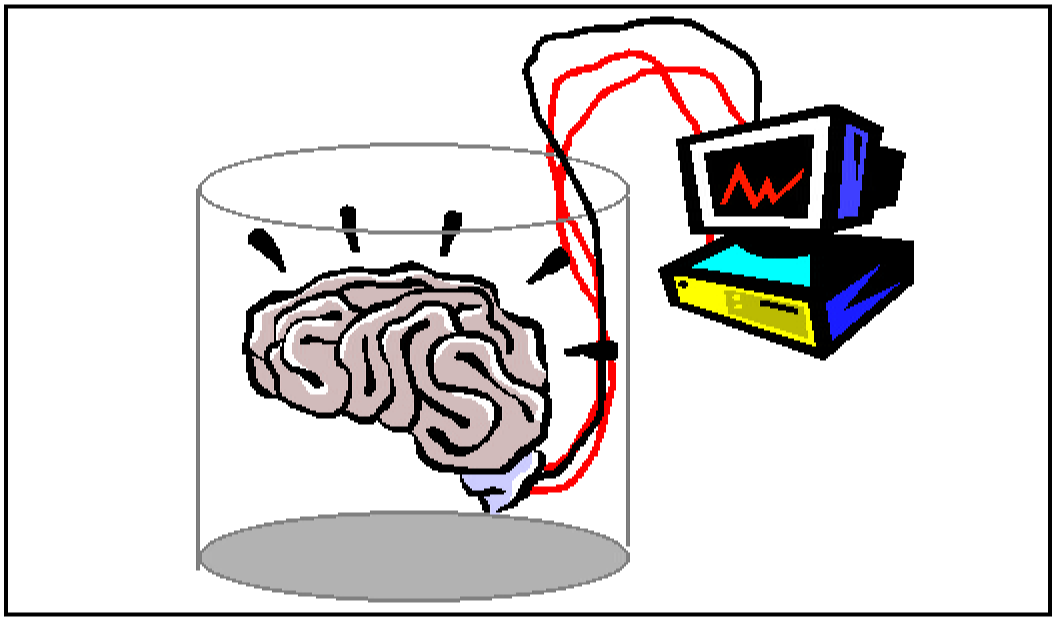


“Brain In a Vat”

Othello Report

Joseph Smarr & Andrew Waterman

Basic Program Structure

Client architecture.

Each turn consists of:

- Calculate the optimal search depth (see Time Management Strategy).
- Find "best" move using α-β pruning in a minimax search.
  - Keep record of the TD(λ) value during search into the tree
- Adjust feature weights using reinforcement learning with the TD(λ) value (see Training Our Program).

Search algorithm.

Our search algorithm integrated several extensions into the normal α-β pruning search algorithm. Highlighting those extended features, our general algorithm is as follows:

```
Maximizer(a , b, boardState, V', depth, l count, …):
    Using GetOrderedLegalMoves, order the successors moves from best evaluation to worst
    If there are no legal moves remaining, return the Greedy evaluation on this node
    For each successor move (in this order)
        Evaluate the move by recursing with Minimizer
        bestSuccessor = the move with the highest evaluation
        Prune normally using $\alpha$ and $\beta$
    V' = V' (no change for odd layers in the tree)

Minimizer(a , b, boardState, V', depth, l count, …):
    Using GetOrderedLegalMoves, order the successors moves from worst to best evaluation
    If there are no legal moves remaining, return the Greedy evaluation of this node
    For each successor move (in this order)
        Evaluate the move by recursing with Maximizer
        bestSuccessor = the move with the lowest evaluation
    $V' = V' + \lambda^{(depth+1)/2}$ * Evaluation(bestSuccessor) → Adjust V' for even layers of tree
    $\lambda$count++ → keep track of how many times $\lambda$ has been multiplied
```

Using this algorithm, we are able to implement several extensions, each of which contributes to the Othello player's efficiency and competitiveness.

Ordered a -b pruning.

During α-β pruning, the goal is clearly to prune the most nodes to save the most time. Since nodes are pruned when they cannot have a better evaluation than nodes already evaluated in the tree, we can prune more nodes by exploring game states with better evaluations first. To take advantage of this property, we implemented a method, *GetOrderedLegalMoves():*

- Store all legal moves from a given board position
  - (using a "move_t" data structure that contains a row, column, evaluation, and player key)
- Use qsort to sort these moves in order of their evaluation,
  - If the move is for the maximizer → sort from highest to lowest evaluation
  - If the move is for the minimizer → sort from lowest to highest evaluation

Thus, we can continue with normal α-β pruning, now searching a node's successor moves in order of their evaluations. This implementation brought a dramatic increase in the number of pruned nodes (see Experiments and Analysis).

