COMP30024 Part B Report

Yifan Yang & YuanYuan Xiang

Approach:

General Agent Approach

After experimenting with several algorithms, we opted for the minimax algorithm on a high level. We made this decision primarily due to the speed and effectiveness of the minimax algorithm when compared to other algorithms. Given the time constraint of 180 seconds per game, the agent can only afford to take around 1-2 seconds for each move (assuming the game plays to turn 343). If we expect the agent to be able to win in a low number of turns, it may have slightly more time per turn, but still only around 5 seconds. This limited time frame makes approaches like Monte Carlo Tree Search (MCTS) less effective.

In fact, we did experiment with an MCTS agent, however it did not perform as well as the minimax algorithm in our tests. The MCTS agent would only be able to simulate around 1000 games (using random moves) in a 5 second time frame, which is insufficient for a game like Inflexion, where the branching factor is potentially over a hundred. Moreover, even if during the simulation phase of MCTS, we use a different move strategy such as greedy moves, the time required for each simulation would significantly increase due to the need of an evaluation function that looks at the entire board. Therefore, minimax seems to be the better choice given the strict time restraints.

After testing the minimax agent with various depths, we settled with depth 3, which allows it to look 3 moves ahead. At this depth, the agent typically takes between 2 to 6 seconds to choose a move to make. We found that this depth strikes a good balance between the time required to make a move and the effectiveness of the move. Of course, increasing the depth would likely result in better moves, but even increasing the depth by 1 would exponentially increase the time required to find the best move according to our evaluation function, meaning that we wouldn't meet the time constraint. On the other hand, if we set the agent to depth 2, the agent is outperformed by the same agent set at depth 3, even though depth 2 only requires around 1 second to make a move. Overall, after considering the tradeoff between time and effectiveness, setting the minimax agent to a depth of 3 appears to strike the best balance for our project, as at the depth, it also allowed for some flexibility to add more advanced evaluation components and game strategy implementation whilst meeting the time restraint.

Our initial utility value was based solely on the difference in total power of the pieces on the board. It proved effective against basic agents like random and greedy agents, as the total power difference on the board fundamentally aligns with the goal of the game, being a substantial metric indicating a player's control over the board. However it failed to perform well against more sophisticated agents, as such a simplistic metric does not take into account important factors such as board positions, strategy at different stages of the game, positionally safe moves and piece mobility. In addition to this, after testing out the effects of various components of our evaluation function, we came to the conclusion that a simple one factor evaluation such as power difference is vulnerable to certain board configurations where a player may have notably lower total power, however are in an advantageous position on the board, due to potential cell tradeability, as well as the mechanic where a lower power cell can consume a higher power cell.

Therefore, to improve the effectiveness of our minimax agent, we adopted a more sophisticated approach to calculating the utility value, and instead divided the game into four distinct stages, prioritising different evaluations for each stage. We considered optimal strategies to approach the game based on careful evaluation and trialling of the rules of the game, and examined through the different test matches our agents played, what the board often looked like at different general stages of the game. In each of these stages, the utility value is a weighted sum of several evaluations, including total_power_diff, num_cell_diff, connectivity_diff, and safety.

<u>Evaluation Factor Explanations (Factor names consistent with Code Variable Names)</u>

- **total_power_diff**: this is simply the difference in total power between the enemy cells and the player's cells
- **num_cell_diff**: the difference in number of enemy cells and the player's cells on the board
- **connectivity_diff**: the difference in connectivity between the enemy and the player. Connectivity is an important factor we considered for our evaluation factor, as a larger connected group of cells have a higher potential for defending against enemy attacks, as this increases a cell's 'tradeability'. Furthermore, as the game progresses, and larger areas of cell clusters form, the board tends to be more fragmented with different player's cell groupings, and at this point it becomes more challenging to build connected networks of cells, therefore existing ones have more value.
  Due to this approach, we decided to place emphasis on cell group connectivity, particularly in the early game, in order to gain an advantage going into the mid game. We calculated the number of connected cell groupings using the union find algorithm, which helps us determine the number of connected components (islands of cells of the same colour) in an efficient time complexity on average.
  Union Find Learning Material :
  https://www.geeksforgeeks.org/number-of-connected-components-of-a-graph-using-disjoint-set-union/
  https://www.thealgorists.com/Algo/UsefulDataStructures/UnionFind
- **safety**: This is an arbitrary measure that takes into account the number of friendly cells surrounding a given occupied cell, with the aim of ensuring that a cell is more protected if it is surrounded by cells of the same colour. For example, if a cell is surrounded by many cells of its own colour, it is considered to be more "safe", because friendly neighbouring cells are more likely to help take back the opponent after the cell is taken by the opponent.
  The factor of safety works most efficiently when paired with connectivity, as safety is a rough approximation of relative safety of your player's position on the board based on allies/enemies in the neighbouring vicinity. This leads to 'clusters' being formed when the agent prioritises connected components, rather than 'stick formations', which are

