are fed into a least-square fitting program to find the weights for the stages so that the mean squared error is minimized.

#### 7.2.2 Discussion

We did not use stochastic gradient descent for a few reasons:

Firstly, the value $v'$ for each state is independent of the weights being trained. So it does not matter if we update the weights or not as we do the training.

Secondly, generating training samples for batch processing offers the possibility of parallelism. We can run the sample-generators in parallel since one game is independent of another.

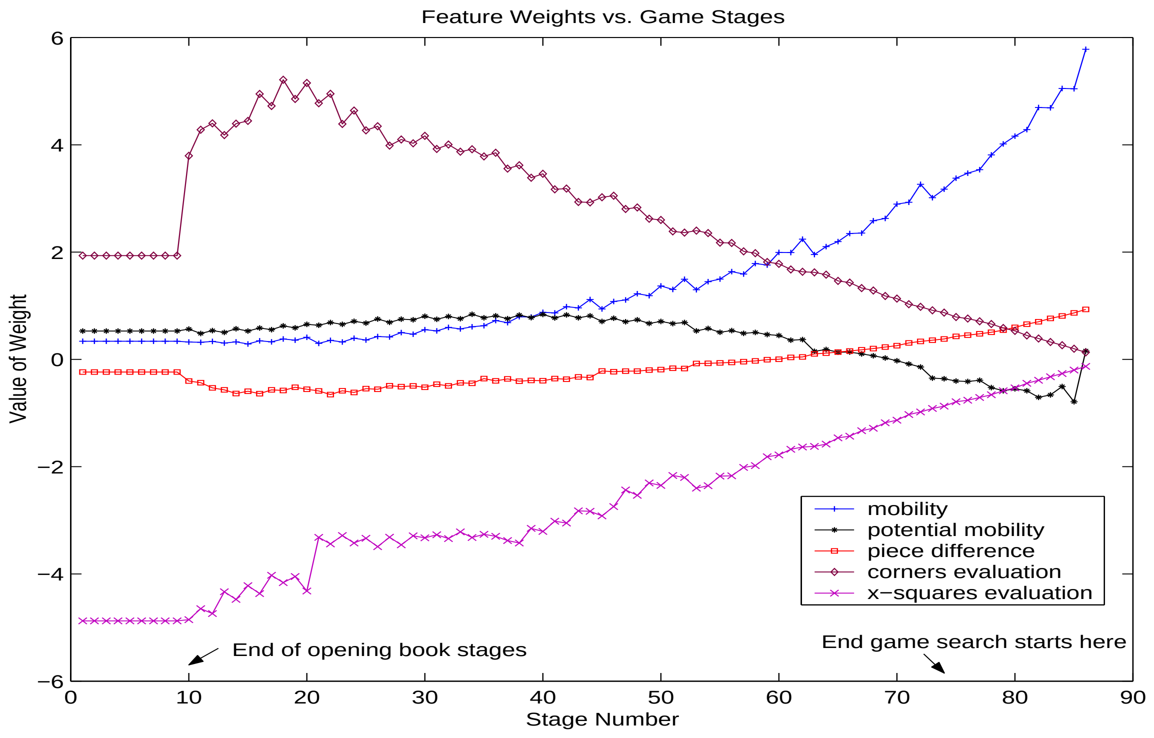
Thirdly, gradient descent is prone to the problem of local minima. Using the standard technique of least-squared fitting in numerical analysis guarantees the convergence of the weights and the resulting linear function minimizes the mean squared error.

It is important to strike a balance between exploring the game tree and getting samples that approximates the states we actually encounter in the tournament. With this in mind, we played the first 25-30 moves randomly, and used a search depth of 6 for both sides for the subsequent moves, until stage 73, when the training actually begins. The random moves allows exploration of different parts of the game tree, while the searched moves ensure that by the time stage 73 is reached the state is a good reflection of what will actually happen in the tournament.

