

Clay Coleman

Eric Fortney

CS 470

## Reversi Lab Report

### **Part A: Minimax Search, Alpha-Beta Pruning, and Evaluation Heuristic**

As soon as the AI receives the updated state from the server, it creates a full tree of potential moves, alternating between moves we could take and moves the opponent could take. This tree is necessary for minimax search, which allows each node to choose a move either maximizing or minimizing the possible scores of its children. Each child node is a potential move if the immediate parent node's move were taken by the other player. Each child is created by finding the possible moves from a parent node. The value of a given child is determined by a score assigned to the child's move in addition to the impact of all of its successive ancestors. Every position on the board has a predetermined weight, and the sum of each position taken by a potential move plus the value of its parent node represents the move's overall score. Additionally, if a move causes new guaranteed safe spaces to be seized, the move holds considerably more weight. The algorithm used to determine safe spaces is detailed below in Part C.

Pruning follows a traditional alpha-beta pruning system on a minimax algorithm. The child-most nodes hold the composite scores of itself plus all successive ancestors. The tree of possible moves is navigated from the root attempting to maximize the score of the current player. The algorithm assumes each player's successive move will be the one that maximizes their own score, and with this assumption, our algorithm can prune moves that are ideal for one player but will always be avoided by the opposing player. With this assumption, the algorithm can effectively prune a significant portion of the possible state tree when it determines a branch would not be explored by its logically acting opponent. To ensure that our pruning function works correctly, we created a test

class (TestPruning.java) which implemented the same tree as our pruning homework, and showed that it returned the right value and pruned in the proper places.

## **Part B: Time Spent on Lab**

*Clay Coleman: 8 hours*

*Eric Fortney: 8 hours*

## **Part C: Algorithm Evaluation**

Our heuristic evaluation function employs the several creative techniques, detailed below. Each addition eventually made our algorithm more effective, though initial implementations led to poorer performance.

### Position Weights with Dynamic Values

#### **Description**

Each board position has an assigned weight value for its corresponding value in the game. This heuristic allowed our agent to determine which locations of the board it should jump to seize, and others it should avoid taking. For example, the corner positions are weighted very highly in our agent, as they maximizes its chance to capture significant portions of the board permanently. In contrast, the positions that immediately border corners are generally disadvantageous to seize, and are thus negatively weighted in our algorithm.

#### **Variations**

Initially we were weighing all edges positively, and all positions one-in from the edge negatively. We realized this didn't work as our algorithm would too frequently take the edge pieces next to corners. Thus, we only weighted those

squares around the corners negatively. This ended up significantly improving our agent's performance.

We also saw that our agent was avoiding the locations around a corner, even after seizing it – which was preventing it from seizing large portions of safe spaces. To avoid this, we implemented some dynamic weight alterations, which changes the weights around the corners to be positive once the corner has been seized. This enabled our agent to quickly establish dominance in corners of the map more effectively than before.

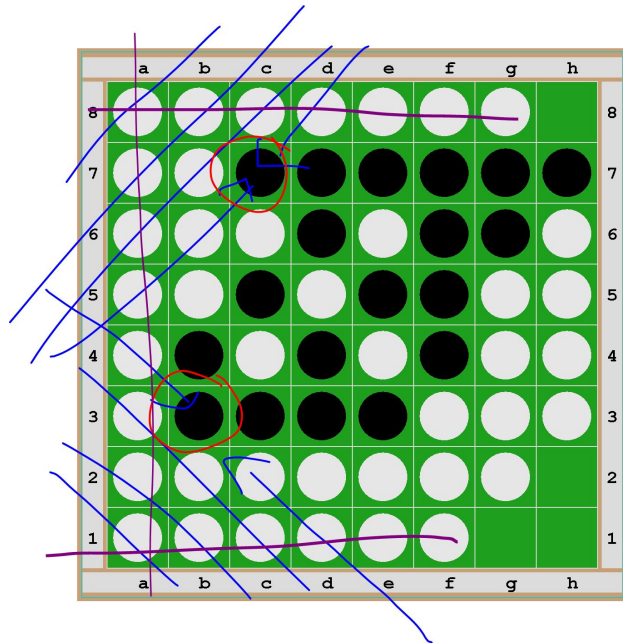
### Discovering and Maximizing Safe Spaces

#### **Description**

When a corner position is held by a player, that player potentially has seized several “safe positions” branching out from that corner. A safe position is a position that cannot be captured by the opponent. For example, all edge pieces owned by a player that have a direct sequence of tokens to a captured corner are guaranteed safe. Additionally, all 45, 45, 90 triangles of a player's tokens that originate from the right angle at the corner are guaranteed safe. To determine this, we implemented an algorithm to scan outwards from the corners owned by a player in a diagonal line from the given corner towards the opposite one, tracing a number of paths across the board perpendicular to the previous diagonal (see the image below – demonstrates this principle very clearly). The algorithm traces paths and tallies up safe spaces until it encounters an opponent token. The tracing is carried out from top-edge to side-edge and from side-edge to top edge so that if/when an opponent token is encountered, the rest of the valid tokens along the same diagonal are properly considered.

