COMP30024 Project: Part B

Natasha Chiorsac, Vincent Luu

Discussion

Search Algorithm

Our team used minimax as the search algorithm. We chose this algorithm due to its potential for completeness and optimally. Having said this, due to the time constraint, the implemented agent uses a depth cutoff and is therefore not perfectly optimal. Other reasons why we chose minimax search include its relatively low space complexity and its potential for optimization. These theoretical benefits of minimax was back by experimental data as discussed in the next section.

Additionally, several modifications to the standard algorithm were implemented in an attempt to increase the speed and efficiency of the algorithm without compromising optimality. These optimisations will also be discussed later on in the report.

Experimentation

During the process of experimentation to find the best game-playing approach, we created several agents, including a simple random agent, a greedy agent following a utility calculation and a minimax agent loosely based upon the pseudo-code from the Minimax Wikipedia article[1]. We also implemented a very basic MCTS agent adapted from "ai-boson"'s article [2]. In order to test the performance of the different agents, we played them against each other 20 times each (switching which agent played RED/BLUE after half the games in order to take into account any advantages that may be had by playing as one colour or the other) and compared their win/loss ratio statistics.

This was done using the script that we created to automate the playout of many games between two specified agents and calculate statistics regarding how well they played. This data was then used to compare our different agent implementations (in particular our main candidates minimax and MCTS) and inform our selection of the best approach. Note that due to the nature of the implementations, minimax is the only agent without any elements of randomness. As such, every game tested was unique apart from minimax vs minimax.

	Random		**Greedy**		**MCTS**		**Minimax**	
	As RED	As BLUE	As RED	As BLUE	As RED	As BLUE	As RED	As BLUE
Random	6/2/2	2/6/2	10/0/0	10/0/0	6/3/1	8/2/0	10/0/0	10/0/0
Greedy	0/10/0	0/10/0	6/4/0	4/6/0	0/10/0	0/10/0	10/0/0	10/0/0
MCTS	2/8/0	3/6/1	10/0/0	10/0/0	0/6/4	6/0/4	10/0/0	10/0/0
Minimax	0/10/0	0/10/0	0/10/0	0/10/0	0/10/0	0/10/0	1/0/0	0/1/0

Table 1: Playout statistics between various game-playing agents. Each cell contains the Win/Loss/Draw ratio for the agent (top row) in games against its opponent (left column)

In this manner it was determined that the minimax agent performed the best. As can be seen from the above figure containing Win/Loss/Draw statistics for playouts between our different agents, the minimax agent consistently beat the random and greedy agent, in contrast

to the MCTS agent which lost to the greedy agent and did not even consistently beat the random agent. Most importantly, when directly playing minimax and MCTS against each other, minimax consistently wins. In addition, when performing these tests minimax agent was noted to be faster than MCTS even without optimization.

In addition to the experimental data, the playouts of many games were analysed, allowing our team to glean more insight into benefits and drawbacks of the various agents. Minimax played extremely well, being able to setup and avoid shallow traps even without optimization. It also makes moves that works towards a win earlier rather than a win at 343 moves. Minimax's ability to look ahead gives it a kind of foresight that MCTS did not have. While MCTS theoretically knows the best move with enough training, it still falls into shallow traps and fails to force wins due to the problems of credit assignment and delayed reinforcement.

Considering all this, we decided to go with the minimax agent with added modifications to further enhance its performance.

Evaluation Function

The evaluation function that our algorithm employed considers the number of cells (i.e stacks) of each player as well as their total power. A higher number of cells of the agent's own color (ally cells) and a lower number of cells of the opponent's color (enemy cells) results in a better evaluation score. Likewise, a higher total power of ally cells (ally power) and a lower total of power of enemy cells (enemy power) also results in a better evaluation score. Strategically, this makes sense since more ally cells or a higher ally power puts the agent in a better position. This position would have more potential to capture enemy cells and a lower likelihood of losing. Furthermore, if 343 moves is reached, the player with the higher total power is declared the winner.

During testing it was found that the instantaneous power was more strategically important than the instantaneous number of cells. As such, the weighting between considering power and considering number of cells is not the same. Specifically, the following formula was used as the evaluation function: $A * (powerAllies/powerEnemies) + B * (numAllies/numEnemies)$.