TD(l) generation of V'.

To generate a "target" value for our reinforcement learning algorithm, we use an exponentially decaying average of the value at successive moves. As shown above, our α-β pruning algorithm maintained a calculation of this exponentially decaying average. Once the search of the game tree has completed, we have two important values:

(1) λcount = the number of λ factors incorporated into V'

(2) $V' = \lambda \cdot V(\text{State at depth}=2) + \lambda^2 \cdot V(\text{State at depth}=4) + \cdots + \lambda^{\lambda\text{count}} \cdot V(\text{State at depth}=2\cdot\lambda\text{count})$

This allows us to normalize V' correctly to the range [-1, 1] that matches the normalization of the weights and other features, as follows:

$$V' = V' / (\lambda + \lambda^2 + \lambda^3 + \cdots + \lambda^{\lambda\text{count}})$$

Training Our Program

Once the search of the game tree has finished and a move has been selected, we use the V' value obtained by the TD(λ) algorithm to reinforce our program's weights. We implement the basic gradient descent reinforcement learning rule, comparing the current game state S to a future game state S' using the selected move, positing future moves by the opponent as given by the minimax search tree as delineated in Method 1 of the Othello handout, rather than by waiting for the opponent to actually take a move. Then, we will update the weight w_i corresponding to each feature_i in our evaluation function:

$$w_i = w_i + \gamma \cdot \text{feature}_i \cdot (V' - V(s))$$

where γ is the learning rate, set within the range γ: [0.05, 0.1], V(s) is the evaluation of the game at state s, and V' is the value obtained above using TD(λ).

We chose this algorithm for a number of reasons. We want to compare the game state S to those states S' that are immediately after S. There is less room for inaccuracy in the game tree between S and S' brought by misjudging the opponent's moves. However, we also want to compare S to those states S' that are near the end of the game tree, because those states will be more representative of the actual va lue of the game, given a move from S. Hence, with good reason, TD(λ) computes a weighted, exponentially decaying linear sum of the values of states further into the game from S. TD(λ) thus preserves information about the different stages of the game tree that are important for correcting errors in our evaluation of the game state at S.

Unfortunately, this reinforcement algorithm is quite complex, incorporating subtle factors into one "overall" evaluation to compare to S. Because of its complexity, it is difficult to test explicitly the algorithm's exact benefits, as compared to a simpler calculation of V'=V(S') on only the game state two moves away from S. Nevertheless, the merit of the TD(λ) algorithm is well-established, and so we feel comfortable depending on it. Generally, we know that a myopic look on the game like the simple V'=V(S') would lose sight of the possible ramifications of a move down the line in the game. Whereas TD(λ), on the other hand, preserves information about features of the game further down the line, which would clearly provide a more "omniscient" way to update the weights accurately.

In preparation for the tournament, we trained our Othello program for approximately 6-8 hours, playing against itself. We kept two separate weight files (separately for the white and red players), and we wrote out the updated weights after every turn. This provided several advantages:

(1) The weights could more easily converge by having two copies of our program running against each other, using possibly different sets of weights. By the nature of having two different sets of weights, there is an added randomness to help avoid falling into local maxima.

(2) We could tune each player to fit the advantages and disadvantages of being the black or white player. There are certain advantages in being the first or second player to move, and correspondingly, there are different types of playing styles that would best suit each player. Our two players are able to do just that, by training each other to fit the role of the white or the black player.

See Experiments and Analysis for more information on the results of training the weights that we observed. In retrospect, we are sure now that we should have spent more time training the weights. Also, we would have tried more than two different combinations of weights, and let the weights converge toward some "master" player. Unfortunately, due primarily to time constraints, this was not possible, and we did what we could to let the weights train.

Time Management Strategy

We managed our time by restricting our search depth on a move-by-move basis. We determined the “optimal” search depth as a function of the time that had already expired and the number of moves we had already made. To find the optimal search depth, we used the product of two competing interests: a conservative bound that prevented time-faulting, and a priority ramp-up which placed less emphasis on the beginning of the game.

Our conservative “allowable depth” was calculated as follows:

$$\mathbf{AllowableLookAhead = MaxLookAhead*(1 - TimeUsed^3)}$$

where MaxLookAhead is our default search depth (we found 6 to be the best balance of depth and speed), and TimeUsed is the fraction of the clock that has already expired. This function starts at 1 and decays down to 0, but it starts moving shallowly and then drops off steeply at the end. The idea was not to get in the way except towards the end when the depth had to be reduced in order to prevent time-faulting. This strategy proved effective at preventing time-faults even on depth 10, while still allowing minimal interference in the desired depth. The problem with just using this function as our time management strategy was that it allowed full depth at the beginning, so all our time was soaked up on the first few opening moves and then we ran out of gas for the rest of the game.

