Artificial Intelligence Search Assignment:

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

Search Strategy Used:

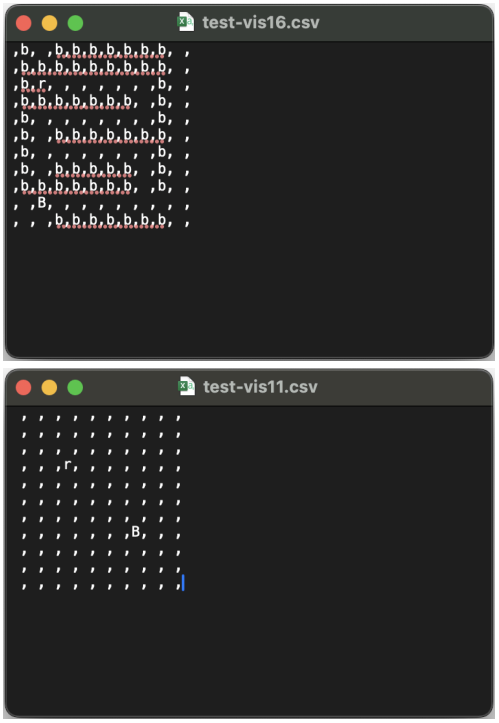
The search strategy we used for our program is A* search. As described in lectures, we expand the most desirable unexpanded node and find subsequent child states, adding non-duplicated states into the queue in order of decreasing desirability. We repeat this process until we reach a goal state. To keep the queue in decreasing order of desirability, we used a PriorityQueue where the priorities of all search nodes were set to the value of the evaluation function f(n) = g(n) + h(n) used for A* search. We also created a PrioritisedItem class (modified from Python Standard Library documentation) which contained the priority, the h(n) value to speed up goal state checking (we only check if a given state is a goal state if h(n) = 0) as well as the node itself.

We also created several other classes and functions to help with search:

- A SearchProblem class: contains an initial starting map state and a target coordinate. Also contains several functions such as a goal tester, a heuristic calculator as well as functions which find all possible PlaceActions and create new board states given a specific node. To find all possible PlaceActions, we implemented depth-limited search (limited to depth 4) to find and return all possible combinations of placing four squares given a particular board state.
- A SearchNode class: contains attributes such as the current state, a pointer to the parent node along with the path cost and the specific PlaceAction used to reach the current state from the parent. Also contains functions to expand the given node and create child nodes.

In terms of time complexity, the worst case for A* is 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)). 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.

We also observed that depending on the map state, the branching factor (and the subsequent time and space complexity) can vary massively. For test_vis16, despite the solution requiring 9 PlaceActions, it took our function under 0.2s and required 16 nodes to be expanded before reaching a goal state. However, for test_vis11, the solution only required 4 PlaceActions but took over 11 seconds and required 379 node expansions. As shown on the right, test_vis16 had a very limited number of possible placement combinations from the starting node to the end node (as the path formed a maze), heavily limiting the branching factor. However, for test_vis11, which only had one red square and one target blue square on the board, the branching factor is much larger (e.g. in the initial state, we can place all 19 pieces in any orientation, which resulted in 189 child nodes being generated upon state expansion by our program). As a result, solving certain maps takes longer and requires more memory.



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 those tests within that time frame.

		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 slightly. 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 restart search with the new map state, again choosing to target the row/column with the lowest estimated cost. We would keep repeating this process until all squares are cleared. Ultimately, we would modify our program to run in a loop; finding the row/column with the lowest cost and calling our A* search function each time; stopping when all blue squares are removed.

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 start off with an empty queue each time we loop our program. 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.