AI part B Report

Introduction:
In part B of the AI project, we have implemented agents to play an adversarial infexion game. We have tried many different approaches and finally we decided to use a minimax search tree optimised by alpha-beta pruning, perfect ordering and top_k.

Random and greedy:
We firstly tried an agent with random play to test the functionality and had a look of how it performed. We notice that it can move as fast as it should do. Then we implemented greedy search, deciding which action to take by calculating the utility value of the board after applying the corresponding action. It has a good time and space complexity which are both O(n). Just as what we thought it would be, greedy search can easily defeat random search. Nevertheless, the disadvantage of greedy method is it cannot foresee the situation of a round and does not take the opponent's possible responses into consideration. These are two easy algorithms we have tried in the early stage, which have low complexity and low possibility to win.

Ordinary minimax:
Then we implemented the minimax search tree and ran the agent class several times. We pass the current board to the function *minimax*. Given the state of the current board, an action list containing every next possible move that we can apply on the current board will be generated. For every action in the action list, we will apply the action on that board and pass the new board to the recursive function *minimax* and keep doing it until it reaches the bottom depth we set for this search tree. At the bottom, we will use the evaluate function to assign a utility value to each board, and return a tuple which contains the utility value of that board together with the action just taken, to the previous level. The min level will choose the tuple with the smallest utility value passed to it and the maximum level will choose the biggest one, and then pass it with corresponding action to the upper level. We will go through the whole tree and every node at the bottom to find the largest possible value. The complexity of time is O(b^d) and the complexity of space is O(bd) where d is the depth of the search tree and b is the branching factor. If we use only the basic version of minimax search tree, we can only go through 2 levels without exceeding the time limit, but there is a small possibility that minimax can not defeat greedy algorithms. If we set depth equals 3, we can ensure nearly a hundred percent chance to beat greedy algorithms, but due to the resource limit, iterating through 3 levels will usually use up 180 seconds after about 20 turns of playing.

Alpha beta pruning:
We know that a minimax search tree with depth equals 2 is far from enough. From then on, we started to work on optimising the behaviour of minimax. Implementations

of some improved versions of minimax search tree including alpha-beta pruning, have been carried out to help us improve the performance of minimax search tree on depth and efficiency. We set alpha to negative infinity and beta to infinity and pass them to function minimax as arguments. In alpha-beta minimax tree, the alpha in max level is used for recording the largest possible utility value in the next level and beta in min level is used for recording the upper bound for utility value of nodes in next level. It will go through the first branch of the last second level of the minimax tree and change the alpha for the whole tree. If there is a utility value of nodes in other branches which is less than beta in the min level, such a branch will be pruned and if there is a larger utility value, we will loop through the remaining parts of that branch until we find a utility value less than beta of the whole tree or we reach the last node of that branch. If we cannot find a smaller largest number in the nodes of that branch, we will update our beta and go through other branches. This will decrease the space and time complexity of minimax slightly in the beginning and the end of the game and significantly in the middle game. The time complexity for the minimax tree using alpha-beta pruning is O(b^(d/2)) in the best case, but still O(b^d) in the worst case when we will go through all nodes in one level before we find the node with largest utility number in which case pruning provides no benefit. In our tests, when the depth of the tree is set to be 3, it will hardly exceed the time limit and have an extremely high chance to beat the greedy search. When the depth of our tree is set to be 4, it will run out of time after about 10 rounds of playing.