To compensate for this effect, we also calculated the “performance priority” for each move, which was a value from 0 to 1 of how important it was to do well on this move. The observation was that in the first few moves, there is only so much one can do and it’s not terribly important as long as you stay alive. Therefore the performance priority starts at 0.5 and linearly ramps up to 1.0 as a function of the number of moves that have been made. It peaks at move 20, at which point its effect ceases. It was calculated as follows:

PerformancePriority = 0.5 + 0.5*NumberOfMovesMade/20

The combined effect of these two features was to start out playing quickly, and gradually take more and more time, peaking in mid game where we felt it was most important to play well. As the game progresses, if time is running down, our depth gets throttled back so we don’t time fault. This throttling wasn’t as harmful to our performance as we originally expected it would be, in part because in the endgame you start running out of available moves and finding terminal states, so we weren’t actually “missing” that much by running shallower searches. For detailed analysis of the results of this time strategy, see Experiments and Analysis.

Our Evaluation Function

We started out with a fairly standard set of "piece counting" features, all of which we normalized to [-1,1] to ensure consistent scaling when performing reinforcement learning:

Greedy. (# pieces) – (# opponent pieces). This is obviously important, since whoever gets the most points wins the game.

Edges. (# pieces on an edge (including corner)) – (# opponent edge pieces). This is important since edge pieces are harder to take than pieces in the middle of the board, and they can be used to form a base for attack.

Corners. (# pieces in a corner) – (# opponent corner pieces). This is important since pieces in the corner cannot be captured and thus are very useful anchor positions.

We then added two additional "strategic" features, which examined how advantageous the board was for future states of the game. At first, we thought that explicit strategy features might not be necessary, since our search looks many moves ahead and could implicitly incorporate such properties of the game into the searches that are already being executed (such as if the search were to "deduce" that the second row is bad since it leads to the opponent taking the edges). However, we felt it was important to explicitly incorporate these properties of the game, for otherwise they might be ignored in lieu of other features with greater weights. In fact, our strategy features turned out to drastically improve the competitiveness of our AI, as well as the appearance of its "thoughtfulness".

Mobility. (# available moves) – (# available opponent moves). This turns out to be important because the more potential moves you have, the more flexible your strategy can be, and thus the more you can pay attention to capturing important regions of the board. Similarly, the less moves your opponent can make, the less likely they are to be able to mount a clever strategy, and the more likely they are to be forced into mediocre moves. The addition of this feature seemed to help us most in the first third or so of the game, because it caused us to really clamp down on the opponent early on and keep it from developing much. In some cases we actually won by completely eliminating the opponent's pieces, often when we ourselves only had 20 or 30 pieces on the board.

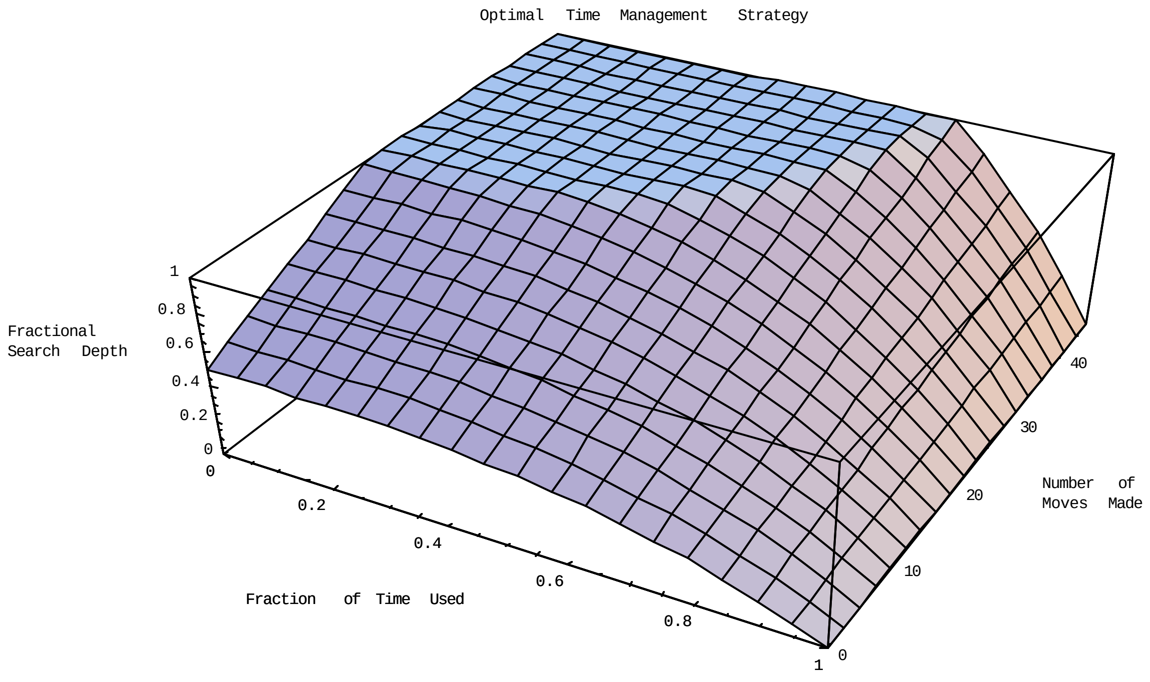
Vulnerability. (# vulnerable opponent pieces) – (# vulnerable pieces), where "vulnerable" means the piece is in a square adjacent to an edge, at least one of the piece's adjacent edge squares is empty, and in the reverse direction along that same line (from the empty edge square through the piece in question), there is an opponent's piece. This feature avoids giving up control of the edge to the opponent, by weighting against taking a piece next to an edge which would allow the opponent to take this piece and place a piece on the edge. However it will go ahead and move into a square next to the edge if it deems the move "safe", i.e. if there are only pieces of the same color going back along the line, if the line is backed by an empty square, or if the line of our pieces runs to the boundary of the board. This was an immensely helpful feature at not losing control of the board, especially early on, and in fact often led us to "sucker" the opponent into taking us by placing a piece in the adjacent-to-edge squares, from which we were able to take control of the edge ourselves.

