Threes AI

Evan Benn 20369194

Emily Martin 21153667

Executive Summary

We implemented 4 heuristics and 3 graph traversal algorithms to develop a game playing agent for the game Threes. We observed that depth was critical in performing well, although due to the exponential time requirement with depth a more cunning strategy was required to improve performance. We split the game player into an easy(fast) and hard(slow) search algorithm, and used the performance of the easy to allow deeper searching in the hard as required. The heuristics we implemented were very simple and fast to calculate, but still showed good performance in our priority queue and ranking.

Threes Theory

We began our investigation with an analysis of the game being studied. Threes has no goal state, instead there is a metric, the score, which is to be maximised. The game ends when no more moves can be played, and the score increases with any valid move. The branching factor is at most 4, although the average value in practice is extremely dependent on the strategy implemented. Move sequences of length greater than 6000 were seen in our investigation, with some teams reporting more than 20 000 moves.

These observations combined result in an extremely large search space, an upper estimate could be to $4^{6000} = 2^{12,000}$ for a typical game. Compare this to chess, with a commonly accepted figure of $35^{80} = 2^{400}$. From this reasoning we can conclude that exhaustive search for the optimal solution is intractable.

Fortunately many of the states in this tree are unreachable due to invalid move sequences, additionally many different sequences of moves result in similar terminal board states due to the simple rules of movement and combination. From this reasoning we approached the problem as a sequential decision problem: treat each move as a separate optimisation question, what is the best single move I can make from the current board state. A policy based approach is possible, however the low branching factor and fast board calculation make state space searching approaches more attractive. Our board simulation and scoring implementation can produce 6 million boards per second, allowing us to search exhaustively to a depth of approximately 10 (0.2seconds per move).

The score is exponentially related to the number of pieces that have been inserted into the board. This suggests that optimising for game length will be roughly equivalent to optimising for score directly. We model the board as a node in the state space graph, with edges corresponding to the four available moves.

Strategy

Our investigation strategy was to implement a simple depth first search, and investigate the performance of various heuristics. We then implemented more complex state space searching algorithms such as priority first search. Finally we combined multiple search styles and heuristics into a strategy based on our observations of the performance of our searches.

Heuristics

We implemented 5 functions to score boards. These heuristics were not admissable or monotonic as studied in our course, they were simply evaluations of the quality of a board with respect to some metric. Each heuristic returns a floating point value between 0 and 1.0, 1.0 being the best score. The detail of each is addressed in appendix H.

1. BoardScore, calculates the score of the board (not between 0 and 1.0).
2. BoardScoreNormalised, BoardScore normalised by the maximum possible score of that board.
3. Empties, counts the number of empty cells on the board.
4. AdjacentMatches, count of adjacent cells that can be combined.
5. AdjacentRuns, see appendix.

Additionally we used one combination technique with our heuristics, a linear weighted sum.

Search Space Algorithm

Search Algorithms investigated

The appendix details the operation of each search algorithm we implemented.

1. DLDFS, depth limited depth first search.
2. Pruner, a priority first search that explores N nodes at depth D.
3. Jogger, using DLDFS, but returning the best N move sequence (instead of 1 move).

We also implemented WideSearch, which used DLDFS, but chooses the path based on the average of the best N terminal nodes instead of best single node. This was an attempt to mitigate the horizon effect by maintaining options, however the performance was not good and it was not pursued further.

Testing

Our testing environment consisted of 5 randomly generated boards with 16 thousand moves available. These were generated with the script provided by the lecturer, modified to limit the number of excess 1s or 2s. Games tend to end with the free spaces on the board totally filled with either all 1s or all 2s, this modification therefore extends the length of games. We averaged the performance of the testing on the 5 boards. A larger sample of boards would have been ideal, however the run time of our tests was up to 10 minutes per game, making more thorough testing infeasible. Please note the java JIT and hotspot compilers caused some anomalies in our data.