Perfect ordering and top k optimisation:
We expect our model to be able to foresee at least two rounds of plays which means the depth of minimax search should be greater than or equal to four, such that it can become a smart agent with the power to beat most of other opponents. Aiming to optimise the current existing strategy, we go over the week 4 lecture slides, and find perfect ordering which can further improve the efficiency of alpha beta pruning. Perfect ordering can significantly reduce the number of nodes that need to be evaluated during the search. In our environment, perfect ordering is achieved by sorting the action list according to the utility value of each action from greatest to least, before starting to iterate through the action list. After the sorting step, we can avoid the worst situation where we will go through nodes with small utilities before finding the largest utility in the alpha-beta pruning minimax tree. The time complexity will be steadily controlled in O(b^(d/2)). Having implemented the sort_action_list function, we confidently started to test our program, but the results were not satisfactory. We could detect that the action determination was only slightly faster than before, but overall speaking, there was little improvement between minimax with alpha beta pruning and minimax with alpha beta pruning along with perfect ordering. This leads to our introspection, one possible reason for such a slight improvement might be that the original minimax with alpha beta pruning has already achieved nearly O(b^(d/2)) time complexity, so we can not see a remarkable progress after applying perfect ordering since sorting the action list will also cost some extra time.

In order to implement a minimax search tree which can go through at least 4 levels, we do some research on adversarial skills and find methods such as top_k, transposition table, and truncated search might be helpful in improving spatial and temporal efficiency. Because we have already written a sort_action_list function, the top_k method will be the easiest one to implement. The core idea of top_k is accelerating the search speed by limiting the branch factor. Every time we just expand k actions with the highest utility value for further investigation instead of exploring all the next possible actions at the moment. The time complexity is reduced to O(k^d). When we set k equals 20, the behaviour of a minimax search tree with depth equals 4 is quite good. Till now, the minimax tree has been modified to our expectation, what remains to be done is testing different utility evaluation functions and adjusting parameters like depth and k.

Parameter adjustment:

After the optimization has been done, we start to test what value of depth and k, the branching factor for every level, should be taken to make our agent more powerful. For the depth of the minimax search tree, our desired value, as we thought from the very beginning, is at least 4. Therefore our test will be carried out to focus on the behaviour of different agents with searching depth ranging from 3 to 6, at the same time, we will try various k values to get the one which can balance performance and time complexity of the minimax tree. The most appropriate (depth, k) pair should be that the agent hardly exceeds the resource limit and is dominant on the board throughout the game. To better see the difference in performance of different parameters chosen, we give red and blue players different parameters and let them play against each other. The battle results and findings during the procedure are attached below.

For clarity, take “D3K40 3:7 D4K12” as an example, this statement means an agent using a minimax search tree with depth = 3 and k = 40 won 3 games out of 10 against an agent using a minimax search tree with depth = 4 and k = 12.

RED	score	BLUE	D4K12 dominant on board, just has a small chance of running out of time
D3K40	3:7	D4K12	
D4K12	7:3	D3K40	
RED	score	BLUE	D5K7 dominant, but lose due to timeout as D3K40 always has a token surviving
D3K40	4:6	D5K7	
D5K7	5:5	D3K40	
RED	score	BLUE	D5K6 wins steadily, and doesn’t show up exceeding the time limit
D3K40	0:10	D5K6	
D5K6	9:1	D3K40	

From the testing outcomes, we have two final selections for depth, k value pair, one is D4K10, instead of D4K12 as it has some probability to go over time, the other is

D5K6. Depth 6 is not taken into consideration because as depth goes deeper, k will be smaller than 6. The agent may not behave quite well if the k value is too small.

RED	score	BLUE
D5K6	20:10	D4K10
D4K10	12:18	D5K6

We run 60 battles between D5K6 and D4K10 and the data is shown above. The statistic indicates a higher winning rate for a D5K6 agent. Hence, we decide the final version of our agent will apply a 5-depth minimax search tree optimised with alpha beta pruning and perfecting ordering with top k equal to 6.