Experiments and Analysis

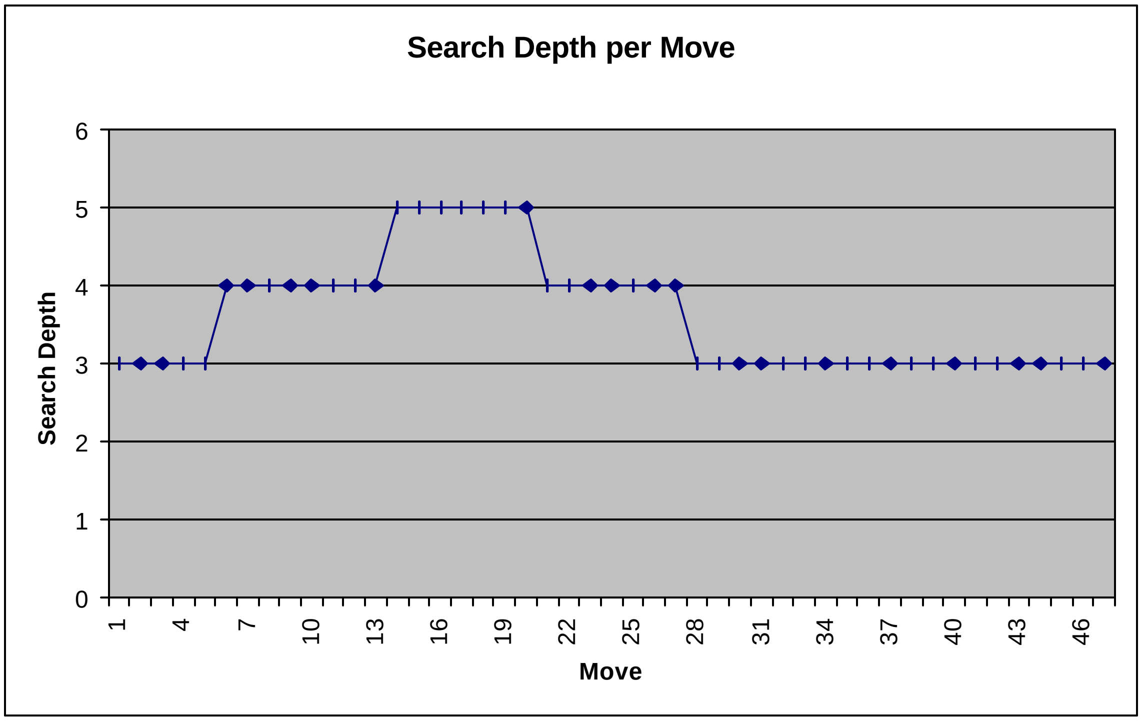
We gathered a large collection of performance data based on our final Othello program playing white against a greedy black (since this was a fast opponent and we were mainly interested in the internal behavior of our AI). The following graphs and analyses were generated with a default search depth of 6, and with reinforcement learning turned off.

