# 1 INTRODUCTION

Project Part B of COMP30024 was to implement a game playing agent for the game Infexion. While this was fairly similar to Part A of the project, the difference primarily is that opponents can make moves and SPAWN actions are enabled. The two main algorithms considered with Monte Carlo Tree Search and Minimax.

The algorithm chosen by my team was Minimax, as we felt it had a more guaranteed success rate compared to Monte Carlo. Due to the large branching factor and low time and space availability, Monte Carlo may not have been able to perform enough simulations to get a viable playing agent. Due to the lack of confidence in the algorithm in this scenario, Minimax was chosen.

# 2 AGENT ALGORITHM

Minimax algorithm looks a certain level of optimal moves ahead to choose the best move based on what the opponents may pick. Despite the large branching factor, the algorithm was optimised to be able to make moves in under a second (See Optimisations for more information).

## 2.1 EVALUATION/UTILITY FUNCTION

For my algorithm, I have chosen the maximising player to be the colour RED. The utility function is just a measure of the power levels RED and BLUE have, described as follows:

$$U(state) = power(red) - power(blue)$$

## 2.2 DYNAMIC DEPTH

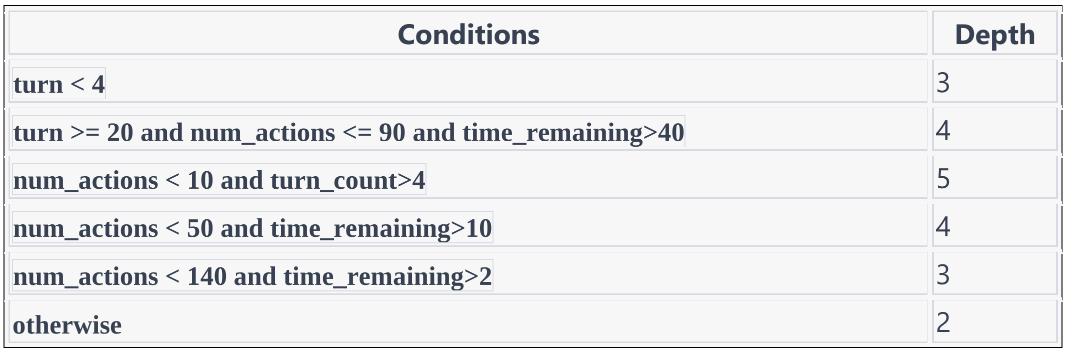
My agent can look up to 3 moves ahead really fast, but a depth of 4 can take a large amount of time if the number of actions (branching factor) is large. After thorough testing, it was found that *large* is when the number of actions is greater than 70, where it could take around 20 seconds to make a move. It also depends on the number of moves the opponent has but abstracting the complexity made it a simpler problem to deal with.

A few things to consider included the turn count, which essentially suggested whether we were in early-game (<20 moves), mid-game or late-game. Early-game is less important than mid-game, as most pieces do not do much capturing early on and the distribution is just based on spawning. Therefore, the depth searched was also dependent on the turn, because more the game progresses, more important the moves get.

Another factor that impacts depth search is the amount of time left, which I broke into a *Panic mode*, *Calm mode* and *just make a move* mode. Essentially, depending on the time left on the board, we would change the depth level. If there was sufficient time and it is mid-game/late-game, the agent would choose a depth of 4, as it is in *Calm mode*. But as the time decreases, the agent too would have less time to make a move and would search till lower depths. This idea was inspired by real strategy game players, who often take more time in the crucial part of the game when there is time remaining, but towards the end have to speed up and often make mistakes due to the lack of time. This brings the idea of dynamic time allocation which I implemented.

## 2.3 DYNAMIC TIME

Similar to dynamic deepening, dynamic time allocation per move is essentially giving my agent an upper bound on how much time they can spend making a certain move. The different modes explained above are functions of how much time is left on the board. Table 1 shows the different time allocations for different points in the game. During *just make a move* mode, the amount of time is a fraction of the time remaining. When the minmax function reaches it's time allocation for making a move, it is forced to return the best move it can find thus far without searching any further. This way it ensures that the player does not lose on time and makes a move even if it is a terrible one with the hopes that they may beat the opponent if the other agent runs out of time.



| Conditions | Depth |
|---|---|
| **turn < 4** | 3 |
| **turn >= 20 and num_actions <= 90 and time_remaining>40** | 4 |
| **num_actions < 10 and turn_count>4** | 5 |
| **num_actions < 50 and time_remaining>10** | 4 |
| **num_actions < 140 and time_remaining>2** | 3 |
| **otherwise** | 2 |

## 2.4 ITERATIVE DEEPENING SEARCH

Another idea explored was Iterative deepening, where it performs a BFS for every level that is explored and keeps going a level deeper till it runs out of allocated time for the move. The allocated time per move was determined as a function of the time remaining and the stage (early, middle, late) of the game the agent was at. The idea stemmed from dynamic time allocation such that it can return the best move of depth 3 before going into depth 4 just in case we run out of time halfway through depth 4.

