Adversarial Search in Gomoku

Janelle Bakey

Chandler Severson

Southern Oregon University  
bakeyj@sou.edu seversonc@sou.edu

# INTRODUCTION

# FOCUS

The focus of this project was to create a program which could play Gomoku. More specifically, we strived to create a program that could make the best possible move available within a 2 second time constraint. Our objective honed in on using MiniMax search methods paired with Alpha-Beta pruning to increase the speed at which we could search for the best possible move.

# APPROACH

## Game State

Our game state (GameState.java) consists of taking in the state provided by the server and parsing it appropriately to decipher which player we are, what the game state is, the size of the board, and what pieces have been placed on the board. We verify whether or not our opponent has missed their turn because in the case where they have not made a move, our program is able to save time by using the last saved game state. Our program keeps a move count and uses this to know whether it is our turn or not. Our program then moves on to calculate the players’ utility to find the best move possible.

## Player Utility

Also located within our game state class, our program defines the utility for a specified player in a specified state. As the program checks each space, it looks for game pieces belonging to both players. Once it finds a game piece belonging to the opponent, it stores this so that it may refer back to the last encountered opponent game piece. Once the program finds a space that contains the specified players own game piece, it verifies whether there are enough spaces to play in a way that could eventually lead to a win (5 spaces). The program checks to the right, then below, then above, then bottom-right diagonally, and then top-right diagonally. If this is true, it checks to see how many game pieces the specified player currently has within those 5 spaces. Based upon these evaluations we rank the utility value for the given player. If there are less than three game pieces or 5 game pieces, the amount of pieces is added to the utility value; if there are three pieces, three is added to the value; if there are 4 pieces and the player has not run into an opponent’s piece, four points are added to the value; else if the opponent is blocking the player at the end of the board, continue on to the next row; else if there are 4 pieces, check to see if a fifth piece can be played to win the game. By placing values on each spaces utility, the program can then quickly choose which space provides the highest utility for the specified player.

## Move Search

Taking the current game board, our program implements the MiniMax algorithm for adversarial search (MoveSearch.java) and returns the best move currently possible, given it finishes within the time constraints. The program iterates throughout the size of the board as the search deepens. In the MiniMax search methods, we both Maxamize as well as Minimize the expected outcomes and then use Alpha-Beta pruning to help speed up the searches performed. If our player is unable to finish the search in the allotted time, we use our last best saved moved from the previous search.

# IMPLEMENTATION

We created a sample game state (SearchTester.java) to analyze the board and test our search methods. Through our implementation we do see that our program is using the minimax search methods as well as implementing Alpha-Beta pruning. We also note that the program takes about 5 seconds to search and find the best possible move available, which is 3 seconds past our 2 second time limit. As 5 seconds will not give us the best possible move available, perhaps reworking our search layout could present a thorough search that is closer to the allotted 2 seconds.