Game Strategy: Theory vs. Practice

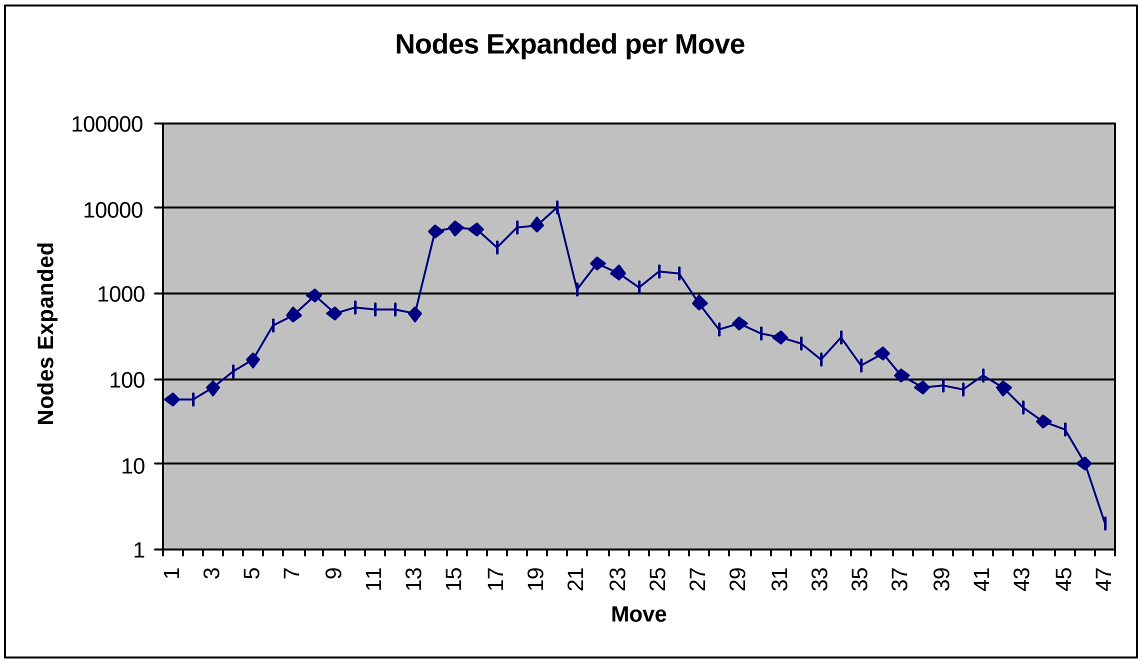
Our time management strategy predicts that our search will ramp up early on, plateau in mid game, and throttle back towards the end. Our optimal search depth can be viewed as a function of the time elapsed and the number of moves made so far.



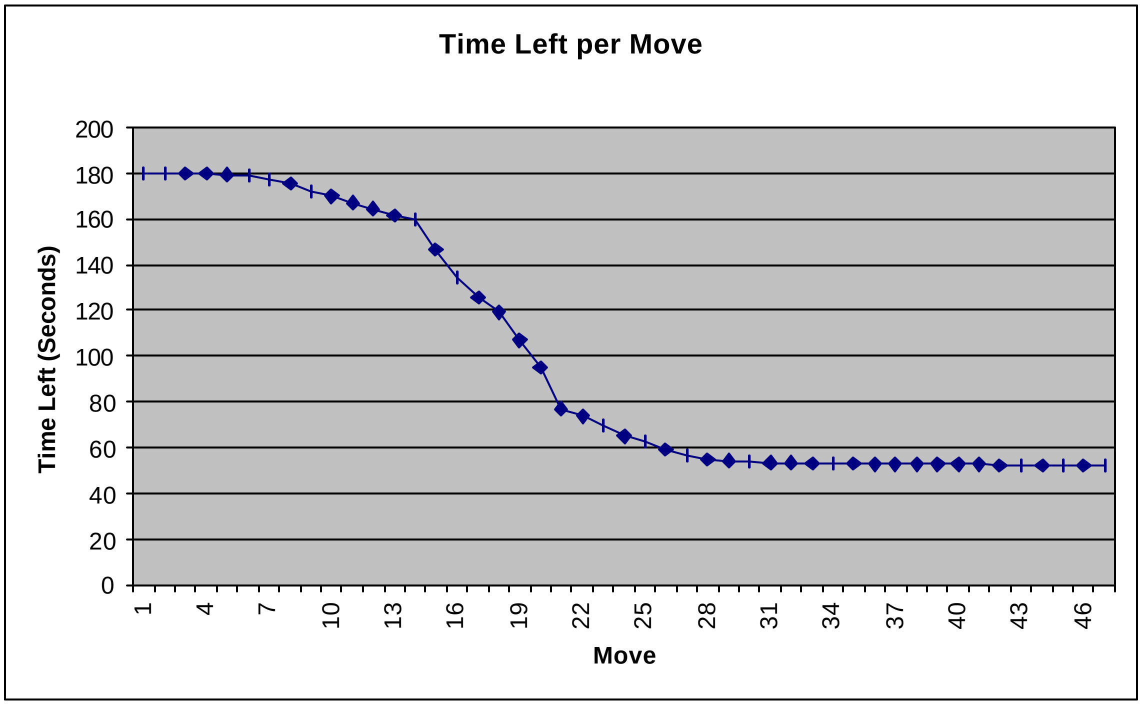
Note that while both these factors increase monotonically, they are not directly correlated, since time used per move is exponentially dependant on search depth. Thus the expected "path" of play along this surface will resemble a horizontal line across this page. Both moves and time elapsed will increase together, so the search depth should ramp up, ride the ridge in the middle, and then die off at the end. To test how well we were actually achieving this desired performance, we collected and graphed the actual search depth used at each move in a game between greedy and white of the form mentioned above:



