COMP30024 Project 1 Report

Benjamin O'Donoghue and Henry Routson

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

Heuristic complexity

Simple O(s) Where s = number of filled squares on board
As the squares are filtered from the dictionary keys to find the closest
And checking distance is O(1)

A* O(b^d)
Remember that this is not a normal a* implementation
This implementation is not searching for a single coord, but an entire line,
and so the time to find this is much shorter given there are 11 times as many coordinates
which results in a solution, which is why it was considered as an effective heuristic.

BFS O(b^d)
The branching factor b is reduced by the use of heuristics.
Although we could have used IDS we had problems with speed before space so we kept BFS which is faster.

Min heap / priority queue
Allows O(log(n)) insertion rather than an O(n log(n)) sort for every insertion where n is number of nodes

Combined complexity

*O(Number of nodes * (node put and pop in min heap + heuristic_calculation))*
*O((*b^d) * (2log(n) + *O(s)*)) =
*O((*b^d) * (log(n) + *O(s)*))

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 based approach, justify this choice.

1 Our heuristic
Our heuristic first finds the lowest number of squares to fill a row or column. The heuristic then divides the number by the largest number of squares you can place down at a time with one piece, to get the minimum number of placements to remove a target piece.

min(min_placements_to_complete_column, min_placements_to_complete_row) such that

(given x is a row or column)
min_placements_to_complete(x) =
min_squares_to_complete(x) /
max_squares_per_placement

min_squares_to_complete(x) =
min_squares_to(x) +
min_squares_to_fill(x)

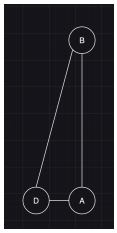
Min_squares_to can be calculated a number of ways, with varying tradeoffs

The simple solution we initially used is to use simple manhattan wrapping distance.
However this does not perform well on certain examples where there are obstacles in the way of getting to the column or row.
Because of this we implemented an a* algorithm to take into account blocking obstacles and line elimination, however after testing this, we found it significantly impacted performance, despite making a number of optimisations.
We decided to use the simple heuristic to avoid going over the 30 second time limit in marking.

Another aspect of our heuristic is using an effective break even to allow us to search more promising paths first.
We chose to use a non admissible break even of the number of squares to complete the row or column.
It's important that the first sorting cost for the heuristic is admissible, meaning it will never overestimate the cost.
This is to ensure that there is not a path which is overestimated and then a more costly path is seen first
and used as an invalid minimum cost solution. This ensures the solution is always optimal.

How much the heuristic improves the time complexity is extremely example dependent.
While in many examples, if the heuristic cost matches correctly to the row or column which is the actual least cost,
it effectively directly finds the solution where otherwise it would have taken 100 times the time. However if there is a local minimum where the lowest cost solution involves the row or column which is not the lowest cost estimate in the solution, the search can get stuck trying to complete the wrong line. This occurs when using our simple heuristic if there is a blockage which means that a partially completed line is not the lowest cost to complete.

2 How does a heuristic speed up search
An admissible heuristic is used to avoid searching the entire tree for an optimal solution.
Below is the simplest possible example for how a heuristic improves search performance for finding optimal paths.
While other algorithms like dijkstra which work without heuristics would first search node D (for distraction),
if you have a heuristic you can search directly to B first (ignoring immediately finding the solution).
For example a heuristic which takes into account the height of nodes, would realize that while A to D is a short path, this doesn't get closer to B, and so while A to B is a longer path, this is shorter than A to D plus the height to reach B.



3. Imagine that all Blue tokens had to be removed from the board to secure a win (not just one target coordinate). Discuss how this impacts the complexity of the search problem in general, and how your team's solution would need to be modified in order to accommodate it

You could use this algorithm to play a game where you were trying to eliminate the piece of other colors.

The complexity of this problem would dramatically increase, simply based on the number of pieces needing to be placed, the depth would dramatically increase. Although all blue tokens can be removed one by one, this would not be the minimum number of placements. The heuristic and finish conditions would need to change to eliminate all blue tokens with the minimum number of squares, and take into account how completing certain rows would remove pieces.