DLDFS

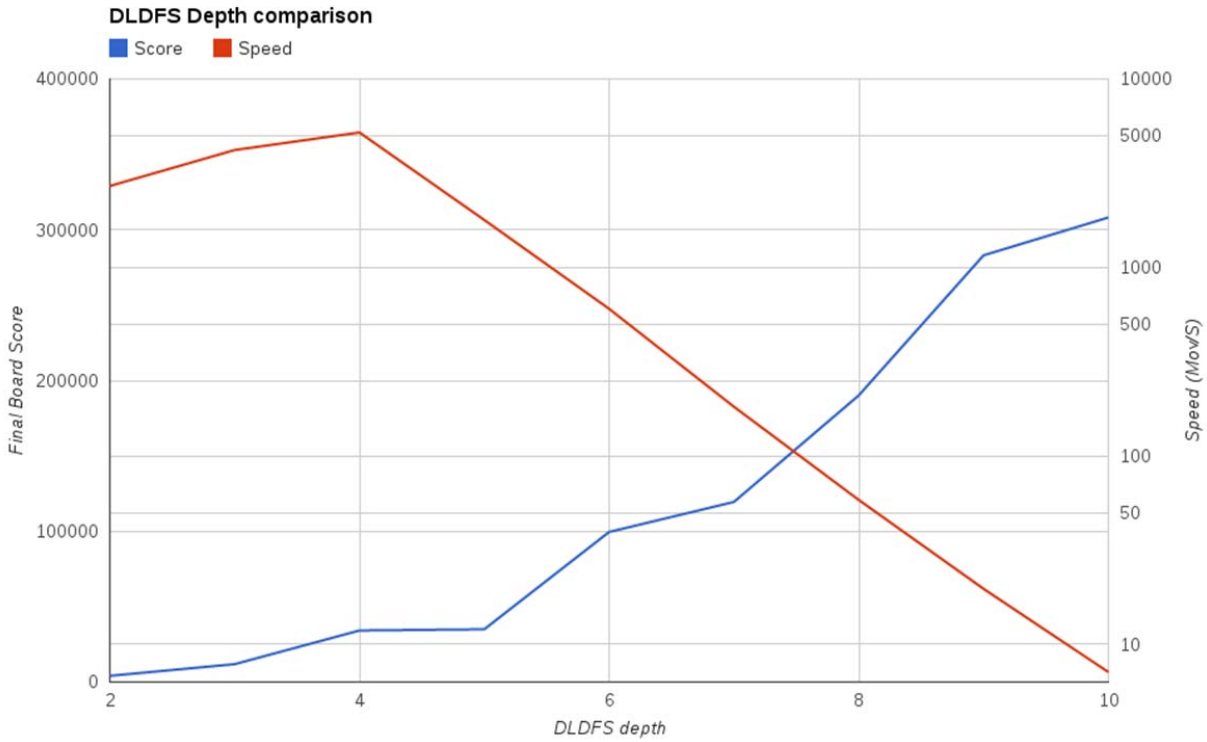

Figure 1: Increasing the depth limit of the DFS. Graphed is the speed and final board score (average).

Figure 1 shows the effect of increasing the depth of look ahead. From this graph it is clear that look ahead or depth of search is extremely beneficial to the performance of the search. The score continues to increase up to the point where moves are too slow for the requirements. Also shown is the exponential relationship of time with respect to depth. The anomaly of time at the low end of the graph is due to the java JIT and Hotspot compilers, which optimise the code as it runs.

This graph was the first we produced during our investigation; it demonstrated to us that obtaining greater depth look ahead would be the most profitable avenue for investigation.

Pruner

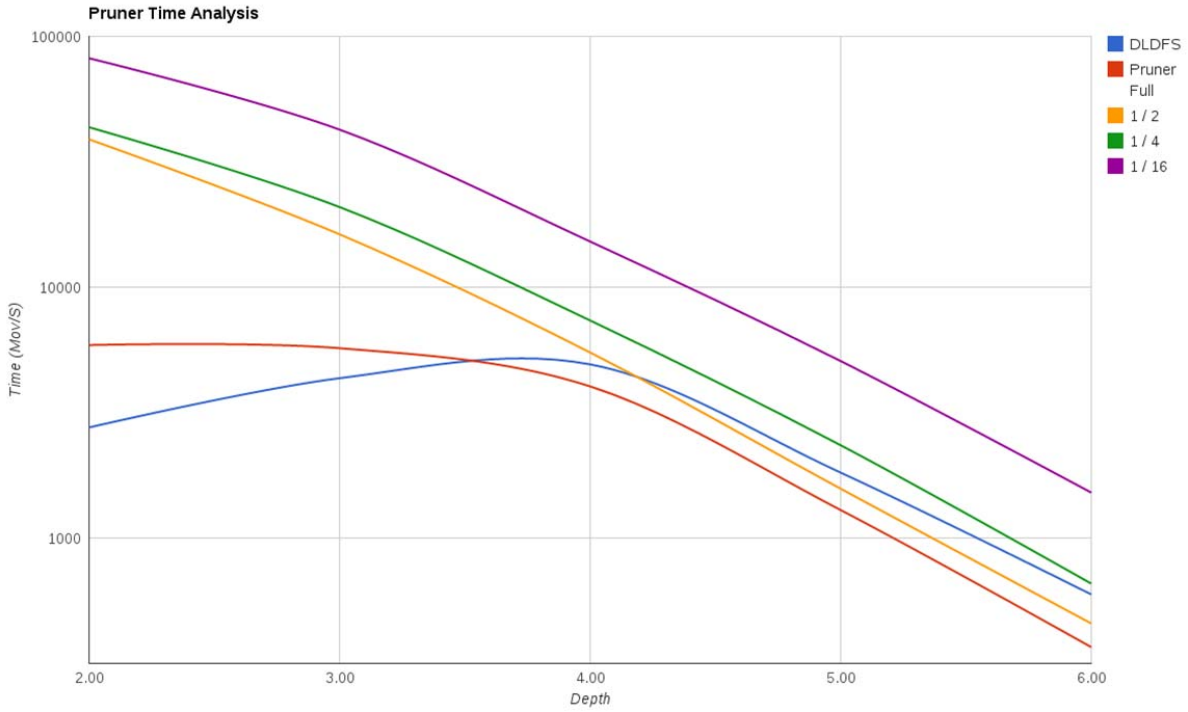

Figure 2: Time analysis of Pruner. Low values of N increase speed. The number labels are a fraction of total node to explore, eg 1/16 at depth 4 gives an N of (4^4)/16.