This was an encouraging sign since it followed the expected behavior. Next we observed the effect this strategy had on the total number of nodes expanded, recognizing that the beginning and end games will not only be pinched for depth, they will have less of a branching factor due to limited move availability.



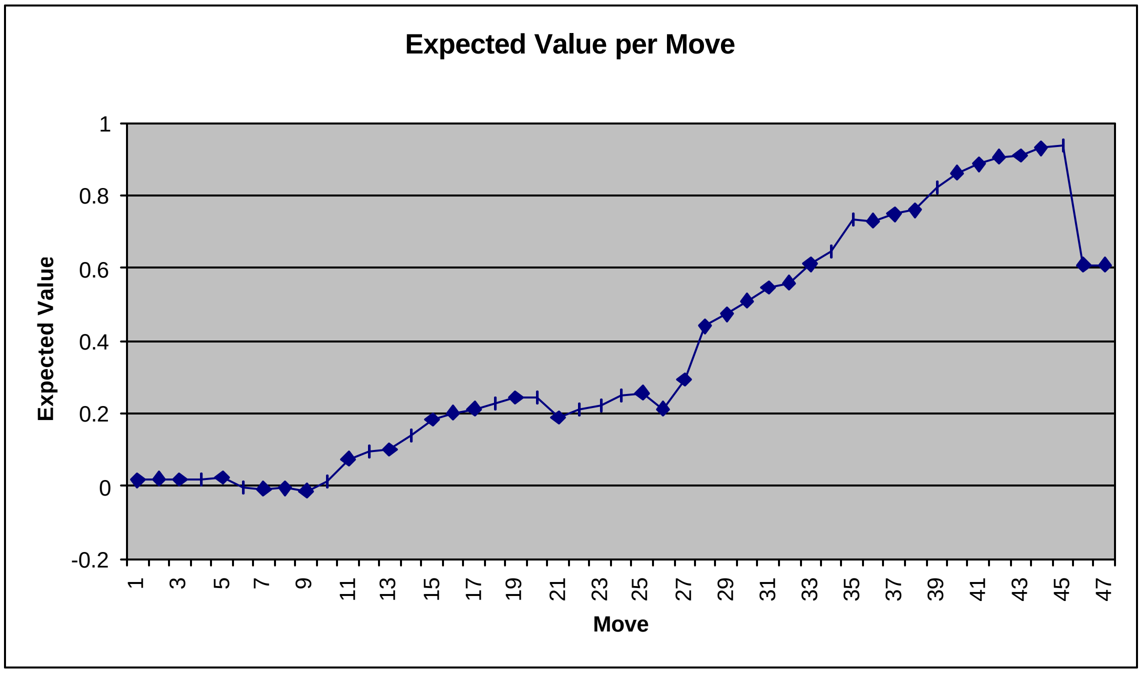
Notice that the y-axis here is logarithmic, and it gives us the expected shape of the curve. If we looked at this graph with a standard y-axis, the middle would be disproportionably high, since the number of nodes expanded for a search grows exponentially with the depth. This was all well and good but we wanted to measure how we actually handled game time when it was all said and done.



This is close to the behavior we were shooting for, because we focused most of our time on the important middle game. However we observed while watching our AI play that it seemed to "stall out" a bit too early, and it started searching quite conservatively even when there was still some time left. This is documented above in that the search depth bottomed out at 3 by move 30, and it is perhaps even more clear in this time graph, where we leave about 50 seconds on the clock from there on. Ideally our curve would have continued the downward trend of mid-game longer and bottomed out later and lower. Part of the problem with realizing this more optimal goal is the coarseness of search depth as a regulator of search time, since adding even a single extra level in the search tree can double search time or worse.

Evaluation Function: Is it accurate?

