Project Part A: Single Player Infexion

COMP30024 - Artificial Intelligence
4th April 2023

Jeremias Baur, 1421759
Tatyana Gordon, 1403898

1. With reference to the lectures, which search strategy did you use? Discuss implementation details, including the choice of relevant data structures, and analyse the time/space complexity of your solution.

We chose to use the A* search algorithm. The underlying principle of A* is a Best-First-Search algorithm which has been enhanced with a heuristic. Our chosen heuristic is explained in the second question.

First of all, we programmed a state data structure called `CoordinateSystem`. Its implementation details are viewable in the "coordinate_system.py" file. Each state is one CoordinateSystem object. The class CoordinateSystem uses a Python dictionary to save all non-empty cells as a key-value pair. The key is the r and q coordinates saved as an (r,q) tuple. The value of the dictionary is the color and power of this specific cell also saved as a ('color', power) tuple, in which 'color' is either 'r' or 'b' for red and blue cells respectively. Additionally, the CoordinateSystem class has helper functions to apply a SPREAD action in a specific direction, count the number of red cells in the coordinate system, and calculate the distance between two cells with consideration of the wrapping.

The initial state is loaded into a CoordinateSystem object and added to the priority queue with priority 0. Then the A* algorithm is started. It pops the element with the lowest priority from the queue. Afterwards, it iterates over every red cell and explores the six SPREAD directions. For each SPREAD direction, a new state is created. We calculate a hash of the state to check if it was already explored. If not, then it is added to the priority queue with the priority: g(s)+h(s), where g(s) is the steps so far to this state and h(s) is the value acquired by the heuristic function, which is later explained. After a state is popped from the priority queue, it is checked if the final state of no blue cells is reached. If it is reached, the A* algorithm terminates and the backtracking of the taken SPREAD actions begins.

The backtracking is implemented with a dictionary. When a state is explored, we save the parent state in the dictionary of the current state. Therefore we can recursively find the optimal path by traversing the dictionary from the final state up to the starting state.

The A* heuristic algorithm has a time complexity of $O(b^d)$ and the space complexity of $O(b^d)$, where b is the branching factor and d is the depth of the optimal solution. An upper bound for the branching factor is $b = 48$ because a maximum of 48 red cells could exist in a state. Each state takes a constant amount of memory to store, therefore the overall space complexity is not increased. The same holds for the time complexity because for each state exploration a maximum of 48 red cells could be spread in six directions, which takes constant time.

2. If you used a heuristic as part of your approach, clearly state how it is computed and show that it speeds up the search (in general) without compromising optimality. If you did not use a heuristic, justify this choice.

The heuristic h1 of our A* algorithm calculates the distance (how many tiles) from each red cell to each blue cell. It saves the shortest distance and divides it by six to ensure that the heuristic is admissible. For a heuristic to be admissible, the heuristic cost must always be less than or equal to the true cost to reach the goal state. The distance calculation would be correct if each red tile had a power of 1 since the distance would equal the number of moves the red tile would have to make. However, since each red tile can have a power of 1-6, dividing the calculated distance by 6 better reflects how many moves it would take to reach the blue cell and makes the heuristic admissible and therefore optimal since it will always be less than or equal to the true cost. An example is given in *Figure 1*.

Another heuristic h2 that we implemented was to calculate the percentage of blue cells in each state. Then we prioritize the states with fewer blue cells since they are closer to the goal state. h2 is also admissible because it will always return a value between 0 and 1. The final state has 0 blue cells so h2 will return 0. Every other state takes at least 1 step to reach the final state and therefore we are always lower or equal to the true cost.

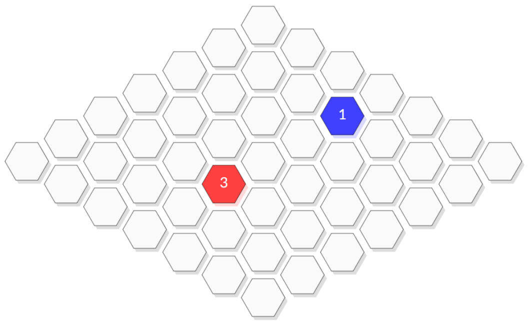

Figure 1: The distance between the blue and red cell is 2. Only one SPREAD action is required to reach the final state because the red cell has a power of 3. A heuristic using only the distance isn't admissible because it predicts a cost of 2. Therefore our heuristic divides the distance by 6 since it is the maximum power.

Then we combine these two admissible heuristics h1, h2 into the final heuristics h=max(h1,h2), which is also admissible and therefore optimal. This final heuristic speeds up the search because they guide the program to explore the states with fewer and closer blue cells first since they are closer to the goal state. A BFS algorithm would take a significantly longer amount of time to reach the goal state because it would explore every state in the order it comes across it, even if it is moving in the opposite direction of the solution.

3. Imagine that SPAWN actions were also allowed in single player Infexion (not just SPREAD actions). Discuss how this impacts the complexity of the search problem in general, and also how your team's solution would need to be modified in order to accommodate it.

Generally speaking, the search space will be massively increased because we could place a new cell onto every non-empty cell in a specific state. Since the branching factor would increase, the time and space complexities would increase proportionately.

One way to modify our solution would be to spawn red cells in every empty cell and add those states to the priority queue after every spawn. This is very inefficient but would ultimately reach the goal state.
A general observation is that every blue cell which can't be reached by a red cell in one or two moves can be turned red by spawning a cell next to it and then spreading onto the blue cell. Our algorithm could use this strategy to reach blue cells faster. New heuristics could be implemented that use this observation and calculate the distance from each blue cell to the closest red cell. Since it would take two moves to spawn a cell and spread it, if the calculated distance is more than 2 moves, then a red cell should be spawned adjacent to the blue cell.