COMP30024 Report

# 1. How have you formulated the game as a search problem?

- **Formulate the game as a search problem:**
  - Initial state: The initial state is the board configuration that was read from a JSON file.
  - Actions: White token can be moved in one of the four cardinal directions or initiate a boom action.
  - Transition model:
    - Move 1 from (1, 3) to (1, 1) lead to a white token to be placed at position (1, 1).
    - Boom at (2, 1) lead to all the stacks in a 3 * 3 area surrounding this token to be removed and exploded.
  - Goal tests: Check if there are any black token left in the board. If all the black stacks have been removed from the board, the game ends.
  - Path costs: 1 per token move.

# 2. What search algorithm does your program use to solve this problem, and why did you choose this algorithm?

- **What search algorithm did we use?**
  - We have formulated the game as a BFS problem. This algorithm simulates all the possible movements of the white stacks and checks if any of the possible movements lead to elimination of all the black stacks on the board.
- **Why did we choose this algorithm?**
  - The BFS algorithm is complete, this means that we will be guaranteed to find a solution even for a tricky test case.
  - It can find the shortest path to win the game.
  - It is also very easy to implement.

- **Heuristics?**
  - **Mark Board:**
    - For each position on the board, we calculate if it booms at this position, how many black stacks would be destroyed and store this mark data in a mark dict variable, which can be visualized by the print board function like the graph beside.
    - We use this dict to limit the boom behavior in our search process. A boom behavior can be stimulated only when it was on the position that have mark in the mark board.
    - We also make use of the position that has highest mark to improve our BFS, which will be discuss in the Dynamic Refresh part.

```
# mark dict
#    +---+---+---+---+---+---+---+---+
#  7 |   |   |   | 1 | B1| 1 | 1 | B1|
#    +---+---+---+---+---+---+---+---+
#  6 |   |   |   | 1 | 1 | 1 | 1 | 1 |
#    +---+---+---+---+---+---+---+---+
#  5 |   |   |   |   |   |   |   |   |
#    +---+---+---+---+---+---+---+---+
#  4 |   |   |   |   |   |   |   |   |
#    +---+---+---+---+---+---+---+---+
#  3 | 3 | 3 |   |   |   |   |   |   |
#    +---+---+---+---+---+---+---+---+
#  2 | B1| 3 | 3 |   |   |   |   |   |
#    +---+---+---+---+---+---+---+---+
#  1 | 3 | B2| 3 | 3 |   |   |   |   |
#    +---+---+---+---+---+---+---+---+
#  0 | 3 | 3 | B1| 3 |   |   |   |   |
#    +---+---+---+---+---+---+---+---+
# y/x  0   1   2   3   4   5   6   7
```

  - **Euclidean Metric:**
    - When thinking about how to reduce the branches of our BFS, we try to reduce the unwanted movement. For example, moving a white stack to a place that is farther from any black stack, obviously this movement is useless for winning the game. So one way is to calculate the distance between the new position and each black stack, and comparing with the distance between the original position and each black stack, if any distance reduce (which means that this movement make the white stack closer to at least one black stack) we will add it to our potential movements, but otherwise we won't consider this movement.
    - Finally, we didn't use it to solve the problem actually, because it needed to calculate the distance for every potential movement for every black stack and compare each distance between after movement position and original position, which would also cost too much time on itself. But based on this idea, we thought deeply about another way that can calculate only once but can be used multiple times which is Trim Board.
  - **Trim Board:**
    - The BFS algorithm will run for a long time if the depth and the branching factor of the problem are high. So, we have implemented the 'trim_board' function to delete all the board positions that do not lead to the desired outcome. This 'trim_board' function works by checking each of the empty positions on the board, if there is no stack in a 3 × 3 area surrounding the empty position, the empty position is then considered to be trimmed and is 'unavailable' for the white token when

```
# trim board
#    +---+---+---+---+---+---+---+---+
#  7 | X0| X0| X0|   | B1|   |   | B1|
#    +---+---+---+---+---+---+---+---+
#  6 | X0| X0| X0|   |   |   |   |   |
#    +---+---+---+---+---+---+---+---+
#  5 | X0| X0| X0| X0|   | X0|   | X0|
#    +---+---+---+---+---+---+---+---+
#  4 | X0| X0|   |   |   |   |   | X0|
#    +---+---+---+---+---+---+---+---+
#  3 |   |   |   |   | X0| X0| X0| X0|
#    +---+---+---+---+---+---+---+---+
#  2 | B1|   | W1|   |   | X0| X0| X0|
#    +---+---+---+---+---+---+---+---+
#  1 | W1| B2|   |   | X0| X0| X0| X0|
#    +---+---+---+---+---+---+---+---+
#  0 |   | W1| B1|   | X0| X0| X0| X0|
#    +---+---+---+---+---+---+---+---+
# y/x  0   1   2   3   4   5   6   7
```

during the move action (The white token will not consider to move to the trimmed position, even if the white token is able to move to that position). This will decrease the breaching factor of the white token movements.

- **Dynamic Refresh**
  - In the process of BFS algorithm, if there is a boom that happen on a position that have the highest mark in the mark dict, and this boom only destroy one white token, we would consider it is a good boom behavior. So, we would stop all the other branches and only keep this branch running with refreshed filter data (refresh the mark dict and the trim board after this boom behavior).
  - This dynamic reduction on the number of available movements for the white stacks further decreases the branching factor, thus increasing the efficiency of the BFS algorithm. However, this “boom behavior” based refresh may damage the characteristic of “absolute shortest path” of BFS. We consider it is worth doing as it can significantly reduce the amount of calculation and still be able to find a path that is almost a shortest path.

## 3. What features of the problem and your program's input impact your program's time and space requirements?

- **Input Stacks Sparsely Located:**

  If the black stacks are located sparsely on the board and there are only certain locations for the white token to boom a group of black stacks to win, then it is going to take some time to run the algorithm. This is because the 'trim_board' won't be as effective when trimming a board with stacks sparsely located around the board, as it would be too risky to delete too many board positions and ending up isolating some of the stacks. This leads to a big branching factor and makes the BFS algorithm to run lots of executions as well as an increase in space complexity.

- **Input Has More White Token:**

  As the result of increasing number of white tokens, the BFS would have more branch in each search step, which would hugely increase the program running time. And if there are more tokens, our trim board and dynamic refresh would not work as efficiently as now because there may be few positions to be limited on the board and the boom action may have to affect other white token.

- **Path depth:**

The number of steps needed to move the white stacks from the initial position to the destination will also affect the speed of the algorithm. The more the steps need, the longer the algorithm is going to run. This is because the number of steps represent the depth of the BFS tree.

- **Dynamic refresh not always applicable:**

When a boom action from a path is initiated, and if there is no white token other than the boomed white token is lost in this action, the BFS algorithm will then only be finding paths after the boomed action branch and throw away all the other branches. This minimizes the time and space requirements. But if there is no boom action that would lead to no white token casualties. This would lead to no branch deduction and would lead to a higher space requirement and would take longer for the program to run.