COMP30024 AI Project 1 Report

Team: Algorithmic Insight

Implementation Detail of Search Strategy

The search strategy implemented is an informed search algorithm, the A* pathfinding algorithm. A* uses the evaluation function *f(n) = g(n) + h(n)*, where *n* is the current node, *g(n)* is the path cost from the start node to *n*, and *h(n)* is a heuristic function that gives the estimated cost from *n* to the goal state and should be admissible to guarantee solution optimality. Therefore, *f(n)* is the estimated total cost of a path that reaches the goal through *n*. The heuristic implemented for this project will be discussed in the next section.

Regarding the use of data structures, we utilised a priority queue for A* search. As outlined in the pseudocode for the algorithm, in each iteration of the outer while loop, the state with the lowest *f(n)* value will be expanded upon, and its children will be generated. A priority queue is the most appropriate as it can maintain an array of unexpanded nodes or states in order of the lowest *f(n)* value to the highest. Our program imports the Priority Queue class from the Python queue module to achieve this.

We represent one state of the game as a class, which contains 1) the game *board* with any PLACE actions already made, 2) its *f(n)*, *g(n)*, and *h(n)* values, and 3) the most recently placed *PlaceAction*. We gave the class a hashable value by writing a method that converts the *board* dictionary into a frozen set. The hashable value is useful here because this allows us to store *State* instances in Python sets, which helps to remove duplicate *State* instances.

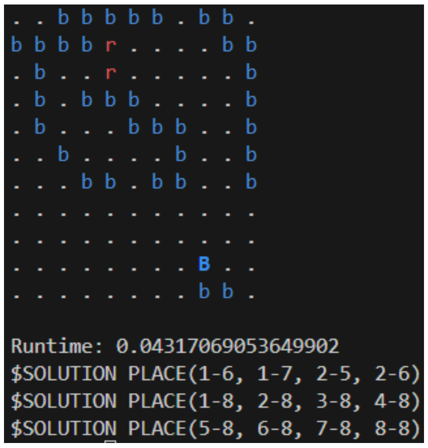
Heuristic Calculation and Efficiency

Please note that our *h(n)* is calculated in terms of the number of tokens a path will occupy, NOT the number of PLACE actions. Thus, to obtain the estimated number of pieces required, divide these values by 4 (the size of one PLACE action). We implemented it this way for simplicity, as it reduces the amount of floating point numbers being stored.

Diving deeper into the heuristic function, *h(n)* represents the estimated cost to reach the goal state, where the target token is deleted from the board. To achieve this, either the row or the column containing the target token must be filled. We estimate this cost by first calculating the nearest straight-line distance between the red tokens and the target column (row). Then, we determine the number of empty tokens needed to fill the corresponding column (row). Finally, we calculate the sum of these two values individually for the column and the row. The estimated cost is determined by taking the minimum of the two sums, which represents the estimated minimum number of tokens required to reach the goal state from the current state.

Our heuristic function h(n) is **admissible** (meaning it does not compromise optimality). It is derived from a relaxed version of the problem. In this relaxed scenario, obstacles between the red tokens and the target column or target row are ignored, and the cost does not need to be achievable by PLACE actions (e.g. *h(n)* can have a value of 13, which discrete PLACE actions cannot achieve). If there are no obstacles on the row or column to be filled, the number of tokens needed to fill it is exactly the number of unfilled tokens. Therefore, by this heuristic, our

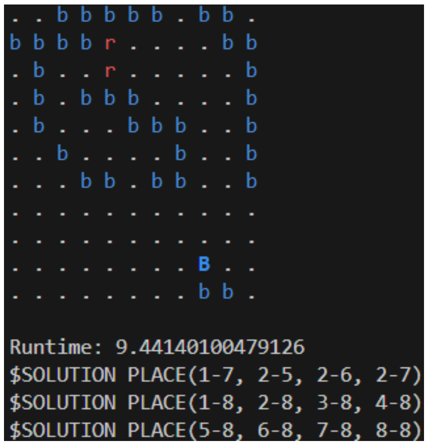
A* implementation can effectively prioritise the nodes that are more likely to lead to a solution, reducing the number of nodes that need to be expanded. This speeds up the search without compromising solution optimality, as shown in the following runtime comparison between the program with and without the heuristic (Figure 1 And Figure 2 respectively) for test case 1.



```
. . b b b b b . b b .
b b b b r . . . . b b
. b . . r . . . . . b
. b . b b b . . . . b
. b . . . b b b . . b
. . b . . . . b . . b
. . . b b . b b . . b
. . . . . . . . . . .
. . . . . . . . . . .
. . . . . . . . B . .
. . . . . . . . b b .

Runtime: 0.04317069053649902
$SOLUTION PLACE(1-6, 1-7, 2-5, 2-6)
$SOLUTION PLACE(1-8, 2-8, 3-8, 4-8)
$SOLUTION PLACE(5-8, 6-8, 7-8, 8-8)
```

Figure 1: Runtime and solution **with** heuristic



```
. . b b b b b . b b .
b b b b r . . . . b b
. b . . r . . . . . b
. b . b b b . . . . b
. b . . . b b b . . b
. . b . . . . b . . b
. . . b b . b b . . b
. . . . . . . . . . .
. . . . . . . . . . .
. . . . . . . . B . .
. . . . . . . . b b .

Runtime: 9.44140100479126
$SOLUTION PLACE(1-7, 2-5, 2-6, 2-7)
$SOLUTION PLACE(1-8, 2-8, 3-8, 4-8)
$SOLUTION PLACE(5-8, 6-8, 7-8, 8-8)
```

Figure 2: Runtime and solution **without** heuristic

Time and Space Complexity

A* search is exponential in time and space. Since the step cost is constant in this game, the time complexity can be written as $O(b^{\epsilon d})$, where d is the solution depth and epsilon is the relative error of the heuristic, which is given by $\epsilon = \frac{h^* - h}{h^*}$, where h^* is the actual cost of getting from the start state to the goal state.[1] In terms of space, A* search maintains all generated states in memory, so the space complexity is $O(b^{\epsilon d})$.

Discussion on Modified Rules

Considering a modified version of the problem where all Blue tokens had to be removed from the board to secure a win (not just one target coordinate), our team would take a different approach. First, we would change the goal state from removing a single target Blue token to removing all Blue tokens. We will not use A* pathfinding, and instead, replace it with Uniform-Cost Search (UCS), which is very similar to A* but without the heuristic. This decision stems from the unpredictability of the position of Blue tokens, which can vary greatly from game to game, rendering our current heuristic ineffective in speeding up the search. Nonetheless, UCS still guarantees an optimal solution, which aligns with our priority of achieving optimality over having a faster algorithm that might yield suboptimal solutions. The time complexity becomes $O(b^{d})$, with no heuristic to help reduce the solution depth, while all generated states are still maintained in memory, so the space complexity is $O(b^{d})$.

[1] Russell, S. J. & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd ed). Pearson. (p. 98)