Choice on utility evaluation functions:

We have some utilities and tested the performance of them. We firstly use the total power difference between two colours as the utility function, which has the similar performance as ratio of total power of different players(My colour power/ opponent's colour power). Then we take the number of cells occupied into consideration because when we apply our agent in the game, it is unlikely for our agent to spread a token with a high power to occupy some other cells, which have a great influence on cells beside it, so we add the second parameter which is the total number of cells occupied by my agent divided by the total number of cells occupied by opponents. After adding this, the change of action taken is subtle at the beginning of the game but significant at the middle stage and end of the game. The final utility function is (my_power / opponent_power) + 3 * (my_cell / opponent_cell). Assume the total power and total cell of the opponent are both 2. A spawn action will always add 2 to the utility, and utilities will increase by 1.5, 3 and 4.5 respectively if we apply a spread action on a token with power 2,3 and 4. The gain of applying a spread action on a token with power greater than 2 will be larger than applying a spawn action. When a token's power is larger, the spread action will lead to a greater increase in utility. In this way, we can assign a greater importance to the spread action on a high power token, which helps us achieve the goal of controlling areas of the board.

Analysis on the attempt of MCTS:

After completing the realisation of minimax, we tried another adversarial search technology called Monte Carlo Tree Search and found its performance quite poor. The basic idea of MCTS is to build a search tree incrementally by simulating numerous possible outcomes of the game, and make a decision according to the win percentage of playout from that state instead of relying on a heuristic evaluation function. Since we abandoned this technique, its implementation will not be discussed in detail here, only key steps will be analysed.

First of all, at the selection stage, we apply the upper confidence bound policy to balance exploration and exploitation. In this way, we could focus on actions with a high winning rate by simulating more playouts from these stages, and at the same time leave flexibility for some other ordinary actions. After that, we expanded b new nodes(a new node contains the new board after applying a certain action, and b is

the branching factor, the number of valid possible moves for the next step), and added to the root node(which contains the current board). Then for each child node, we perform random playout to a terminal state many times, record its winning rate and backpropagate these statistics. Actually backpropagation is even unnecessary as we only expand one level. The number of simulations for each child node is determined by the UCB1 algorithm at the selection step. After the whole process is done, an action with the highest winning rate will be chosen.
Although MCTS has been shown outstanding in a wide range of AI applications, it does not behave well in our game environment. The first problem is the time efficiency. It is true that the complexity of MCTS is linear, not exponential, but it is still very slow in action determination. It takes about 30 seconds to carry out 1000 times of random 343-turn simulation. However, the maximum computation time per game, per player is only 180 seconds, which means MCTS will certainly lose if it can not win against the opponent in 6 rounds. This outcome is ridiculous so we must make some adjustments. The total times of simulation should not be decreased, we even consider 1000 playouts far from enough. Perhaps the total turns of every game can be made smaller. After changing the total turns from 343 to 50, MCTS can give out an action in about 5 seconds. However, when we use MCTS to play against the greedy agent, MCTS never wins. Even worse, MCTS is unable to win against the random agent. Based on these observations so far, we can make a conclusion that MCTS is not a suitable adversarial search algorithm for our agent design. We would say MCTS will certainly be a powerful algorithm as long as enough simulations are carried out because simulations allow us to learn from the actual outcome of a game. However, the time resource is strictly limited by the referee, so MCTS is not suitable under this constraint. Without regard to time limit, we may iterate 10000 times of playouts, and during each playout use a heuristic to pick which action to take next instead of just randomly choosing one from the action list.This strategy will turn out to be a dominant agent in specific situations, but the drawback is obvious, too time and space consuming. Therefore, MCTS will not be considered.

Conclusion:
In this project, we tried random search, greedy search, Monte Carlo tree search, minimax search and some variations of minimax search tree are implemented to see how they perform. The results of testing show that the minimax search tree with alpha-beta pruning, perfect ordering and top_k is the best choice for the agent. Also, utility function equals (my_total_power/opponent's_total_power) + 3 * (my_occupied_cells / opponent's_occupied_cells) is applied on calculating utilities of different states. At the end, the pros and cons of Monte Carlo Tree Search are analysed. Poor performance in limited time and space are the main reasons why MCTS is abandoned in this project.