often weaker in terms of cell attack tradeability, which is why we attributed weights to both connectivity and safety.

<u>**Evaluation Weighting Strategy**</u>

Below are more detailed explanations of the weightings for how the utility value is calculated at different stages of the game, as well as brief explanations about the logic behind the weightings:

- <u>Opening game</u> (the first 15 moves of our player, that is, turn 1 to turn 30):
  *Utility value = 0.4*total_power_diff + 0.45*num_cell_diff + 0.1*connectivity_diff + 0.05*safety*
  In the opening of the game, we place a near-even weighting for total_power_diff and num_cell_diff, with a slight emphasis on num_cell_diff to occupy more space on the board. This places a higher importance on Spawn actions in the opening of the game to help the agent gain more control of the board. We also consider connectivity and safety, as scattered cells are not well protected, and can be easily taken by any decent opponent, as mentioned in detail above)
- <u>Early game</u> (move 16 to move 30 of our player, that is, turn 31 to turn 60)
  *Utility value = 0.5*total_power_diff + 0.35*num_cell_diff + 0.1*connectivity_diff + 0.05*safety*
  In the early game, we place a higher emphasis on the total power of our board than in the opening. After our agent has made 15 moves, we should have a fairly good control of the board and can shift focus to a more aggressive approach. However we still value connectivity and safety, as it's still early in the game where there isn't that many cells or power on the board, hence a slip up in defence at this stage can result in devastating vulnerability, resulting in a chain reaction of lost cells, which could be pivotal in any game, as seen from our test games.
- <u>Mid game</u> (move 31 to move 50 of our player, that is, turn 61 to turn 100)
  *Utility value = 0.7*total_power_diff + 0.3*num_cell_diff*
  In the mid game, our agent should focus more on the total power of our player on the board. We no longer consider connectivity or safety, because we should already have many connected components on the board with high safety and tradeability. Now is the time to be even more aggressive, and prioritise Spread actions in order to capture enemy cells and build on the strong position we hopefully developed in the early-mid game, and capitalise on this through reducing the enemy's power through taking more map control.
- <u>Late game</u> (move 51 and above, or turn 101 and above)
  *Utility value = 0.95*total_power_diff + 0.05*num_cell_diff*
  In the late game, we place a very high emphasis on the total power of our player on the board to hopefully close out the game quickly. Since we are using a depth of 3 for our minimax agent, each move takes around 2-6 seconds, if we are not winning hard at this stage, we risk losing on time, and switching to a final resort greedy heuristic move choice (mentioned after).
- These weightings are a combination of logical reasoning as well as empirical testing. Although they may appear somewhat arbitrary, they reflect our understanding of the game's dynamics, and our attempt to balance offence, defence, and time taken across

different stages of play. However, one factor to consider would be that board states are not always consistent with whether the game is in early, mid or late game based on the number of moves which have been exhausted. It was observed that in less common cases, in the mid game (or potentially at any stage of the game), a series of spread interactions between both players could result in multiple high-powered cells (built up from this series of interactions) would evaporate from the board due to hitting a power of 7, and the game board would be in a way, reset, to an 'earlier' more basic configuration. However, after trying to implement different strategies to take into account the dynamic stages of the game, the increased complexity of using these advanced dynamic strategies seemed to not have a notable increase in performance against most agents, however at the cost of valuable time to evaluate each move at different stages of the game. Therefore, we ultimately decided to stay with the more generic fixed-stage weighted evaluation function.

