Project Part B: Playing the Game Report

ChengMing Liew 1266882, Aixuan Tan 1276569

May 2024

Approach to Tetress

For our game-playing program, our implementation employs the Monte Carlo Tree Search (MCTS) algorithm.

- The algorithm does not depend on a heuristic function where different assumptions are made from the game, rather, it depends on a utility function. This utility function is computed from the actual state of the game where many playouts / simulations are ran from a specific state of the game.
- Furthermore, in the game of Tetress, where 19 different pieces can be placed in an 11 x 11 board, the branching factor would be large. Hence, using the MCTS algorithm would significantly help in looking several more moves ahead compared to an algorithm like Alpha-Beta pruning.
- In our utility function, we made use of a selection policy called **upper confidence bound** applied to trees, and has the formula UCB1(n) for a node n:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{logN * (Parent(N))}{N(n)}}$$

  - U(n) is the utility of all playouts through node n, or in this case the number of wins
  - N(n) is the total number of playouts through node n.
  - Parent(n) is the parent node of n in the tree.
  - C is a constant that balances the exploitation term $\frac{U(N)}{N(n)}$ and the exploration term $\sqrt{log(N) * (Parent(N))}$, which is $C = \sqrt{2}$ in our implementation.
- Using a UCB constant of $C = \sqrt{(2)}$ is a correct theoretical value as referenced from Artificial Intelligence: A Modern Approach, Fourth Edition by Stuart Russell and Peter Norvig.

Performance Evaluation

To judge the performance of our program, we have played it against a random agent, and also comparing itself with different values of C, the UCB1 constant.

- Running the agent with MCTS algorithm against a random agent, where it selects any random valid move, we notice a high success rate for the MCTS algorithm. Testing the 2 agents multiple times in 50-game splits, we can conclude an average of an 80% win rate of the MCTS algorithm against the random agent.
- We also run MCTS algorithm with different UCB constants, C = 0.25, $\sqrt{2}$, 1. After running the agents against each other, we find out that $C = \sqrt{2}$ performs the best compared to the other.

Other Aspects

Finding Possible Actions

In our implementation, we created a method that looks for every empty cell that are adjacent to the turn-coloured cells, **find_free_adj_color**. However this did not account for when the piece's origin was NOT the specified cell.

- For a piece to "reach" a cell that is free and also adjacent to the turn-colored cells, it would need to be within a Manhattan distance of 3. A list of 24 coordinates, CHECKSPACE, mapping the distance between **free_adj_color** and its surrounding cells (alternative origins), was manually computed.
- To reduce the numbers of valid surrounding cells, we change the origin of **pieces._TEMPLATES** to be consistently on the top-most, left-most cell. This allowed us to narrow down CHECKSPACE to only 12 coordinates, because the removed 12 coordinates could only have the starting **free_adj_color** as their piece origin.

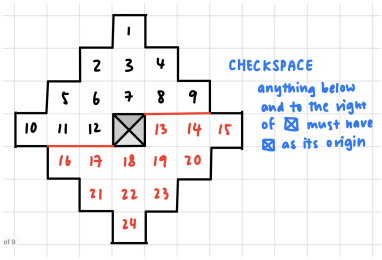

Figure 1: CHECKSPACE

- However, this was still not optimal – there were overlaps between the CHECKSPACEs of each **free_adj_color**, and each cell had different valid pieces. We then noted that if an origin cell had its directly adjacent RIGHT and DOWN cells filled, it would be an impossible origin, i.e. no piece could be placed, hence creating the gating function **check_botright**.

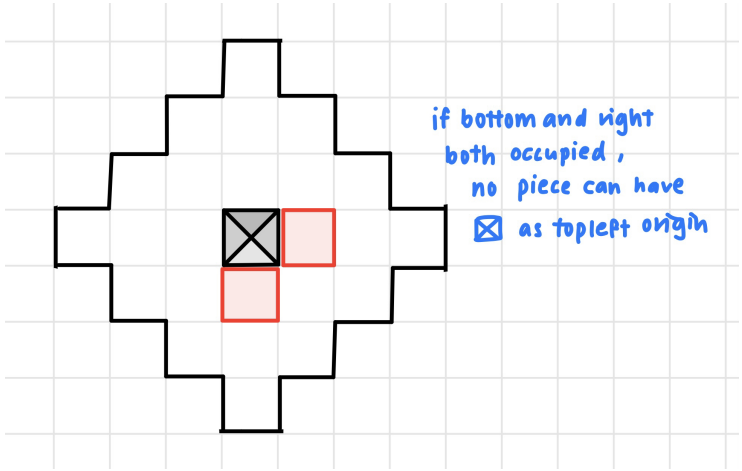

Figure 2: No piece can have a top-left origin if bottom and right cell is occupied

- After numerous iterations, CHECKSPACE was not effective, as mapping all surrounding cells from each empty free_adj_color was computationally taxing. We combined our existing knowledge and made a 2D array of all empty cells on the board and all possible piece types.
- It updates continuously, indexing its origin and the piece type being tested, and checking whether the move was valid. After testing, the array entry is removed to prevent testing the same action twice. It runs continuously until the maximum branching factor is achieved for valid actions, or until all array entries were exhausted.The array would update continuously, indexing its origin and the piece type being tested, and indicating whether the move was valid.
- After testing, the array entry would be removed to ensure the same action was not tested twice. This testing would run continuously until the maximum branching factor was achieved for valid actions, or until all array entries were exhausted.

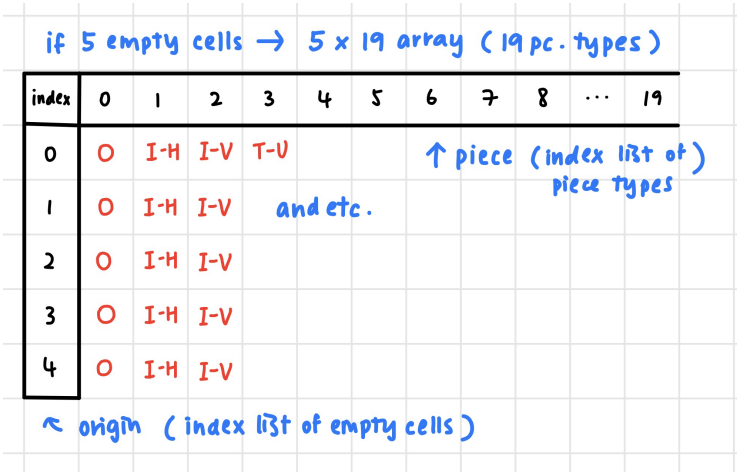

Figure 3: Array of Pieces

Supporting Work

Bash Script

To evaluate the performance of our algorithm, we need to run different agents against each other many times. We can run the function using the command:

```
python -m referee <red module> <blue module>
```

However, calling this function in the terminal many times and manually recording the number of wins for each agent might not be the most optimal way to do it.

- To solve this problem, we created a bash script that can run the selected agent with multiple-game splits.
- In our run.sh file, it is noted that we run our selected agents in 50-game splits and the result is then outputted into a text file in the folder.
- We then check the output of the text file in order to find the winningest agent.

This was referenced from EdStem Thread #219: Cohort Condition