After training the last 15 stages of the game, we get the optimal weights for those stages. We then used a similar strategy to train the previous stages – the program plays against itself until 10 moves before the first trained stage, then the training begins using the minimax value of a 10-ply search as the $v'$ of the state. Since such a search will return the evaluation of the state using the weights that are 10 moves ahead, and those weights are already the optimally trained weights, we get a very good approximation for $v'$.

We also adopted the suggestion of the handout and took an exponentially weighted average of the even-ply $v'$ of the states encountered as we go along. In this way we get the training data for the 10 moves before stage 73, and so on.

This training process results in a lot of computation (since we need to play thousands of games for every 10 stages, although later we switched the search depth to 8 as the deadline approaches). And we need to do a least-square analysis for each stage of the game. The result is 78 sets of weights, from the end of opening book till the beginning of end game search. Although the sets of weights are trained separately, they show an interesting trend across the different stages:



At the beginning of the game corners and X-squares are weighted most heavily. Second to these are mobility and potential mobility. Piece difference is surprisingly weighted negatively, which agrees with the well-known strategy of minimizing the number of pieces at the beginning of the game.

As the game proceeds, mobility becomes increasingly important, while corners and X-squares gradually becomes secondary. Piece difference weight goes from negative to +1 at the end-game.

The most fascinating aspect of this learning algorithm is that there is absolutely no human knowledge spoon-fed to the program. The program picks up knowledge about the game from the end-game search, and gradually propagates the knowledge backwards. Strategies such as minimizing the number of pieces at the beginning of the game and maximizing mobility throughout the game are results from the analysis of the game through self-playing, and coincides with commonly used strategies by human experts.

## 8 Time Management

### 8.1 Basic Considerations

We need to come up with a time management algorithm that is able to accommodate the following restrictions:

- Since we are using the opening book there is no need to allocate much time for the steps where the opening book is used.
- Since we perform a complete end-game search near the end of the game, there will be one step that takes an unusually long time (in order to do the search) before being able to make the next move, but moves after that do not require much time since we have already found the optimal goal.
- At each stage we need to determine how deep (in terms of plies) we should search for the next move. While the number of levels is proportional to the amount of search time, it is not the case that the time required to search at a particular level is the same in all stages of the game. This is because the branching factor at each level is different as the game progresses, so we simply cannot have a table that lists the time needed to perform a search at certain level and gauge our time management in that manner.

### 8.2 Basic Algorithm

In order to cope with the above restrictions we use the following plan for time management:

- During the stages where the opening book is used, we do not put a time limit for the execution of those stages, since those stages require extremely little time.
- We set a stage number where the end game search is performed (currently set to step 74, provided that we have at least 25 seconds left in the game). We currently allocate 15 seconds for the search to complete. Like the opening book stages, we do not put a time limit for the subsequent stages since they also require a very short amount of time (see 'Advanced Features' for more information).

For the remaining stages we perform two tasks:

- We determine the time that we allocate for each step. This is found by multiplying the total amount of time left in the game by a ratio, which is calculated by taking the ratio between the 'time weight' for the current stage and the sum of time weights in previous stages. The values of the time weights were found by experimentation and were hand tuned (the weights are contained in the file `timeweights.dat`). Basically we allocate the most amount of time to the stages immediately following the ones where the opening book is used, and gradually decreases as we move towards the end game stages.

- We estimate the time that we spent on each node and the branching factor in the last search. The time per node is estimated by dividing the search time in the last stage by the total number of nodes that were examined in the previous stage. The branching factor, meanwhile, is estimated by

  $$\sqrt[\text{number of levels that were searched last time}]{\text{total number of nodes searched}}$$

  This is a reasonable estimate since the branching factor is relatively constant across each level in the same stage of the game.

With these data we estimate the number of levels that we can search in the subsequent step using the formula

$$\text{time} = (\text{last branching factor})^{\text{level}} \times \text{time spent per node}$$

And we take the maximum level that we can accommodate without using more time than we were allocated.

