CS 470, Section 001
February 9, 2022
Matt Christensen
Sihnyoung Park
Carter Madsen

Reversi Artificial Intelligence Writeup

Implementation:
https://github.com/mrchristensen/ReversiAI/blob/master/ReversiBot_Python3/reversi_bot.py

Algorithm Analysis

The first essential part of our Reversi Artificial Intelligence was to implement a minimax algorithm, which essentially explores the potential state space of the game by exploring the possible boards for the list of possible boards. We are able to iterate through various depths of exploration by simulating the following possible moves for the opponent recursively. Finally, by representing each depth in the context of a maximizing player (who will pick the largest score) and the minimizing player (vice versa), we can reason which initial move would be best for us. A notable weakness to consider with this algorithm is the fragility of our assumptions. Primarily, we are assuming that the opponent will act in a logical way, assuming that they make the decision that will benefit them most.

Additionally, we extended the minimax algorithm with alpha–beta pruning. While effective, the minimax max solution does not scale well for a game as complex as reversi. It has a time complexity of O(b^d), where b is the number of possible moves (the branching factor). With alpha-beta pruning we prune branches and sections of our game tree by recognizing when those areas are irrelevant. The pruning that the alpha-beta algorithm provides allows us to cut the time complexity down to O(b^(d/2)) because we essentially prune half the tree (on average).

In order to explore future game states, we wrote custom functionality to simulate the state of the board after making each of the moves specified by minimax & alpha-beta. This functionality allowed us to analyze every piece on the board and every move along the way, giving us plenty of data to work with in our heuristic evaluation!

For said heuristic evaluation, we used a combination of four elements:

- The ratio of our score to our opponent's score
- The ratio of mobility (how many moves were available to us vs. how many were available to our opponent)
- The ratio of corner squares taken by each player
- The ratio of X and C squares taken by each player.

We experimented with weighting each of these values differently to see which weights gave us the best results.

To make the heuristic evaluation for our algorithm, calculating the score of the difference between a player and an enemy is the first step. So while we call our minmax alpha-beta pruning function and figure out which states to go, we calculate the score by simply subtracting the player's score - the enemy's score. We realized that mobility, x and c squares, and corners of the board are also important components of the program. Therefore to give consistency for calculating each components and weight them easier, we decided to change our score formula to ratio which is 100 * (our score - enemy score) / (our score + enemy score). In addition, even though mobility, x and c squares, and corners are important, the main component is the score so we gave this evaluation a weight of 1.

Let's assume that scores of a player and an enemy are the same. How can a player make a better decision to win the game? That's why we need mobility in heuristic evaluation. In that situation, a player can make a move that opens up more other moves should be favored. We need to measure the potential mobility for a player's and an enemy's as well. To get an enemy's mobility, we made another get_mobility function for the enemy. Then, we calculate the ratio of mobility using this formula: 100 * (our mobility - enemy mobility) / (our mobility + enemy mobility).

We did some experiments on which component is more important between score and mobility. In the first experiment, we let the program fight each other but one just returns score for heuristic evaluation and another returns score + mobility. The result was the AI with mobility won whenever they fought each other. Through this experiment, we knew that mobility affects the algorithm. In the second experiment we weighted the mobility differently. so we ran our program AI with score + .5 mobility and AI with score + 2 mobility. The result was the AI with less weighted mobility won. According to the second experiment, we can conclude that mobility is the important component to make good heuristic evaluation; however, score is more important than mobility for the heuristic evaluation. Therefore we gave this evaluation a weight of .5 which is less than the score.

The corner squares were interesting. We knew that corner pieces could not be captured, and as such were very useful in increasing our overall score at the end of the game! Our Heuristic evaluated the number of corner pieces owned by us vs the number owned by the opponent after each move, and returned a ratio of pieces owned by us to pieces owned by the opponent. This way, if a move could lead to our opponent capturing more corners than us, the total heuristic score was lowered- decreasing the chances of that move being chosen. We ended up giving the corner portion of our heuristic a weight of 2 after our testing.

X and C squares, defined as all squares adjacent to a corner, are the stepping stones to taking a corner piece. If one player places a piece in one of those squares, their opponents can now take the adjacent corner! Since we want the corners for ourselves, we had to instruct our AI to avoid taking the X and C squares- if our opponent takes one of those squares themselves, our corner evaluation would jump into action & snap up the corner piece from our opponent. We again evaluated this as a ratio of X and C squares owned by us vs by our opponents, but returned a negative value instead of a positive value- as such, the more X and C squares we would own than our opponent after a given move, the less likely our algorithm will be to make that move. We gave this evaluation a weight of 0.5 after testing.

We'd like to further refine the evaluation for corner pieces as well as X and C squares- for example, after a corner piece has been taken by a player, we should no longer let that corner piece factor into our evaluation. We may be better off trying to capture one of the two adjacent corners so as to ensure that our opponent cannot capture an entire row! Likewise, once a corner has been captured, the X and C squares surrounding it are not nearly as important to our evaluation as they were before being captured. We have not decided how to refine this portion of our heuristic, but it definitely warrants further testing & analysis!

It's also worth noting that our algorithm's performance differed dramatically depending on the maximum depth it dove into the tree. A depth of 3 or 5, which were generally fast enough to use in 3-minute games, gave mediocre results- however, when we gave it a depth of 4, the algorithm really took off & started to smoke everything it went up against. We think that when we look so far ahead our premise of the opponent always acting logically becomes too fragile and leaves us vulnerable to a less predictable opponent. It is also worth noting that our premise of our opponent acting logically is in relation to our heuristic function. If the opponent doesn't value the same things our heuristic function does we will not perform as well as we expect.

In conclusion, we were impressed with the difficulty and importance in balancing the different weights of our heuristic function. Even though we understand that each piece of the heuristic is important and valuable, when they were unbalanced the AI performed very poorly. We also found it very difficult to test our AI in meaningful ways. Seeing as how minimax relies on predicting our opponent, we found it difficult to empirically test the AI against an opponent that thought differently than our implementation.

Time Spent (approximate)

Matt Christensen: 9.5 hrs
Sihnyoung Park: 6.5 hrs
Carter Madsen: 6 hrs