To determine the best values of constants A and B, our team utilised supervised machine learning. We ran various agents with various weightings against each other in an automated suite of tests, using a regression learning methodology to estimate an optimal weight. We used a linear regression model and clamped B to 1 in order to limit variables. In this way, it was determined that the best weighting was approximately 6:1, meaning that the evaluation function would be: $6 * (powerAllies/powerEnemies) + (numAllies/numEnemies)$. This ratio is significant because in Infexion, the max power of a stack is 6.

The findings from the machine learning correlates to the theoretical observation that stacks with a higher power is somewhat more strategically important than more stacks. By taking this into account, the algorithm avoids what our team has dubbed "walking" where cells are SPREAD around for no strategic reason. In addition, considering the instantaneous changes in power at each depth allows minimax to set up shallow traps for the opponent.

Although the relative positioning of red and blue cells is also very strategically important, our evaluation function does not explicitly take positioning into account. However, due to the nature of minimax, the search algorithm still determines a good position and avoids falling victim to traps by looking ahead a certain number of moves. Additionally, the current evaluation function determines the viability of the game state extremely quickly, allowing for a greater depth minimax search. This increase in search depth was found to be much more viable than a slightly more detailed (but more expensive) evaluation function.

Optimisations

In order to improve the performance of minimax search, our team implemented various optimisations from the lectures and from Lars Wächter's article "Improving Minimax performance" [3]. In this report, a node means a generated game state and its corresponding data (minimaxValue, parentAction etc).

Our program employs a depth cutoff of 4 in order to adhere to the 180 second time limit at the cost of optimality. While this makes minimax search sub-optimal, a 4 move look ahead still allows the implemented agent to avoid traps and set up some of its own.

Alpha-beta pruning was also used to reduce the number of nodes expanded by the algorithm. We used John Fishburn's "fail-soft alpha-beta" variant based on the pseudo-code in the Alpha–Beta Pruning Wikipedia article [4]. This variant initialises the current value of the node as $\pm\infty$ when minimising and maximising, respectively. "Fail-soft" should theoretically explore slightly fewer nodes than the standard "fail-hard", where the value is initialised to α or β.

To improve the benefit of alpha-beta pruning, our implementation also pre-sorts moves. This involves sorting moves based on the evaluation score. The moves are sorted in descending order when maximising and ascending order when minimising. By doing this before expansion, alpha-beta pruning does the pruning earlier, reducing the total number of nodes explored. With "perfect ordering", alpha-beta search would have a time complexity of $O(b^{m/2})$, where b is the branching factor and m is the maximum depth. As a point of comparison, standard minimax has a time complexity of $O(b^m)$.

In addition, the implemented generation and sorting of moves makes it so that SPAWNing is preferred over SPREADing when they have the same evaluation score. This has a twofold benefit. Firstly, it controls the branching factor so that the algorithm will theoretically need to explore a fewer amount of nodes over the course of the game, thus increasing efficiency. SPAWNing instead of SPREADing reduces the branching factor because each SPAWN action increases the branching factor by 6, while each SPREAD action increased the branching factor by approximately 6 multiplied by that cell's power minus 1. The second benefit is that SPAWNing is often more strategically viable than SPREADing when the evaluation scores are the same. This is because SPREADing can result in falling into traps set up by the opponent whereas SPAWNing can can result in our agent setting up its own traps. These situations can be identified by the minimax algorithm's "foresight" which it then uses to make an informed move.

We also changed the standard algorithm to check for instant termination in "forced moves". This is a simple optimisation that allows minimise and maximise functions to return early, separate to alpha-beta pruning. When maximising, if a node involving a game winning move is found, that move is guaranteed to be the highest valued node, so there is no need to keep searching that depth level. Likewise, if a game losing move is found while minimising, that move is guaranteed to be the lowest value node. This simple modification has a proportionally small effect, only reducing the search space by fractions of a percent, but helps to guarantee optimality.

To ascertain the effectiveness of each of the aforementioned optimisations, we played the minimax agent against the greedy agent, tracking the number of nodes generated. The greedy agent was set with a seed of 30024 and the game was cutoff at turn 30 for blue and turn 29 for red. This was done in order to keep variables consistent between tests and gain more meaningful comparative data. The data from the alpha-beta related modifications is shown in the figure below.