As we added more evaluation functions, we became concerned that our AI would not sufficiently realize that in the end the game is decided only by piece differential, and that losing in a tactically advantageous position is not a desired outcome. We decided to test whether our evaluation function was deceiving itself by plotting the expected outcome of each move and seeing how reality agreed with our predictions.



For the most part this graph was very good news—as our player took more and more control of the board, it didn't seem to get too cocky, nor did it undervalue its progress. The little dips indicated periods where black mounted a bit of an offensive, and it was comforting to see our evaluation function react to reflect this turn of events. Looking several moves ahead certainly helps smooth this curve it seems. It was somewhat disturbing however that we gave up so much ground at the very end of the game. In the last couple of moves, black took back quite a number of pieces. We're not quite sure why our look-ahead didn't pick this up and plan accordingly, but it provides good evidence of how a pre-calculated endgame database could potentially be of immense value since you would know near the end whether your current position was sustainable and not lose valuable footing at the very end.

Ordered Alpha-Beta: How Much More Do We Prune?

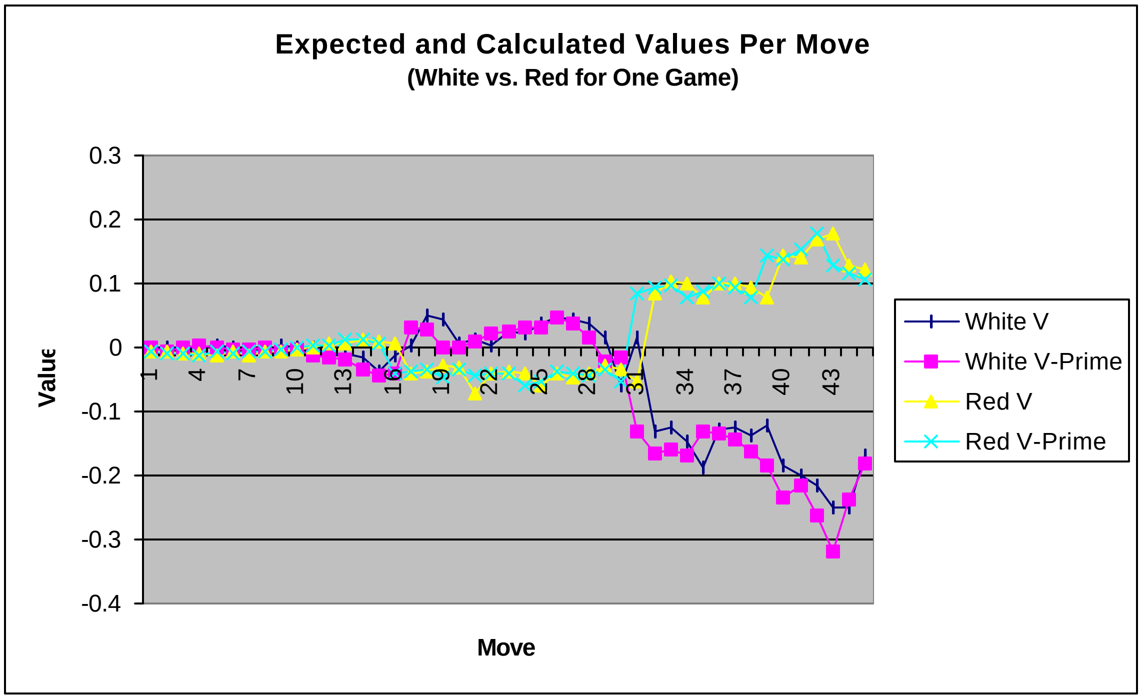
When we first implemented alpha-beta pruning for the milestone, we saw a dramatic reduction in the number of nodes expanded during the opening move for any given depth. Our hope was that by ordering our successor nodes by their heuristic value, we could increase our chances of finding a good move first and thus give alpha-beta more power. We tested our new search on the same conditions as standard alpha-beta pruning—the opening move at depth 4, 6, and 8.

Depth	Standard	Enhanced	Savings
4	131	65	50.4%
6	1283	375	70.8%
8	11658	2736	76.5%

This technique resulted in a tremendous amount of extra pruning! Furthermore, the deeper the search depth, the more we saved, since pruning a tree early means an exponentially less number of nodes to search. We originally intended to perform more extensive tests of our enhanced search, but these results were compelling enough that we focused our attention on areas where the advantage was less clear.

V-Prime and TD(l) Evaluation: Are We Being Led Astray?

