Minimax

For our game-playing program, we chose to use minimax to select the next move. Our minimax algorithm is upgraded using Alpha Beta Pruning, Iterative Deepening and Move Ordering. Furthermore, we defined the red player to be the maximising player, thus a board with a positive heuristic score means red has an advantage.

Minimax vs Monte Carlo (MCTS)

In Infexion, certain moves can dramatically affect the board state, eg a red power 6 token spreading over four blue power 5 tokens. Hence, it is important to consider all possible moves when looking ahead.

A standard MCTS uses randomness to simulate a playout, which might miss certain critical moves, thus returning a sub-optimal next move. Minimax, on the other hand, attempts to explore all possible moves up to a certain depth. Assuming our heuristic is sophisticated enough, minimax should have a lower chance of missing critical moves and return a better next move. Hence, we chose Minimax.

Time & Space Complexity

- **Standard minimax:** Time and space complexity is O(b^d), as we would need to exhaustively search all nodes, and keep all nodes of the tree in memory.
- **Minmax with optimised alpha-beta pruning:** Time and space complexity is O(b^(d/2)). Given the same amount of time, it can search up to 2 times Standard Minimax's depth limit.

Modifications

Alpha Beta Pruning

(How it works)

Alpha beta pruning adds 2 new variables, alpha and beta, to the standard minimax algorithm. These variables are used to pass information between nodes or minimax function calls. For example, a parent node can inform a child node what max value or min value it has already seen. With this new information, the child node will know if the parent node already has a better alternative compared to itself, thus the child node can stop exploring its own children and prune itself from the search tree.

(Benefits)

This can greatly reduce the amount of nodes explored, hence reducing the amount of time required per search.

Iterative Deepening (IDS)

(How it works)

Iterative deepening repeatedly calls the minimax function on the root node, increasing the depth limit by 1 each time. The first minimax function call will have a depth limit of 1, e.g. we only explore the immediate children of the root node. This is repeated until the following conditions are met

- eg time is over, reach desired depth etc

Furthermore, each node now stores an array containing its children nodes, so it doesn't need to repeat creating its children between iterations.

(Benefits)

This enables us to always have a "best next move" ready, regardless of when our pre-set conditions stop the search.

For example:

- **Normal minimax:** Suppose the search depth is 7. However, during the search, we run out of time and can't fully explore all 7 levels. Then, minimax may return a very bad "next move".

- **Iterative deepening minimax:** Suppose the current iteration search depth is 7 and we run out of time and can't fully explore all 7 levels. Then, we can simply return the "best next move" found from the previous iteration with a search depth of 6.

Diminishing Returns Cutoff Principle
When deciding which depth minimax should search up to, there is a tradeoff between time usage and accuracy of the perceived best move. For example, if we choose a lower depth, then we will use less time, but the search may result in a less accurate best move, vice versa. Hence, we aim to find a suitable depth that balances these 2 factors.

To achieve this, if the previous iterations of IDS minimax keep returning best moves with similar evaluation scores, then we should stop searching and return the best move found so far. As searching further will likely expend much more time resources whilst yielding similar results, hence enter a situation of diminishing returns.

To implement this, if the last 2 iterations returned a best move with evaluations within m% of each other, then we will stop searching and return the latest iteration's best move found.

After extensive testing, we chose m = 10%. This enabled us to search deeper depths when required, otherwise stop searching early, thus better distributing our time usage.

Alpha Beta Pruning + Iterative Deepening + Move Ordering
(How it works)
With Iterative Deepening, after each iterative search, each node will have a corresponding min or max value (depending on whether it is a min or max node respectively). Then, for the next iterative search, we can first sort each node's children based on their previous iterative search's value before searching, in order to explore the most promising children first. Eg if a node is max, sort its children in descending order, as a max node will choose the child with highest value. Similarly, if a node is min, sort its children in ascending order.

(Benefits)
Iterative Deepening enables Move Ordering to be used, which then enables Alpha Beta Pruning to prune more nodes earlier.

It may seem wasteful to repeatedly explore the higher level nodes. However, since the branching factor is always greater than 2, the leaf level will contain the majority of the nodes. Hence, if we can prune more nodes earlier, then we can skip exploring more leaf nodes. Thus, ultimately still explore less nodes.

Time Per Turn
For each turn, the time spent searching for the next move is dynamically allocated depending on which of the following conditions are met first.

- **Max Time Allowed:** Determined by uniformly distributing the total time remaining between the max number of turns left.
- **Diminishing Returns Cutoff Principle:** (as explained above)
- **IDS Iteration Skip:** We know that the next iteration is likely to take longer than the previous iteration. Hence, if the previous iteration uses more than half the Max Time Allowed, then we will know that there isn't enough time to search the next iteration. Thus, we should stop searching and return the best move found so far.

Re-using the Search Tree
(How it works)

Our minimax search tree is kept in memory and updated between turns. After each turn, we find the child node of the old root node, which corresponds to the action just played. Then, this child node is set as the new root node. Afterwards, we discard the old root node and other nodes which are not part of this new subtree.