## 8.3 Advanced Features

Our time management strategy has the following features:

- **Dynamic Scheduling**

  Note that our time management algorithm is neither pessimistic nor optimistic. It is generally a dynamic management algorithm: if one stage ends up using less than the time that it is allocated then subsequent stages will share that extra time. If it ends up using more time than allocated then all subsequent stages will cut down in their allocated time. We chose such algorithm to ensure that we do not run out of time in the game. Thus we would rather have each stage use less than their allocated time and have time left at the end rather than pushing to the limit and bet on our luck during the tournament.

- **Emergency Search Abort Mechanism**

  In addition to the dynamic time allocation, we keep track of the time as we do the search and call for an abort of the search if the time spent on it exceeds twice the allocated time. This is done by calculating the time spent on the search at every node at the top three levels of the search tree. This is necessary since the branching factor is not constant throughout the game tree, and sometimes the time management module underestimates the branching factor based on previous moves.

  After the abort, the time management procedure reallocates the time and recalculates the search depth, and the search is performed again. The information from the previous search is not totally wasted since most of them should still be in the transposition table.

  An alternative solution to this is to use iterative deepening as the search algorithm. In that case we can just return the result of the search to the last level in case the current search exceeds the time limit. We have implemented iterative deepening and compared the result with our current algorithm. It turns out that iterative deepening slows down the search by so much that it is not a good choice. In fact with the transposition table the time needed for doing another search is very short compared to the original search.

## 9 Overall Performance

In tournament conditions, with the time management and opening book in effect, the program achieves an average search depth of 7 at the early-game to mid-game. The search depth gradually increases near the end-game stage, reaching an average depth of 9 to 11, before the 14-ply end-game search, and it usually finishes with over 30 seconds remaining.

## 10 Experiments with Different Versions of Lap Fung the Tortoise

Our trained program performs much better than the untrained version. This improvement increases with the search depth. The reason for this is that we placed a lot of emphasis on the corners and X-squares evaluations in the hand-tuned version. The trained version sometimes are more willing to take the X-squares than the hand-tuned version (in trying to increase its mobility), resulting in risky moves that should be played only when a very deep search is performed. Since we are under a time limit in the tournament, the weights for corners and X-squares should be given more emphasis than what our training reveals.

To confirm this claim, we did an experiment using a version of the trained program with the weights for X-squares and corners scaled by a factor of +2. The program with scaled weights is played against the original program. On a saga machine with 2 CPUs, the original program beats the scaled program (thus confirming the optimality of our trained weights). However, under tournament condition, the scaled program adopts a more aggressive strategy in taking the corners and forces the original program to lose due to the loss of the corners.

| Setup of White Player | White Score | Red Score | Setup of Red Player |
|---|---|---|---|
| T×2 | 53 | 39 | T |
| T-5 | 36 | 56 | T×2 |
| T-6 | 41 | 51 | T×2 |
| T | 36 | 56 | T×2 |

Legend

| | | |
|---|---|---|
| T×2 | – | the program with scaled weights under tournament condition |
| T | – | the trained program under tournament condition |
| T-5 | – | the trained program using a fixed search depth of 5 |
| T-6 | – | the trained program using a fixed search depth of 6 |

In the mini-tournaments, the trained program with scaled weights performs the best. So it is chosen to participate in the tournament.

## References

[1] Anderson, Gunnar. The inner workings of strong Othello programs. <`http://www.nada.kth.se/~gunnar/howto.html`>.

[2] Lee, Kai-Fu. A Pattern Classification Approach to Evaluation Function Learning. Pittsburgh: Carnegie-Mellon University. October 1986.

[3] Lee, Kai-Fu & Mahajan, Sanjoy. BILL: A Table-Based, Knowledge-Intensive Othello Program. Pittsburgh: Carnegie-Mellon University. April 1986.

[4] Galvin, Peter & Silberschatz, Avi., Operating System Concepts. 5th ed. New York: John Wiley & Sons, 1999.