We performed a time analysis on our Pruner searching algorithm Figure 2. The different traces show the time performance as a function of depth, while varying the number of nodes visited (N). The most basic comparison is DLDFS vs Pruner full; pruner is slower at evaluating every node in the graph. This is because of the overhead of extra memory usage, priority queue maintenance, and additional heuristic evaluation. However by reducing the N to visit only a quarter of all nodes Pruner outperforms DLDFS. The N can be further reduced to increase speed even more. We next investigated how the effect of N could be used for trade-offs in speed/performance, or for increasing the depth of search.

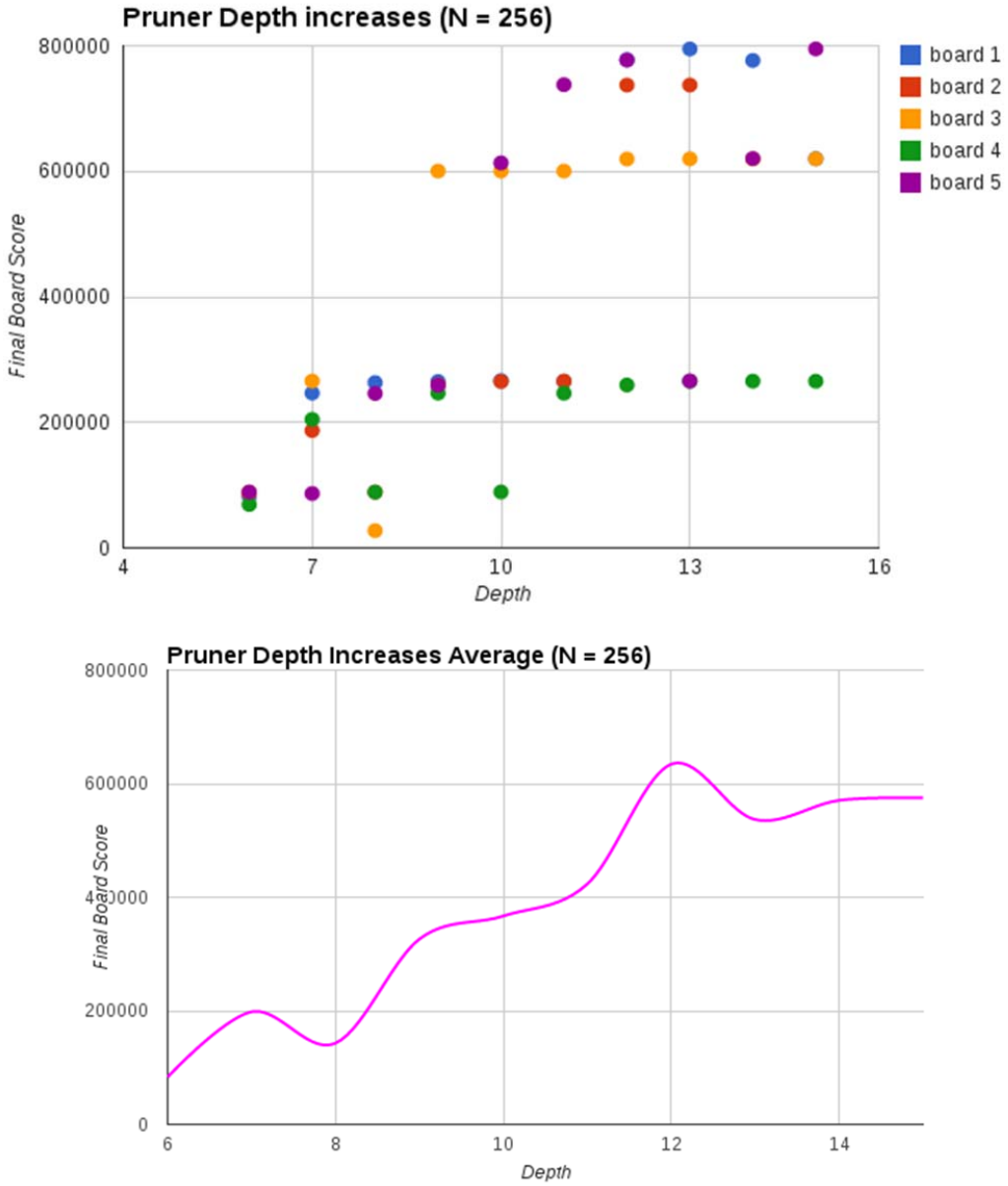

Figure 3: Sweeping the depth look ahead for Pruner with constant N = 256. Above is the data for each board, below the geometric mean. Note the step function style of the above graph.

