Artificial Intelligence Search Assignment:

Alex de Vryer and Ben Cox (Team 'DROP_TABLE_Students')

Search Strategy Used:

The search strategy we used for our program was A* search as described in the lectures. To run search, we created several Python classes to help guide search:

1. A SearchProblem class which defined an initial map state as well as a target coordinate and contained functions which found the heuristic value, determined if a given state is a goal state, as well as found new possible map states by finding all possible piece placements.
2. A SearchNode class which acts as a node in a search tree. It contained attributes such as the current state, the parent node along with the path cost and the specific PlaceAction used to reach the current state from the parent.
3. A PrioritisedItem class. This is the class which is inserted into the PriorityQueue (PQ) when conducting A* search. This contains a priority of value f(n) = g(n) + h(n), where g(n) is the cost of the current path to the initial state, and h(n) is the value of the predicted cost from the current state to a goal state. It also contains the SearchNode along with its heuristic value of the current state to speed up goal state checking (we only check if a given state is a goal state if its h(n) value is 0).

When running A*, we expand the current node with the lowest f(n) value to find child states. To do this, we implemented depth-limited search with a limit of depth 4 which found all possible shape combinations which could be achieved with four squares. We then created new child nodes for these states, calculated their f(n) value and placed non-duplicated states into the PQ and repeated this until we found a goal state.

In terms of time complexity, the worst case for A* will be O(b^d). In edge cases where our heuristic does not work well, we may have to generate every possible node up to the depth of the optimal solution until we reach a goal state. For an average case, we will say that our heuristic underestimates the actual cost of the solution by about ⅓ (33.33%) as observed by empirical evidence gathered whilst running test cases . Therefore, average time complexity will be O(b^(d/3)). Branching factor increases as search progresses; as more pieces are placed, there are more adjacent squares available for other pieces to be placed.

In terms of space complexity, A* keeps all expanded nodes in the PQ whilst running. As we only search up to the depth of the closest goal node, it has O(b^d) complexity.

Heuristic:

For our search algorithm, we used a simplified version of our game as our heuristic. Instead of placing pieces consisting of four squares, our heuristic determines an estimate of the cost by working out how many individual squares we would need to place to: a) get from the row/column of the current closest square to the row/column of the target coordinate, and then b) find the cost to fill up the row/column of the target coordinate.

Our heuristic calculates two values; the cost to fill up the row of the target coordinate as well as the cost to fill up the column of the target coordinate. It then returns the lowest value of the two to ensure admissibility (as the cost to reach the goal state with our simplified game is either (a) the same, or (b) lower than the proper game).

When calculating the cost to fill up the row/column, our function factors in the number of gaps in the row/column that need to be filled. For each gap, if both (a) the number of squares required to get from the closest red row/column and (b) the number of squares required to fill the gap is equal to 4 (the size of a piece), we just add the size of the gap to the heuristic value. As we have already factored in the cost to move from the closest red square to the target row/column in the heuristic function, we might be able to eliminate this gap by placing a piece from this closest square (so we do not overestimate the cost). If not, then we add a cost of 4 (the cost to place an entire new piece) to the heuristic value as we would need to place a whole piece to fill this gap anyway. For gaps larger than 4 squares, we just add the size of that gap to the cost. These strategies ensure that the value of our heuristic is closer (or exactly equal to) the true cost of the solution without overestimating it, meaning it is admissible.

Our heuristic does speed up search. As shown by the spreadsheet, we ran 19 tests using A* search with our heuristic and uniform-cost search (by always setting the heuristic value to 0). Under A*, our function completed all tests in under 30 seconds, but with UCS, our function could only complete two of these tests within that time frame. Both strategies had the same cost for test_vis2.csv as there is no solution (therefore we have to check every possibility). For test_vis1.csv, A* took 0.077s and expanded 11 nodes whereas UCS took 4.450s and expanded 1056 nodes. Our results are also always optimal in these test cases as with an admissible heuristic, A* provides an optimal solution.

		A*				UCS		
TEST		COST	TIME (s)	NODES		COST	TIME (s)	NODES
1		3	0.077	11		3	4.450	1056
2		0	0.059	42		0	0.054	42
3		4	19.132	1815		-	-	-
4		4	5.880	364		-	-	-
5		3	0.110	11		-	-	-
6		3	1.483	77		-	-	-
7		4	19.045	1908		-	-	-
9		3	0.494	18		-	-	-
10		3	11.437	493		-	-	-
11		4	11.295	379		-	-	-
12		4	13.595	614		-	-	-
13		5	0.369	70		-	-	-
14		4	0.545	50		-	-	-
15		3	3.154	165		-	-	-
16		9	0.177	16		-	-	-
17		4	0.135	55		-	-	-
19		4	0.230	18		-	-	-
21		7	7.661	967		-	-	-
22		4	8.789	1002		-	-	-

Modified Game Goal:

With a change in game goal from clearing a target blue token to clearing all to secure a win state, the complexity of the algorithm will increase. We could largely port our current algorithm to this new problem, but before commencing search, we would run our heuristic to find the estimated cost to clear each row/column with blue squares in them. We would then choose to target the row/column with the lowest estimated cost. Upon clearing that row/column, we would take the current board state and restart search using that state as the new initial board state and choose to eliminate the new lowest cost row/column. We would keep repeating this process until all squares are cleared.

Our initial search problem had an average time complexity of O(b^(d/3)) and a space complexity of O(b^d). As we are restarting search each time we clear a row/column, our space complexity would still be O(b^d) for normal A* search as we clear the queue each time we restart. However, our average time complexity would now be O(n*b^(d/3)), where n is the number of rows/columns which we need to remove to eliminate all blue squares.