COMP30024 Part 1 Report - Yifan Yang & Yuanyuan Xiang

Search strategy:

For this problem, we decided to use the Iterative Deepening A star (IDA*). This algorithm builds upon the A star algorithm ***(week03.pdf lecture slides, informed search strategies)***, and combines it with the idea of iterative deepening search ***(week2.pdf lecture slides, uninformed search strategies).***
The IDA* still uses the same cost estimate function as A star, as defined in the lectures:

f(n) = g(n) + h(n)

The difference between IDA* and the typical A* lies in how the search is performed. Unlike the typical A*, in IDA*, there is no need for a priority queue, as it is a DFS algorithm. At each iteration, we cut off the branch when its cost f(n) exceeds a given bound, and then trace back the stack to search another branch. So initially, we set the bound to be the estimated cost f(n) of the initial state. Then, at each iteration, we choose the bound used for the next iteration to be the minimum cost of all values that exceed the current bound (essentially we're exploring branches to a deeper level with each iteration). The search continues until a goal state is reached (whose cost does not exceed the current bound).

Our search strategy is quite similar to the pseudocode given on the following page, so I will not go into further detail about how the search is done:
https://en.wikipedia.org/wiki/Iterative_deepening_A* **[2]**

Optimality: Just like A*, IDA* is also optimal as long as the heuristic is admissible, this is because of the way that we increase the bound. In other words, we can never "overshoot" a level that contains the optimal solution.

Time complexity: The exact time complexity of IDA* is incredibly difficult to calculate. "The running time of IDA∗ is usually proportional to the number of nodes expanded. This depends on the cost of an optimal solution, the number of nodes in the brute-force search tree, and the heuristic function" **[1]**.

However, let's assume a worst case scenario where our heuristic function does nothing, h(n) = 0, then our IDA* search essentially becomes an IDS, where we explore each depth multiple times until we reach the depth with the optimal solution. So based on the lectures, the time complexity is **O(b^d)**. Where *d* is the maximum depth of the search (so d=343 as that's when there's a draw and the search is stopped). For the branching factor *b*, let's again assume a worst case scenario of 48 red cells on the board, in which case each red cell has 6 directions to spread to, producing 48*6 = 288 outcomes. So our branching factor *b* = 288.

That being said, in practice, our search tends to find solutions quite quickly because of the heuristic, allowing us to essentially "skip ahead" in terms of the depth levels of each iteration. And, the better the heuristic, the better the time complexity. In addition, after some testing between using IDA* versus the typical A*, we find our solution to be a lot faster when using IDA* for most test cases (using the same heuristic for both search algorithms), we suspect that this is also due to the fact that we do not need to keep track of a priority queue (heap), in which maintaining the heap would require log(n) time, and since the branching factor is so large, this time can blow up quite quickly. **[4]**

Space complexity: The space complexity for IDA* is **O(bd)** in the worst case, where *b* is the branching factor (b = 288), and *d* is the maximum search depth (d = 343). This is because the search acts like an IDS, so we only keep the nodes of the current branch that we're exploring, down to the current depth limit.

Heuristic:

Let's relax the rules a little bit, and consider what happens if each **red cell** is given infinite power to spread in one direction. Essentially, every time a red cell spreads in one direction, that entire line (all 7 cells in that direction) becomes covered. That means if a red cell happens to be on the same "line" as a blue cell, it can take over that blue cell. This problem therefore becomes: find the **minimum number of lines** that it takes to connect all the blue cells (these "lines" can be a combination of lines covering a row, lines covering a column, and lines covering a diagonal). Clearly, this would be an admissible heuristic, as it is just a simplification of the original problem.

But after some research, it turns out that finding the **minimum number of lines to connect all blue cells** is a set cover problem in disguise, which is an NP-complete problem **[3]**. So we decided to simplify the problem even more, and look for an even more relaxed heuristic. Instead, we ask the following question: What is the **least number of "expansions"** that it would take for all **blue** cells to be connected? (***an "expansion" being a blue cell spreading infinitely in all 6 directions***). That is, when a **blue** cell "expands", it covers all 3 lines: the row, the column, and the diagonal that it is on. This is still an admissible heuristic, because it actually gives a value that is **smaller than or equal** to the minimum number of lines to connect all blue cells. This should become apparent once we explain the algorithm.

The algorithm

1. Initialize a queue with all the blue cells, the ordering does not matter.
2. Initialize an array (with 21 elements) to keep track of the visited "lines", that is, all 7 rows, columns, and diagonals.
3. Initialize a counter for the number of expansions needed
4. While the queue is not empty:
   Deque a blue cell:
   If it is on any of the visited "lines", continue to the next cell in the queue.
   Else, "expand" the cell and mark all 3 lines that are covered as a result. Increase counter by 1.
5. Return the counter

Since there are at most 48 blue cells on the board which we need to put into the queue, this heuristic can return a value anywhere from 0 to 7, which enables the search algorithm to differentiate between really bad options and not so bad options. Combined with the fact that it's admissible, it is able to speed up the search substantially.

Spawning effects

If we allow spawning of red cells, we wouldn't have to change the heuristic, as it is always admissible even with spawning. This is due to how we relaxed the rules in order to come up with the heuristic, as we assume red cells can spread infinitely in a given direction, and we completely disregard the positioning of red cells in relation to the position of blue cells. However, this heuristic would be slower with the addition of spawning, as spawning would worsen the time complexity of the search algorithm, as the branching factor would increase by 48 (because red cells can spawned in any of cell on the grid), resulting in increased number of board states at any point.This would considerably slow down our search function, as our heuristic would return the same values regardless of whether the move is spread or spawn.

To improve our search, we would need to add the mechanical function of spawning cells on top of spreading, we could also modify the heuristic to take advantage of spawning, such as looking at whether or not there are existing red cells on the lines occupied with blue cells, and in this case it may be more efficient to spawn a red cell on those lines instead of spreading.

References:

1. Time complexity of iterative-deepening-A∗ Richard E. Korf, Michael Reid, Stefan Edelkamp https://www.sciencedirect.com/science/article/pii/S0004370201000947
2. Iterative deepening A star https://en.wikipedia.org/wiki/Iterative_deepening_A*
3. Bipartite graph-covering, special case https://math.stackexchange.com/questions/839784/minimum-vertices-set-bipartite-graph-covering-special-case
4. IDA* versus A* search https://cs.stackexchange.com/questions/118121/why-is-ida-faster-than-a-why-does-ida-visit-more-nodes-than-a#:~:text=IDA%E2%88%97%20does%20not%20have,at%20all%20by%20IDA%E2%88%97.