Figure 3 analyses the effect of increasing depth of pruner search. In the averaged graph, there is a clear benefit to increasing the depth of the search. However inspection of the data for individual boards shows an interesting effect. The score tends to move in jumps from one plateau to another. This is clearest for the yellow dataset with the jump in score at depth 8-9. We believe this is due to the search overcoming a difficult point in the board input sequence.

This analysis lead us to the conclusion that focussing search efforts on overcoming these difficult points could lead to a good trade-off between computation time and performance. Two approaches to address

this were identified, creating heuristics to attempt to improve board management, or using variable strength search algorithms to focus on important sections. The second approach was followed.

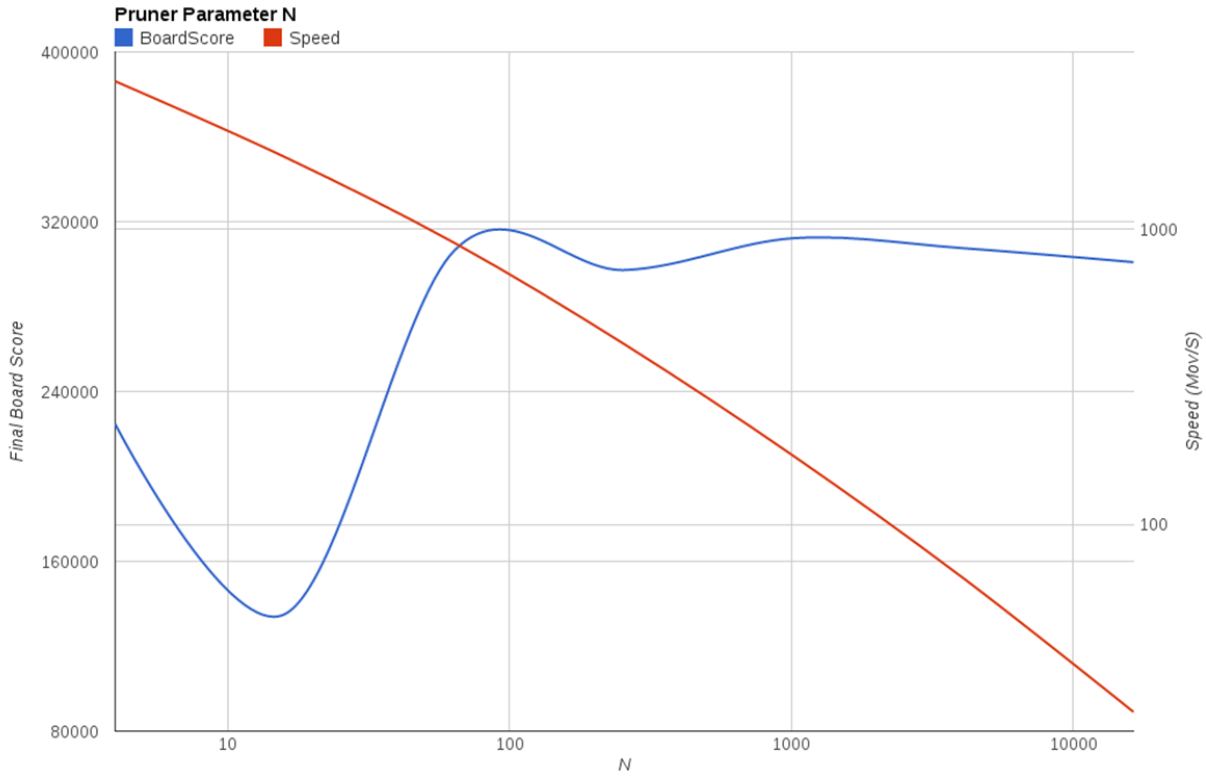

Figure 4: Pruner effect of varying the N for constant depth 9.

The final analysis of the pruner algorithm investigated the optimum value for parameter N (Figure 4). The value 256 was selected as higher values do not seem to improve score performance, while lower values did hurt performance. Note the logarithmic axes. Following this analysis of Pruner we implemented Jogger, as another potential 'fast' search algorithm.