(Benefit)
This way, we can save time by not having to regenerate the same children nodes between turns.

Opening moves
(Inspiration)
In chess, there are set opening moves which a player can follow, regardless of what the opponent plays.

(Implementation)
In Infexion, after play many games between ourselves, we found a potentially strong set of opening moves to be 3 tokens in a triangular formation, which maximises the protection between them. However, without further experimentation, e.g. with machine learning techniques, we are uncertain what is a strong set of opening moves.

Hence, we only hard coded the initial 1st move and used minimax for the remaining moves.

(Benefits)
By hard coding the initial n moves, then using minimax for the subsequent moves, we can greatly reduce the total number of nodes explored, e.g. all the nodes in the first 2n layers of the search tree.

Heuristic

Our heuristic consists of 4 basic components, which will then be used to derive the two main features of our heuristic.

Component 1: Total power per side
(Inspiration)
In chess, a common heuristic is “Material Count”. It would assign a value to each piece. For example, assign 1 point to Pawn, 3 points to Knight or Bishop etc. Then, the heuristic would compare the total Material Count of each side.

(Implementation)
In Infexion, we can implement a similar idea by assigning a token’s power as its value, then comparing the Total Power per side.

Component 2: Defender score per side
(Inspiration)
In chess, another common heuristic is “Strong Pawn Structure”. Essentially, the heuristic would consider how well a given side’s Pawn Structure can defend its own pieces

(Implementation)
In Infexion, we don’t have different pieces like chess, however we do have token structures. After playing many games of Infexion between ourselves, we realised that a Strong Token Structure is a dense circular structure, as it maximises the number of ally tokens a given token can protect.

We say token A can defend token B, if and only if token A can spread to token B.

The importance of protection is shown with the following simple example. If token B gets taken by the enemy, then token A can spread to regain control of token B, thus not lose power.

This essentially maximises the protection between all ally tokens.

Component 3: Attacker score per side
(Inspiration & Implementation)
After playing many games of Infexion between ourselves, we realised we can extend the idea of protection between ally tokens to sufficient protection.

We define sufficient protection as follows:

- Suppose ally token A can be attacked by 2 enemy tokens.
- Then, we will need at least 2 other ally tokens, which can protect token A, in order to sufficiently protect it.

For example, following from the above scenario:

- If the enemy uses 1 token to take control of token A.
- We can use 1 ally token to retake control.
- Enemy can then use the 2nd token to retake control.
- Then, we also use the 2nd token to retake control.
- In the end, we gain final control of token A and not lose power.

Component 4: Power Sum of Tokens Insufficiently Protected (PSOTIP) per side
(Inspiration)
In chess, another common heuristic is “King Safety”. It essentially checks whether the King is under threat, e.g. being checked.

(Implementation)
In Infexion, we don’t have 1 most important piece, such as the King in chess, which determines win or lose. However, we can consider our high power pieces as more important than low power pieces, since they can impact a larger area of the board. Hence, we should check the power of the pieces that are insufficiently protected.

Combining All 4 Components: The Entire Heuristic
The two main features of the heuristic are constructed from the 4 components described above, as shown in the following equation.

Heuristic = Situation score + Effective Power
= [X * diff(Defenders Score) + Y * diff(Attackers Score)] + diff[Total Power - PSOTIP]
, where X and Y are coefficients

After extensive testing, we chose X = 0.5 and Y = 0.5 so that the Situation Score doesn’t dominate the heuristic.

These two main features of our heuristic complement each others’ weaknesses.

- **Situation Score:** Measures a player’s ability to defend and attack, e.g. the tokens’ positions. However, it treats all tokens equally regardless of their power. But this is addressed by Effective Power.
- **Effective Power:** Measures the total power a player has, which is sufficiently protected. However, it does not take into account where these sufficiently protected tokens are placed. But this is addressed by Situation Score.

Calculation of the Heuristic Pseudocode

```
[Finds the defender and attacker scores for both teams, and set up the calculation for effective power]
For each cell on the board:
    For each position reachable by this_cell:

        If other_cell occupying other_position on the same team:
            # this_cell acts as a defender to other_cell
            +1 to Defender score to that team
            Protection_score_array[other_position] += 1

        If other_cell occupying other_position on a different team:
            # this_cell acts as an attacker to other_cell
            +1 to Attacker score to the attacker cell's team
            Protection_score_array[other_position] -= 1

[Now calculate the effective power]
For each cell on the board:
    # is this position sufficiently protected?
    If Protection_score_array[cell_position] <= 0
        # NO, it is not sufficiently protected, do not include in calculation of effective power
        continue
    Else:
        +cell_power to the team of this cell

[Now calculate the heuristic]
Situation_score = X * diff(#Defenders) + Y*diff(#Attackers)
Effective_power_difference = Red effective power - Blue effective power
Heuristic = Situation_score + Effective_power_difference
```

Supporting Work

(Testing our agent against other agents)

We created different agents, such as an agent which plays random moves and an agent which plays greedy moves, to test our agent against.