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.

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 be discussed later on in the report.

Evaluation Function

The evaluation function that our algorithm employed considers the number of cells 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.

Although the relative positioning of red and blue cells is very strategically important, our evaluation function does not explicitly take positioning into account. Due to the nature of minimax, the search algorithm still determines a good position and avoid falling victim to shallow traps by looking ahead a certain number of moves. However, the agent may still succumb to more elaborate and drawn out traps. Still, the faster computation of the simpler evaluation function makes this a reasonable trade-off.

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 psuedocode 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, "run_games.py", 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.

	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	0/10/0	10/0/0

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 significantly faster than MCTS. Considering this data we decided to go with the minimax agent with added modifications to further improve its performance, as described in the next section.

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].

Our program employs a depth cutoff of 4 in order to adhere to the 180 second time limit at the cost of optimality.

In addition, alpha-beta pruning was 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 or maximising, respectively. ”Fail-soft” should theoretically explore slightly fewer nodes than the standard ”fail-hard”, where the value is initialised to α or β.

In addition to these improvements, our program also pre-sorts moves and checks for instant termination.

Pre-sorting moves 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 $O(b^m)$.

Checking for instant termination is a simple optimisation that allows minimise or maximise functions to return early. 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. Likewise, if a game losing move is found while minimising, that move is guaranteed to be the lowest value node. This simple modification has an proportionally small affect, only reducing the search space by fractions of a percent.

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. Since instant termination only has a small effect at towards the end of games, it's data is not shown. This data from the other modifications is shown in the figure below.

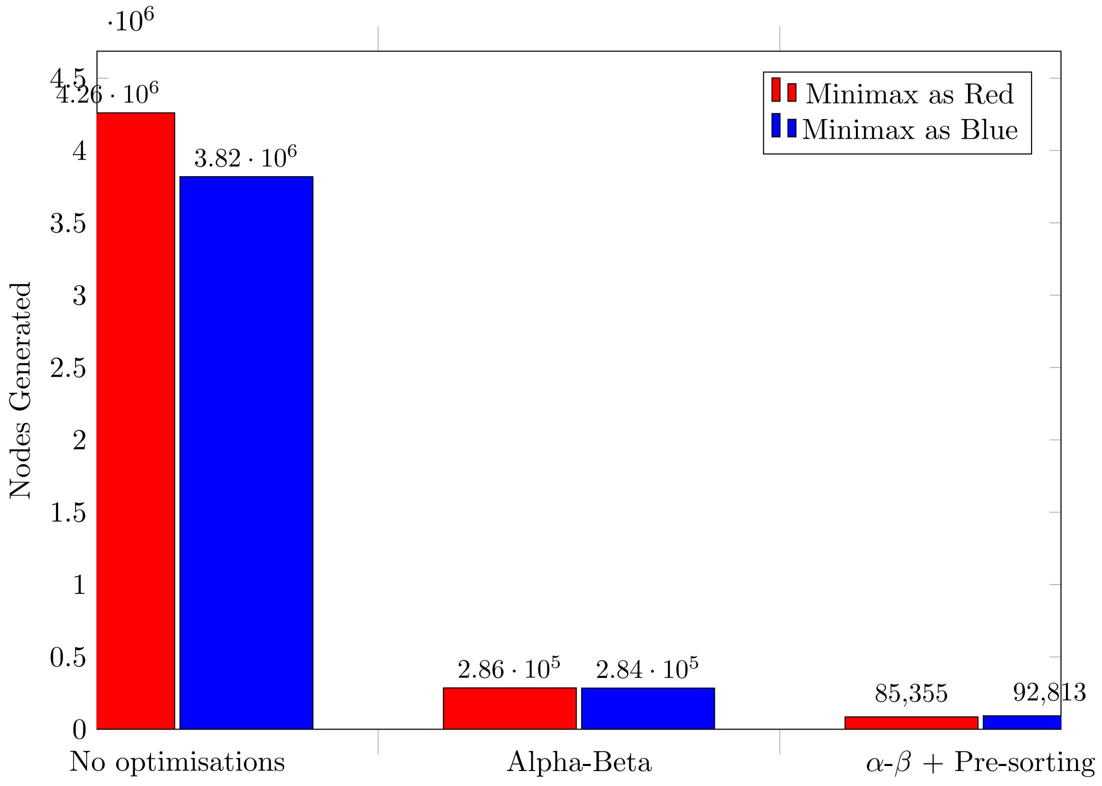

Figure 1: Number of nodes generated by minimax agents with different optimisations

During testing, other optimisations were trialled but ultimately dropped for various reasons. For example, the branching factor could be controlled by choosing SPAWNs over SPREADs when they have the same evaluation score. This would theoretically reduce the branching factor and by extension the total number of nodes explored. However, testing showed that, in practise, SPREADing is usually the better move even when the evaluation score is the same. In addition, we tried to implement 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, quiescence search had little to no positive effect.

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