Jogger

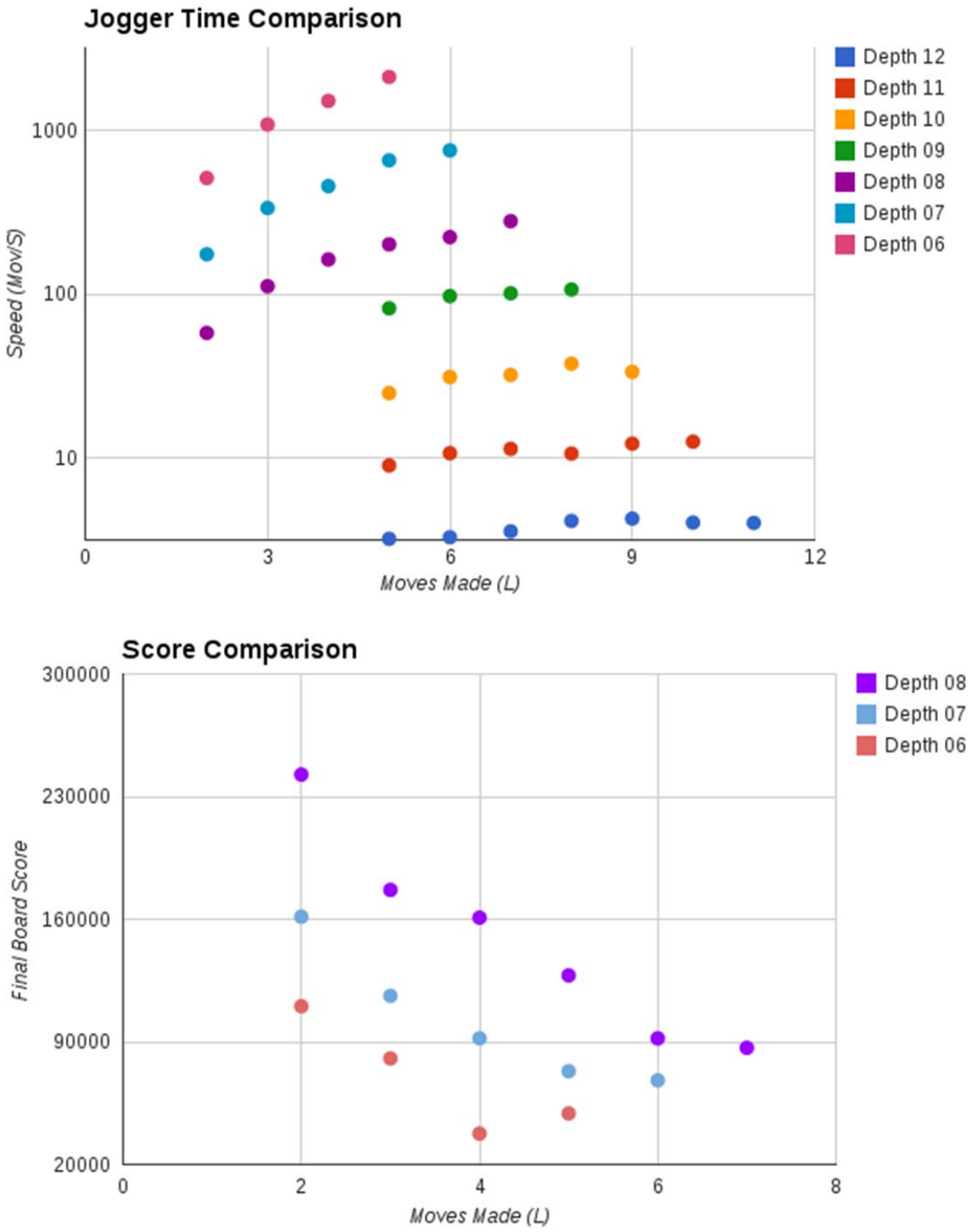

Figure 5: Comparison of speed (above) and score(below) for varying depth and L using Jogger.

Jogger was implemented as a extremely fast algorithm. The idea is to perform multiple board movements based on a single search in the state space. Figure 5 shows the speed for various depths and L values. Notice the flattening of the curve for greater depths. For this reason we choose a depth of less

than 8. Figure [] shows the score performance. This graph again shows the benefit of depth, in addition the fewer moves made per search the better the score, however this increases the runtime. A trade-off must be chosen when using Jogger and Pruner to speed up searching.

