AI Project Part A
Jiwon Bae 1229059 Jihoon Yang 1269967

Search Strategy

For our implementation of part A of the project, A* search algorithm was used to efficiently find the sequence of actions to win the single player version of Infexion. A* is an informed search algorithm that uses cost from initial to current state (g) and an estimated cost from current to goal state (h) to prioritise the nodes that finds the optimal solution. It is complete unless there are infinitely many nodes with f <= f(goal) and also optimal as we used an admissible heuristic function To implement A*, we have utilised priority queue imported from the Python standard library, where priority value was the estimated total cost (f) of each node. The priority queue was updated each time when a new node was expanded and was sorted according to their priority value.

We have also implemented BFS algorithm using queue with similar approaches to A* except that BFS did not use an evaluation function. It allowed us to compare the time and space efficiency between the two algorithms. The BFS implementation can be tested on code by observing and commenting the imports of the program.py file.

For A* search implementation, we created a variable ‘move’ and ‘h_n’ inside the class ‘Node’. The variable ‘move’ stores the value of the path cost g(n) and ‘h_n’ stores the heuristic value h(n) of the node calculated on node generation. When pushing the generated nodes into the priority queue, their arrangements are determined via the __lt__ comparison function defined inside the class ‘Node’, which compares the priority value f(n) of the nodes where f(n) = g(n) + h(n).

Length of solution (d)	Number of nodes generated to find solution			Average number of nodes generated
2	289	223	223	245
3	2185	823	1321	1443
4	1417	703	5827	2647
5	16699	27451	21589	21913
6	27673	50107	21331	33037
7	54595	51121	324487	68877

Table 1

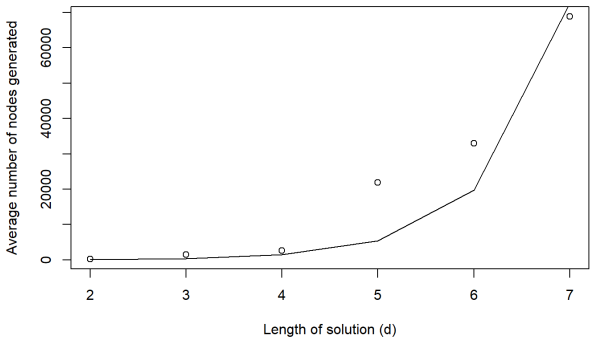

Figure 1.

The A* search algorithm has the same complexity for time and space, and they depend on the goodness of the heuristic used. The worst case complexity is $O(b^d)$, but with a good heuristic, the time complexity is $O(e^{[(\text{relative error in h})*(d)]})$. We have generated multiple test cases that give variable lengths (2~7) solutions (d) and the number of nodes generated, which is proportional to time and space complexities (Table 1). Furthermore, in Figure 1, we have plotted the average number of nodes generated against solution length (d) and have fitted an exponential curve, $f(x) = 8e^{1.302*d} \in O(e^{1.302*d})$. Our algorithm has $O(e^{1.302*d})$ time and space complexity.

Heuristic Function

Our heuristic calculates the minimum number of straight lines needed to cross all the cells inside the hex board that contains a blue token. The implementation was taken from a solution of a leetcode question online, and the function is commented on its source and details.

The implementation is as the following:
We assume that the number of blue hexes (B) on the board is greater than 0 at all times. We start off with the maximum number of lines that cross through B points, which is B - 1 since we can always connect 2 points with a straight line. We compare the slope of 2 adjacent points (lines) using the cross product and if they are the same, decrease the number of lines needed by 1.

As each spread move can only take over the blue cells in a straight line direction, the minimum number of moves required to take over all blue cells is always equal to or greater than the heuristic value. Therefore, our heuristic is admissible and does not compromise the optimality of A* search algorithm.

	# nodes generated (nodes)	**Time taken to find solution (s) (to 2dp)**
BFS	368569	2.79
A*	16699	0.11

Figure 2*. Number of nodes generated and time taken to find solution, using BFS and A* search algorithm. The test case is from the 'test.csv' file given in the skeleton structure.*

As shown in figure 2, the heuristic function allowed the search to find the least cost solution efficiently and sped up the process. Since it provided a rough estimate of the cost to the optimal solution, it allowed the search to avoid expanding nodes that would take extra steps to reach the goal state leading to significant decrease in time and memory.

However, using the heuristic function did not always guarantee that the solution it found was time and space efficient. For example, when there is only one blue token in the board, the heuristic for any expanded node would be 1 since the minimum number of straight lines needed to cross the cell is always 1. If A* and BFS were used to find the optimal solution for this scenario, A* would take longer time than BFS since calculating heuristics and adding the node to the priority queue takes time even though the heuristic would not enhance the search.

Spawn Action

To accommodate the SPAWN action, we would need to modify our spread function in node.py so that the child nodes generated include SPAWN actions on empty cells in addition to the original 6 direction movements for every red node existing.

This would result in increasing the branching factor (b) to a maximum of 53 (assuming at least 1 red hex and 1 blue hex remains - 6 SPREAD movements for 1 red hex and 47 empty hexes to spawn on).

Therefore in general, the complexity of the search would increase since the branching factor is higher than the normal single player and it would take much more time to find the optimal solution. However in some cases the SPAWN action could reduce the time taken since the depth of the search tree could be decreased. For example if there are one blue and red hexes in the board and they are far apart, spawning a new token adjacent to the blue hex will help find the solution faster as unnecessary spread is not needed for the red hex to reach the blue hex.