Final Project Specification: DRAFT

Michelle Cone | Theresa Gebert | Yuan Jiang

mcone@college.harvard.edu | tgebert@college.harvard.edu | yuanjiang@college.harvard.edu

1 Brief Overview

We would like to write a game solver for Blokus. We plan to compare and contrast several different algorithms that have been used in the past for other game solvers. Our goals for the project are to find a good algorithm for solving a two-player Blokus (and similar games) by implementing different algorithms and letting them play against each other.

2 Feature List

Core Features:

1. As a "control" algorithm, we will be implementing a random Blokus player. It will choose a random valid move to play. We do not expect this algorithm to do very well, but it will act as a baseline to test our other algorithms against. Practically speaking from our project's perspective, it will give us the chance to develop our user interface.
2. We will also implement a greedy algorithm (Monte Carlo method) for our Blokus solver and compare it to the random method. There is an example online of a Blokus solver which uses a Monte Carlo method. For any given turn, our algorithm with return a subset of all possible moves with the remaining pieces, and then pick the best move. We will choose

our subset smartly by giving greater priority to pieces with higher area. In addition, the "best" move will be define by how much additional area the piece covers, how many new corners it supplies the board, and its Manhattan distance to the center of the board. We can imagine all possible Blokus moves as a branching tree, where each branch represents a possible move. The reason we are only looking at a "subset" of all the branches is due to the fact that there are many pieces, and each piece has many different rotations as well as positions to be placed on the board. This may be very computationally expensive, so we must compromise by taking a random subset.

3. Finally, we will implement a Minimax Search with α-β Pruning. Consider a Minimax Search without α-β Pruning for a moment. The basic principle is that every sequence of potential moves in the game is assigned a point value dependent on an evaluation function of the board. For Blokus specifically, our evaluation function, which maps a formation on the board to a score, would consider the number of squares that have been placed on the board and the number of opponent's moves that have been impeded. A Minimax Search will consider each sequence of moves as the leaves of a tree and each level i of the tree as the possible moves at turn $(i+1)\text{mod}2$ for Player 1 if i is odd and Player 2 if i is even. One player, let's say Player 1, will seek to maximize these point values at each level of the tree (i.e. make choices that will increase their chances of winning) while the other player, Player 2, will seek to minimize these values (i.e. make choices that will decrease their opponent's chances of winning). The algorithm will retrace each sequence of steps from the leaves of the tree assuming that each player plays rationally until it has "filled in" all nodes of the tree and has reached the root. As a result, the algorithm will be able to tell Player 1 which sequence of moves in order to earn the highest point value of the evaluation function. Notice that the success of this function is very much dependent on the quality of the evaluation function we write! Finally, α-β Pruning is, at a high level, a way to reduce the amount of calculation required to implement a Minimax Search. Particularly in a game like Blokus, with several possible pieces to be played every turn of the game, such a tree might be computationally difficult. α-β Pruning "prunes" the tree by recognizing when certain branches can be cut

because they are not desirable moves.

Cool Extensions:

1. It would be interesting to implement a feature where a user could play against the computer. For this, we would need provide a feature where the program can take user input. We would enumerate all the pieces, and then just have the user input a number, a rotation, and a coordinate/position on the board.

2. Another thing we could possibly implement in our scoring function is adding a factor that takes into account how many corners a move can block the other player's "frontier" (corners).

3. Ideally, we would end up giving our project a much more appealing visual interface and put it up on a webpage. This would be ideal for presentation and accessibility purposes.

3 Technical Specification

Modularity

- One obvious way to modularize our project is to separate out the different algorithms we would like to implement: Random, Greedy MC, and Minimax.
- In order to achieve this, we need to decide how to represent the board and the various pieces. For our project, we will be implementing the two-player (one color each) version of Blokus. Once we have decided on the UI, we will be able to independently write algorithms which can interact with the common UI.
- Our common UI will keep track of board state: open spaces, occupied spaces, and if a space is occupied, which player is occupying it. We will also have a visual representation of the board: some sort of print function that will allow us to see what each configuration looks like. Part of our UI will also be to keep track of the types of pieces (i.e. shape and size) that each player has left.
- For Random, we will need these functions: a function that picks a random piece with the current highest possible area, and then returns a random state for that piece
- For Greedy MC, we will need these functions: a function that picks a random subset of the biggest pieces, and then returns all possible moves/states of that pieces; a function that calculates a "score" for that moves based on area, number of new corners, and Manhattan distance to the center (may need a function for this as well), and returns the highest scoring move
- For Minimax, we will need these functions: a function that finds the number of the possible moves for a given player, a function that finds the number of squares placed on the board for a given player, a carefully defined evaluation function that can take in the current configuration of the board that returns a score (for Player 1), which is a combination of the number of possible moves for each player and the number of squares placed for each player. Remember that we would like to minimize the number of possible moves of the opponent (Player

2), and maximize the number of possible moves for Player 1. In addition, notice that we can alter our evaluation function (based on, for example, how much we would like to weight gaining moves vs. preventing opponent's moves) and play different versions of Minimax against each other to find the optimal weighting.

Interfaces

- We will be implementing the back end of this project completely in Python using iPython Notebook. Some useful libraries include NumPy, SciPy, and Pygame (important). Should we have enough time to put our project online, we will be using Javascript and HTML.

4 Next Steps

1. We will be implementing the back end of our project in Python.

2. We will be setting up a GitHub and Dropbox in order to share our work.

3. We will be using iPython Notebook to comment and compile our code. We chose to use this because it is particularly useful when presenting a project and also allows us to incorporate user input by presenting our work, results, and then allowing users to try the game themselves. We would like to do our final writeup in iPython for these reasons.

4. By next week, Sunday, April 20th, we would like to have at least finished coding a playable game board, and finished writing functions for calculating the "score" for any given move.