Heuristics

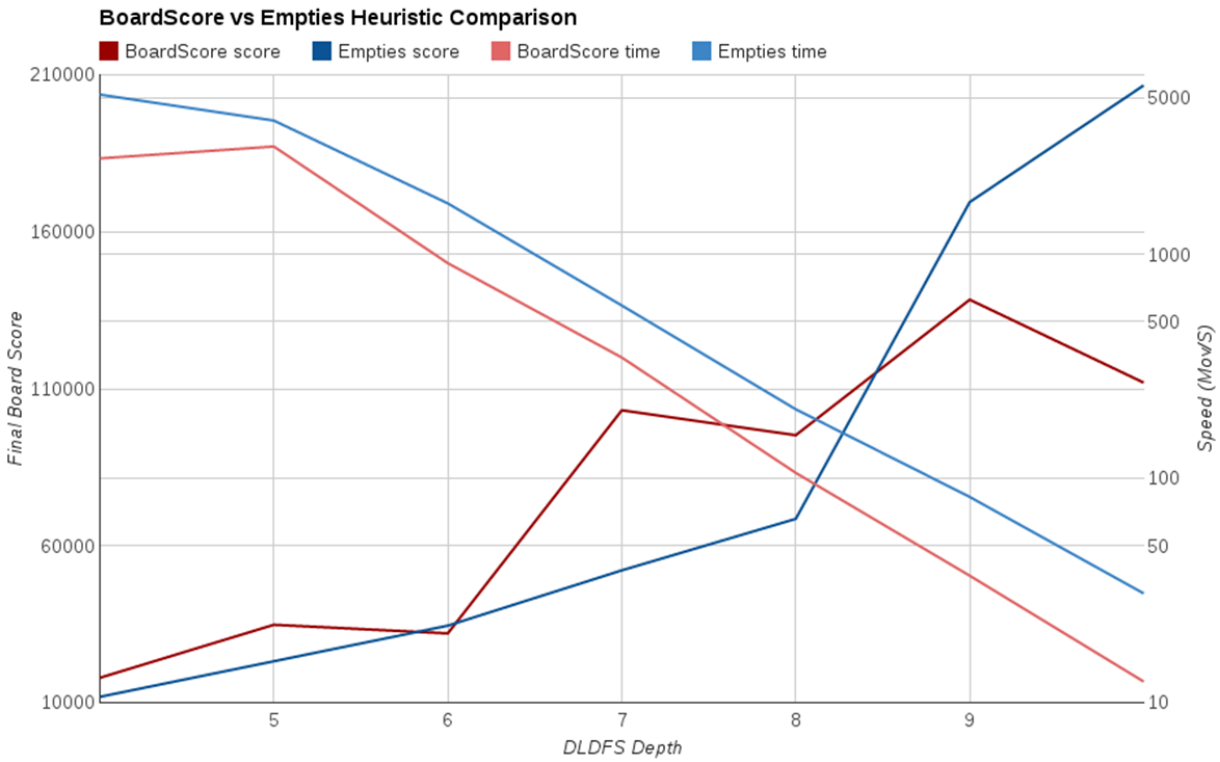

Figure 6: Single Heuristic performance comparison.

We compared the performance of the heuristics Empties and BoardScore Figure 6. The graph illustrates the noisy and erratic data we observed in our investigation, even after averaging the performance over 5 boards. Especially of note is the dip in performance for BoardScore at depth 10. The conclusion from this graph was that the final score performance of Empties and BoardScore were comparable, although empties has a substantially faster runtime. This illustrates our conjecture that optimising for staying live increases the score, empties makes no attempt to increase the score and yet performs similarly to BoardScore.

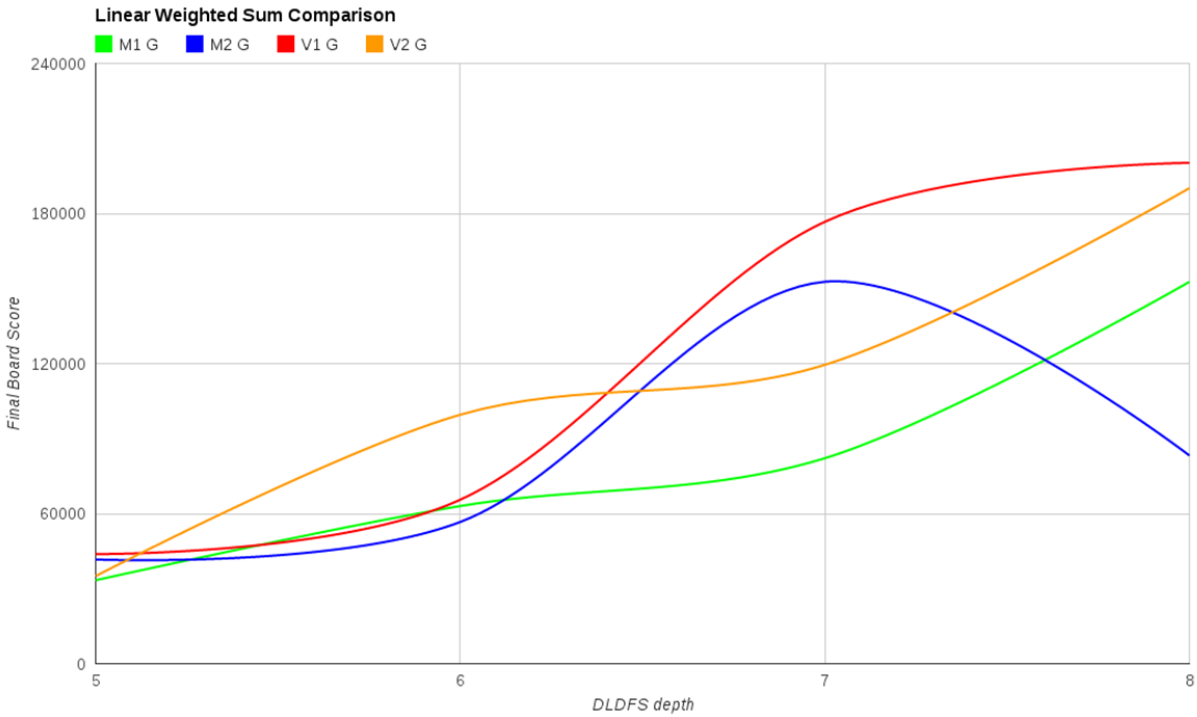

Figure 7: Linear weighted sum comparison for 4 chosen variations of weightings.