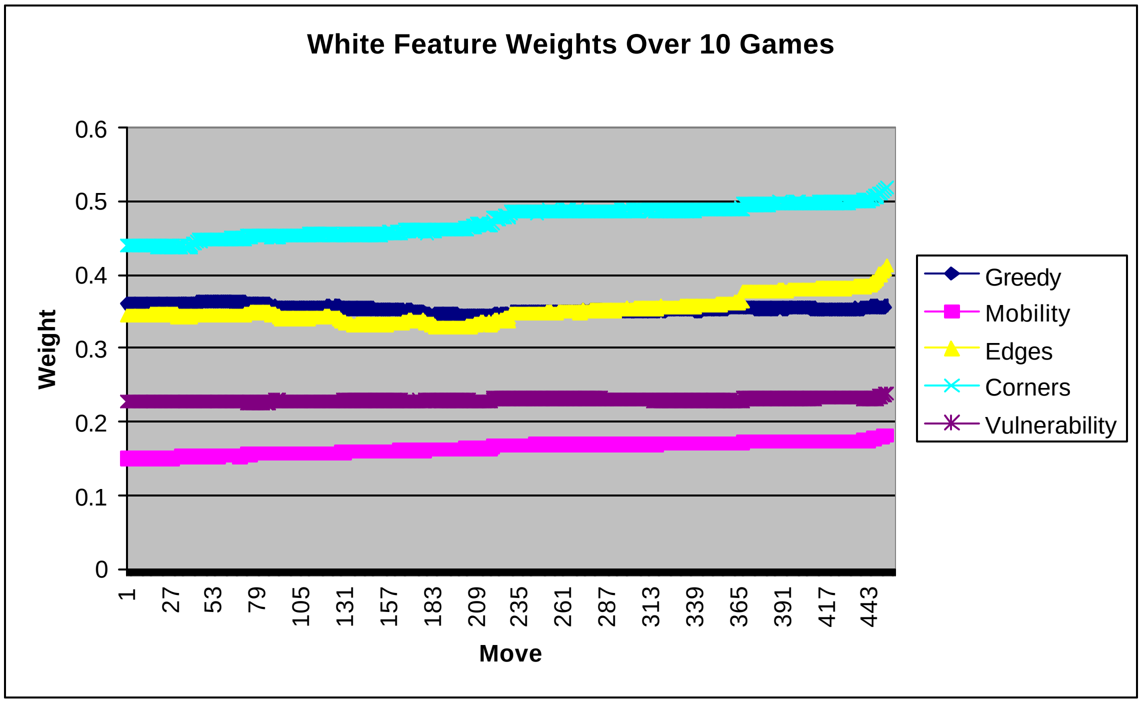
We had faith that reinforcement learning would make our AI smart even though it was only guessing. We believed that TD(λ) would help this process even out. But we always live by the maxim: trust but verify. Therefore we pit our AI against itself and recorded the expected value and calculated v-prime for each move to see whether our predictions were proving accurate:



This is a great graph, because it really shows the back and forth struggle between the programs, the eventual pull-away of the winner, and that v-prime is constantly staying ahead of the game. There is an apparent symmetry to the expected values of white and red, which is good since this is a zero-sum game. We see white winning in the middle, but red bides its time and pulls away at the end with a decisive win, though white does manage to close the distance somewhat in the very end. But the important detail we were looking for is the relationship of v and v-prime for both white and red. As you can see from the graph, our v-prime is doing a very good job of anticipating where the game is headed, which means that it will indeed make a good teacher.

Learning: Are We Seeing Any Progress?

We conclude our set of analyses by examining what (if anything) our program is learning by using reinforcement adjustment of its weights. We set up a match of white vs. red with our program playing both sides, and ran it for 10 games, tracking the weights for both sides at every move. We used a learning rate of 0.1 which we kept constant the whole time.



We regretted waiting to create this graph until after the tournament was held, because the signs are clearly here that more training would have given us a different breed of player. Over the course of these 10 games, the importance of grabbing the corners was noticeably increased, and in truth that was a weakness we observed in the tournament. Mobility also became more weighted, which was somewhat unexpected to us, though we did start it with quite a low value so this may have simply been compensation. Perhaps the most telling effect is the crossing of edges and greedy evaluation weights—a symbol of strategy overcoming impetuousness, a sign of experience!

Thoughts for the Future

There are a score of ideas we had at the outset of this project and that we thought of along the way, but that just never got implemented in this triage process we call schoolwork. While we don't want to take too much time talking about could-haves, one area that seemed very promising and tangible that would directly relate to the reinforcement learning is the use of a genetic algorithm to create a population of players with different feature weights, each of which would play and reinforce in parallel. After a predefined period of time, the players would "procreate" and the create copies of themselves with slight mutations. The most successful programs so far would procreate the most, and the least promising ones would be killed, so that our resource base would not grow exponentially. Since picking initial weights and finding good maxima is one of the most important elements of success in reinforcement learning, we thought this would be a great way to effectively "explore the terrain."