The problem with this however was that the amount of time it took to accomplish depth 3 was already large, and then rerunning it for depth 4 was a waste of time just to look one move ahead. Treating this as another agent against dynamic depth and time would consistently make it lose on time. To make it's time allocation better, I chose to reduce the amount of time it gets per move, but this often cut short depth 4 search, and the extra time it took to do the depth 4 search was just a waste as that search did not contribute and the best move of depth 3 was returned. As each agent only gets 180 seconds in total, wasting time like this was not viable and thus iterative deepening was dropped.

# 3 OPTIMISATIONS

## 3.1 ALPHA-BETA PRUNING

To make the agent faster, alpha-beta pruning was used, which we will not explain as it has already been covered in the course content. However, alpha-beta pruning is dependent on how well the moves are sorted, which can change the complexity to up to $O(b^{\frac{d}{2}})$. As b is on average a large number (>50), half-ing *d* could be very beneficial and hence moves were pre-sorted.

## 3.2 PRE-SORTING MOVES

Moves were sorted when the set of possible actions are being returned. The two algorithms that were considered was Python's $list.sort()$ and $sorted()$. Generally, $list.sort()$ is 13% faster and uses less memory than $sorted()$ (Dahlitz, 2019), with the trade-off that it is not a stable algorithm. Luckily, the stability does not matter for the moves as the initial order did not mean anything. The key used to compare to moves was the greedy algorithm, where we apply the action and see the evaluation of the board from that action only. While this is not a perfect sort, it provides a good general idea of what a better move might be and helps the pruning as such. The complexity of pre-sorting moves is $O(b \ log \ b)$, where $b$ is the branching factor/ number of actions.

## 3.3 MOVE PRUNING

Another way to optimize is to reduce the total number of moves available for the agent. While this does not change much, it will reduce the number of moves by a bit which can cascade to be significant as the depth increases. For example, we would never want to SPAWN at a cell which neighbours the opponent, and hence those can be excluded. Similarly, we would prefer SPAWN actions next to our cells, and therefore those ones will be prioritized and returned.

## 3.4 BITBOARDS (NOT IMPLEMENTED)

BitBoards is another concept that was considered for our project. A large portion of time consumed is taken by the *deepcopy* of the Board, which is stored as a dictionary. The average case copy complexity of a dictionary us $O(n)$, which is large when we have $n = b^d$ copies of the board. A solution to this is a hashed board or storing it as a list, but due to the scope of the project this was not done. Moreover, using a different data structure would require restructuring of how possible moves are found and other functions, and thus not used. This could have been a very powerful extension, however, as it enables the use of memory tables as well to lookup predetermined board costs.

# 4 PERFORMANCE EVALUATION

In order to test how well our agent was faring, we creating multiple agents with slight variations and tested them locally to see which one was the best. Initially we were testing with a minmax agent without any optimisations, then created an alpha-beta pruned one and so on. After multiple rounds, it was found that RED AGENT played best with depth 3, while BLUE AGENT played best with a dynamic depth. In order to ensure it never ran overtime with blue agent, the MAX_TIME was updated, which describes the runtime per move.

After the depths were configured, the next changes done were to the evaluation function. We tried out multiple evaluation functions such as number of tiles left, weighted power, total power, etc. Trying out distance measures was also considered, but from Part A, it is known finding distance is a fairly costly calculation and give the time and space constraints, unless thoroughly examined and optimised, it would not be feasible.

# 5 References

Dahlitz, F. (2019). *List sort vs sorted list*. Retrieved from medium.: ttps://medium.com/@DahlitzF/list-sort-vs-sorted-list-aab92c00e17#:~:text=The%20previous%20investigations%20showed%20us%2C%20that%20list.sort%20is,use%20list.sort%2C%20you%20will%20lose%20your%20original%20list.

Foundation, P. S. (n.d.). *Time Complexity*. Retrieved from Python: https://wiki.python.org/moin/TimeComplexity

Wächter, L. (2021). *Improving Minimax performance*. Retrieved from gitconnected: https://levelup.gitconnected.com/improving-minimax-performance-fc82bc337dfd