We tested several different weightings for our linear weighted sum (Figure 7). We also investigated the effect of sweeping each of the individual parameters, the results were not easy to interpret. The effect of changing the weightings was often unpredictable, and quite sensitive to the board it was tested on. One detail was clear however, the combination of heuristics did perform better than any of them in isolation. We chose 4 of the best performing weightings to plot in Figure 7 and selected the best of those to use in all further linear weighted sum usage.

A more in depth analysis of weightings could have been the topic of this report, for example a training system to test different values. However we learnt from other students that this approach did not work well for the threes game. Additionally our heuristic for AdjacentRuns was found to give no performance benefit, possibly due to the simplistic way the concept was coded. More complex heuristics or agents to train the parameters were one option for how to proceed from here. We instead opted to attempt to come up with a high level strategy.

Search Strategies

We explored higher level search strategies after observing the behaviour of our searching algorithms on our test data. In general we noted that even when increasing the computation time, such as by increasing the exploration depth, scores did not always improve. Scores tended to plateau at certain values due to difficult points in the game input sequence, and then jump up to new plateaus as the depth limit became sufficient to navigate these difficult points. Further, we noted scores occasionally decreasing as more depth look ahead was added, this was difficult to analyse, we propose that the horizon effect could explain these anomalies.

From the observations made we created several higher level strategies. In general they used fast, shallow state space searches by default, and swap to a slower search when some trigger is reached. In this way the costly slow search is amortised over the cheap faster search.

1. Strategy - guess the locations of 'hard' sequences, swap to hard searches in these areas
2. StrategyBT - continue with easy searching until game ends, then backtrack and use a harder search to try to escape death. After escaping death use easy search again.

StrategyBT was a simpler design, and outperformed Strategy, the analysis of Strategy is not shown.

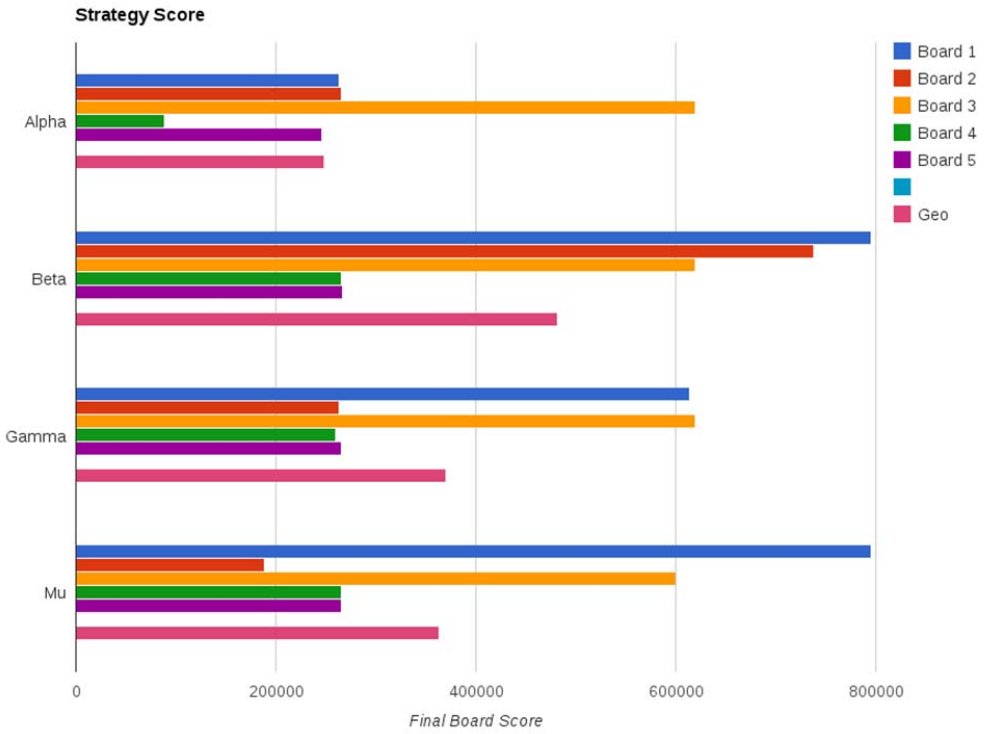

Figure 8: Comparison of scores for the 4 configurations of StrategyBT tested Individual board scores and the mean are shown.

