COMP30024 Part A report

Anthony Hill 1305284
Liam Anthian 1269187

When selecting an appropriate search strategy for any given issue, the main motivating factor is always the end goal, with the maximum branching factor and max depth of the state space for the problem helping to narrow down the strategy. For this project, optimality is the most important objective - a solution needs to be reached with the least possible moves, and efficiency of finding this solution and everything else comes secondary. Regarding the case of tetrominoes on an 11x11 'wraparound' board, the number of possible moves for a given player at any point in time is highly variable, but it is especially high in the worst case scenario of a sparsely filled grid - a single red tile on an otherwise empty board has 164 possible next moves. Even from the starting point in the *test-vis1.csv* example there are 19 possible next moves. With a branching factor this massive, it is especially important to choose the correct search strategy.

We ended up implementing a variant of an A* search as shown in Week 3: Informed Search Algorithms. We queued up states in a custom defined "Priority Dictionary" (PD), with behaviour mimicking the inbuilt python priority queue (Queue.PriorityQueue), but better insertion time for states. Priority was derived from a combination of current count of moves taken g(n), and a heuristic h(n), to indicate how much closer to the goal a current state now is. For cases where priority is equal between two states, the most recently expanded (and in turn typically deeper) state is placed first (LIFO for equivalent states) in the PD - this provides a depth-first element of exploration towards the end goal without sacrificing the optimality of solution seeking. This approach also removes the need to generate and expand unnecessary branches, effectively pruning them as search continues; finding the final goal state faster. The Priority Dictionary structure allows us to still retrieve the lowest priority item in O(1) time, but also insert new items in O(1), an improvement of Queue.PriorityQueue's O(log(n)) heap insertion where n is the length of the queue.

The next most ideal search algorithm to use could've been Breadth First Search (BFS) (Week 2: Problem Solving and Search), but after planning and discussion, it was deemed highly inefficient due to the high branching factor of the project as stated above. An alternative additional improvement could be queuing moves instead of states, but our PD's size could increase by up to a degree of 164 (as previously noted for all possible moves) - and search would become quite flat.

Space/Time complexity

Let 'b' be the branching factor.
Let 'b*' be the effective branching factor (reduced exploration of branches due to heuristic).
Let 'd' be the depth of the shallowest goal node.
Let 'm' be the effective depth of the solution. (m <= d)

Both space and time complexity can be very high given the branching factors and max depths present.
For time complexity, in the worst case:

- **Worst case time complexity would be O(b^d**) where every possible state is explored until the goal state is found - this occurs in situations that require non-straight forward progress towards removing the target.
- However, in most scenarios our heuristic optimises the path of exploration thereby significantly reducing search space. The **effective/average time complexity is O(b*^d)**.

For space complexity, in the worst case:

- All unexpanded nodes are stored in priority dictionary
- All expanded nodes are stored in a 'seen' list (and no duplicates occur)

- Thus, the **worst case space complexity would be O(b^d).** Since not all branches are explored and heuristic is optimal, **effective/average space complexity would be O(b*^m)**.

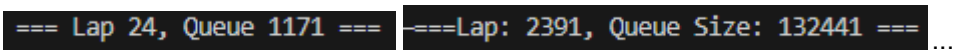
Our heuristics developed were all purposefully admissible, that is, less than or equal to the minimum number of moves still needed to reach a goal state. Resultantly optimality could never be compromised - at worst, if any chosen heuristic was uninformed we would still reach an optimal end goal (greediness avoided!). Our end heuristic was a measure of the closest red tile to a most filled axis of the target tile, represented in least possible moves. Walked through:

- To clear the target, either the y or x axis must be filled with tiles
- To approach (and begin filling) either axis, a pre-existing red tile must be built off of
- It follows that an optimal solution has to have more than or equal to the cheapest amount of tetrominoes placed to reach and fill an axis
- Thus the heuristic finds the best minimum sum of a red tile to axis approaching and filling for each given board state

Calculation:

```
h = 15      (max possible value for h)
for cell in red_cells
        a = distance to target x-axis from cell + free cells in x-axis
        b = distance to target y-axis from cell + free cells in y-axs
        If min(a,b) < h:
                h = min(a,b)
return h
```

With our heuristic, the solution to *test-vis1.csv* is found after exploring only 24 states, and ends with a queue of 1171 other generated but unexplored states. Without our heuristic, the code was unable to find a solution after over 2000 states had been checked, and a remaining queue size of over 130 thousand generated but unexplored states. This second run was left running for over 1.5 hours to no success.



...

(left - Heuristic, right - No Heuristic & cut short)

If all Blue tokens had to be removed from the board to win, the task's complexity would significantly increase, possibly becoming NP-hard. Significant care would be required to reapproach the problem at hand. The difficulties of this changed goal state increases complexity by:

- **Increasing search space** significantly
- Implicit creation of **interdependencies** as the removal of one Blue token impacts the strategy to remove other blue tokens, which is extremely difficult to plan for
- Depth increases so **optimisation becomes more difficult** (finding the least number of steps to get to goal state).

An attempt on this adjusted goal state would be to include:

- Creating a **move evaluator** to evaluate long-term implications of moves on the board, rather than immediate changes.
- **Pruning techniques** as shown in Week 4: Game playing and Adversarial Search, in order to reduce search space. This would require evaluating game state at a greater depth than current.
- **Adapted A* heuristic**, which takes into account which token to prioritise removing based on their relative positions and future potential on the board.