Take the given example below. White owns two corners and traces diagonal lines (in blue) outward from the corner until it encounters an opposing token at (c, 7) and (b, 3). It then sums up the safe straight lines as seen in purple

and adds it to the tally. The final tally is formed by getting the size of the set of valid indices.



Our evaluation heuristic considers these safe places when evaluating move scores, allowing it to favor moves that will ultimately cause more and more safe moves to be captured.

To verify this heuristic worked as expected, we created an additional class, `TestSafeNodes.java`, with a variety of specific input on a dummy board. We then compared the result of the heuristic function within that controlled environment and verified it produced the anticipated results.

### **Variations**

Initially, we were weighing every move by the raw number of safe places it resulted in. However, we realized that this was preventing our agent from making certain moves up front that were necessary. Instead, we decided to weight the number of additional safe spaces created by a move. This gave our algorithm more insight into which moves would drive the most value at the proper times.

### Aggressive Elimination and Self-Preservation

## Description

During our tuning of our algorithm, we discovered that our algorithms would sometimes play into situations that would ultimately end allow the other player to capture all of our agent's tokens, thus preemptively ending the game (this was mainly due to a bug in our tree building algorithm, which we promptly fixed). To this end, our algorithm stores the total number of tokens for each player after each potential moves. If any given move causes the opponent to own only 3 tokens or less, then that move is aggressively weighted as very effective – allowing our algorithm to center in on killing moves and end the game preemptively. Similarly, if a move by an opponent reduces the agent's token count below 3, then it is heavily weighted in the negative direction to avoid these moves at all cost and avoid being wiped out.

## Variations

We had experimented with different number of tokens to represent the “danger zone”. Initially we had it set at 0, but we found that the moves leading up to extinction were not being discovered fast enough to play a significant role in the algorithm. Thus, we upped the number of tokens to 5, which ultimately ended up focusing the agent too highly on these moves instead of on other, more valuable moves.







## Algorithm Performance and Efficiency

The following chart demonstrates the average time the computer player takes to play a full match against an identical AI of the same depth according to the in-game clock.

Depth	Time
1	>1 second

2	>1 second
3	1 second
4	2 seconds
5	4 seconds
6	20 seconds
7	4 minutes 7 seconds

As of a depth of 7 onward our test computer's RAM began to be overloaded leading the program to store state on disc causing a significant bottleneck. The algorithm consistently did better against versions of itself with less depth--allowing it more time to operate does seem to directly improve its performance.

Name	Status	50% CPU	95% Memory	0% Disk	0% Network
>  Java(TM) Platform SE binary (2)		45.4%	3,980.4 MB	0 MB/s	0 Mbps
>  Java(TM) Platform SE binary (2)		0.3%	3,937.3 MB	0 MB/s	0 Mbps
>  Google Chrome (21)		0%	1,289.1 MB	0.1 MB/s	0 Mbps
>  Slack (7)		0.5%	382.4 MB	0.2 MB/s	0 Mbps
>  Visual Studio Code (32 bit) (14)		0%	352.7 MB	0 MB/s	0 Mbps
>  Spotify (32 bit) (4)		0%	195.3 MB	0 MB/s	0 Mbps

## Algorithm Effectiveness

We feel our agent is quite strong. It regularly beat ourselves and our roommates. It frequently eliminates random players before reaching a full board state. Even at a tree-building depth of just one, our agent consistently beat the random player. To determine its current system of weights, we played several rounds against ourselves and the included 'RandomGuy' player, changing the

weights as we maximized how often it won. We are eager to face off against other students' AI systems and see how it fares!

### **Future Improvements and Learnings**

A direction we can take to improve upon this is to directly pit the AI against a nearly identical one where the sole exception is the weight assignments to identify optimal combinations. Additional aspects we'd like to investigate is how we can more effectively store state, so as we do not encounter a bottleneck as we saw from a depth of 7 onward. Another approach would be to investigate how to most effectively set our weights for a number of different metrics that affect the overall score of a given turn (eg: safe space points, corner positions, edge positions, etc).