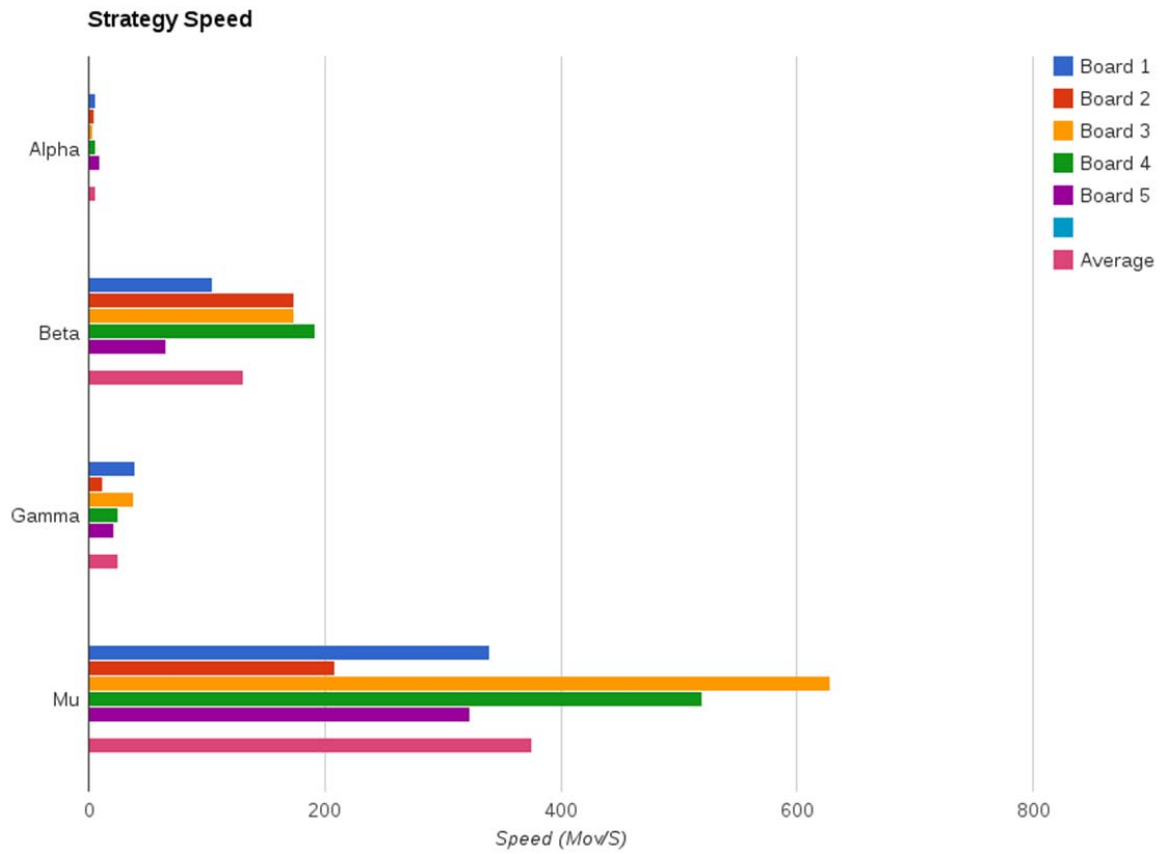

Figure 9: Speed of each of the four configurations of StrategyBT tested.

Name	Hard Searcher	Easy Searcher
Alpha	DLDFS D=12 (LWS)	Jogger, D=6, L=4 (BoardScore)
Beta	Pruner D=15, N=1024 (LWS)	Jogger, D=6, L=4 (BoardScore)
Gamma	DLDFS D=12 (LWS)	Pruner, D=9, N=256 (Empties)
Mu	Pruner D=15, N=1024 (LWS)	Pruner, D=9, N=256 (Empties)

Figure 8, Figure 9, show the performance for 4 different configurations of the backtrack (StrategyBT) searching strategy. The configurations Beta Gamma and Mu have good performance for score, with beta winning overall due to a high score on board 2. Boards Beta and Mu also have excellent speed, with Mu

almost 400 times faster than Alpha. The observation is that Pruner outperforms DLDFS in score and speed for the hard searching, for easy searching both Jogger and Pruner perform well.

Combining

The final strategy is to use the strategies Beta and Mu, which are both extremely fast, and select the strategy that produces the best final board score. This approach helps to mitigate the somewhat random nature of the score of a particular strategy on different boards. This approach could be extended to attempting different heuristics and searching strategies.

Conclusion

We finished with a very fast and still high performing algorithm, which still has a lot of room for further improvement. There are many parameters that can be tweaked to alter the performance of our results: the heuristic parameters, easy and hard search parameters, and the backtrack amounts. These could be trained by for example a swarm optimiser or genetic algorithm.

The design of our backtrack searcher was an all or nothing style, an extension of this would be to use gradual values for the backtracking and thoroughness of search. This would allow gradually ramping up the search power.

We found the most challenging aspect of the threes game was the extremely noisy score performance. We theorise that this is due to the horizon effect.

We found our use of Java and object oriented principles extremely beneficial in this project; our algorithms use inheritance and polymorphism extensively. This allows any heuristic or search strategy to be dropped in and used with minimal effort.