To avoid our agent losing due to time constraints, we have implemented a few defence mechanisms. One such mechanism is to switch to a "greedy" mode when we have a significant advantage. We define a significant advantage to be when the total_power_diff is larger than 12 in our favour, or when the num_cell_diff is larger than 10 in our favour. In greedy mode, our agent adopts a simple greedy algorithm that selects the move with the highest total_power_diff, without considering the more computationally expensive utility value calculations as mentioned previously. We found that the greedy algorithm is quite effective at closing out the game quickly when there is already a large advantage on the board, and it also saves time. If we do however start losing the advantage during the greedy mode, we switch back to the minimax mode, but this is unlikely to occur. Another mechanism we have used to avoid losing on time is to keep track of the total amount of time taken so far. If the total time taken reaches 160 seconds, we switch to the greedy mode. The downside of this is that the moves made may not be optimal, especially if we don't already have a big advantage. However this is better than losing on time. In some cases, we may be able to stall out the opponent until they lose on time, or we may already have an advantage on the board and can close out the game quickly.

To further speed up the search, we used alpha-beta pruning. This allowed us to discard unpromising branches early, thereby significantly reducing the number of nodes that need to be evaluated. On top of that, we optimised for alpha-beta pruning by placing moves with higher utility values (based on the stage of the game) at the front of the list of generated moves. This improved our search efficiency even more by allowing us to quickly find the more promising branches of the tree, and thereby allowing us to prune off a larger number of branches of the search tree. By combining these two techniques, we were able to reduce the search time of our agent and make it more competitive against stronger opponents. However, in order to incorporate all these strategies and more advanced evaluation factors, we needed to consider the worst-case scenarios for the time it took for each move. Hence, after generating the possible moves to evaluate in our 'best_move' function, we set a maximum limit for the time it took an agent to search for the best move in its turn. We initially simply set this time limit on the random list of all generated moves, however quickly realised that this would often cut off the 'best' evaluated moves from a certain state of the game, without the opportunity to even explore them, which is why we included move-ordering into the agent, as it not only allows for more effective alpha-beta pruning, but also allows for the most ‘promising’ moves to be explored first, even when there is a time limit imposed on the time it takes for an agent to select a ‘best move’.

<u>**Performance Evaluation**</u>

We evaluated our agent based on three key metrics: speed, winrate, and number of turns taken to win. We prioritised speed over win rate and aimed to improve the win rate without drastically affecting the speed. If an agent had a high win rate was too slow to finish the game, we considered it to be a poor performer. For instance, the MCTS agent with a high number of playthroughs or the minimax agent with a higher depth may have a high win rate, but they would not perform well under time constraints. We also considered the number of turns taken to win as a useful metric when comparing two similar versions of an agent. For example, if we made minor changes to the agent that didn't really impact speed or win rate, we could compare the average number of turns it took to win against a particular opponent. If the agent could win in fewer moves, it was considered to be an improvement.

Our final version of the minimax agent performed well in terms of speed, winrate, and turns taken to win. This is due to the numerous strategies outlined in the previous sections. In terms of speed, we measured the average time taken to make a move across a large number of games. On average, the agent takes around 2-4 seconds to make a move, with a maximum cap of 6 seconds. In terms of win rate and turns taken to win, our agent achieved an impressive 100% win rate against random agents, and over 95% win rate against greedy agents. It also typically wins against any simple agent in under 60 turns.

To test the agent against other adversarial search agents, we pitted it against various versions of minimax agents, some using simpler strategies and others with more complex strategies, as well as our MCTS agent. We found that our current version of the minimax agent outperformed the others in terms of win rate and turns taken to win, while also maintaining a reasonable speed. It’s worth noting that while our agent may occasionally lose to a more sophisticated opponent (such as a minimax agent with higher depth), its performance against simple agents, which are often used as benchmarks, was outstanding.

<u>**Supporting work**</u>
Apart from creating different versions of minimax and MCTS agents, as well as simple agents, to test against our agent, we also developed a script that automated the testing process for us. The script enabled us to input various opponent agents and automatically play a large number of games against each one. It records the wins and losses against each opponent and outputs the win rates for each agent in the end. This was a valuable tool for gathering statistics and verifying that our agent's performance was actually improving over time, rather than just winning a few games because of luck. Additionally, it saved us a significant amount of time that would have otherwise been spent manually running hundreds of games. With the automated testing script, we were able to leave the testing running overnight or while we were away from the keyboard, thereby significantly increasing our productivity.