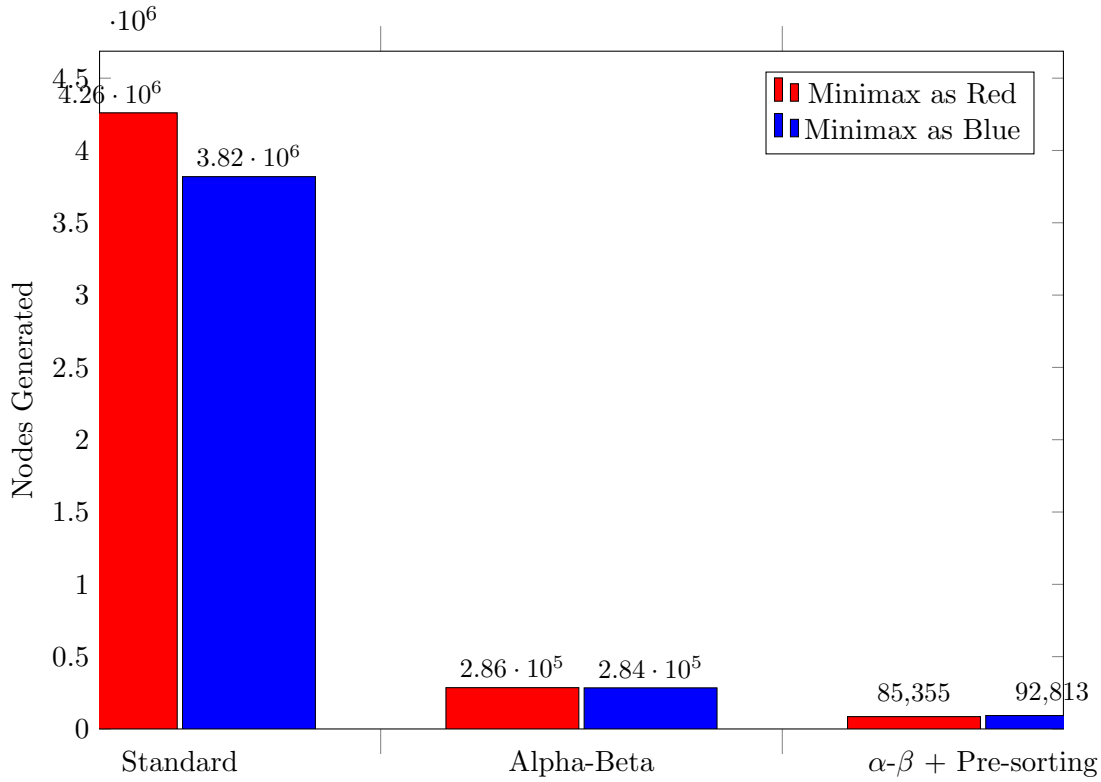

Figure 1: Number of nodes generated by minimax agents with different optimisations

As evident from the figure above, alpha-beta pruning has a great effect in improving the efficiency of minimax. In this test, alpha-beta pruning reduces the number of nodes generated by a factor of approximately 15. Additionally, pre-sorting provided a smaller but still significant reduction in nodes. This reduction in generate nodes is proportional to a reduction in computation time and memory usage.

Since instant termination only has a small effect at towards the end of games, it's data is not shown but can be observed in full playouts. On the other hand, preferring SPAWN other SPREAD changed the playout of games too much to compare to the other optimisations. However, independent testing showed that this preference reduced the number of turns needed to win and also reduced the number of nodes generated, inline with the theory outlined above.

During testing, other optimisations were trialled but ultimately dropped for various reasons. One such optimisation was a quiescence search to reduce the impact of the "horizon effect" [5]. However, due to the nature of the evaluation function and the instability of Infexion rounds, a consistent way to determine "quiet" nodes could not be found. As such quiescence search had little to no positive effect and would actually sometimes cause the best course of action to be overlooked. Our team also tested out the usage of bitboards as a way to represent game states. However, despite the advantage in lower space complexity and efficiency they can be compared, there was a few drawbacks. For example, generating and updating "children" states was less efficient and straight forward than using separate dictionaries as in our current design.

Overall, the strategic evaluation function combined with the minimax agent makes for a strong adversarial game playing agent. Coupled with the many significant theoretical and algorithmic optimisations discussed prior, this agent comfortably beats random, greedy and other adversarial agents with a high degree of time and space efficiency.

References

[1] https://en.wikipedia.org/wiki/Minimax

[2] https://ai-boson.github.io/mcts/

[3] https://levelup.gitconnected.com/improving-minimax-performance-fc82bc337dfd

[4] https://en.wikipedia.org/wiki/Alpha-beta_pruning

[5] http://satirist.org/learn-game/methods/search/quiesce.html