Zach Cora

Austin Moore

Vivek Bigelow

B351 Project: Gomoku Opponent

The game of Tic-Tac-Toe is a simple exercise. One player writes an X or an O on a 3x3 board and the other player places their own different letter. The players alternate until one of them achieves three in a row or the board is full. Now imagine that instead of a 3x3 board, we have a 15x15 board. Also, instead of 3 we now require 5 in a row to win the game. This is the game of Freestyle Gomoku. Gomoku is an evolution of the game board Go. It has been known in many different variations and with many different rules throughout the ages and between different countries. Freestyle gomoku operates in the way we previously described - and this will be the topic of focus for our AI implementation. We chose to focus on this method for two reasons: firstly the simplicity of the rules and the ease of creating our own game (as we were not able to find code available online in Python, we were required to make our own), and secondly because the AI approach for Freestyle Gomoku is more simple. It has only one win condition and few limitations whereas variations have moves that are not permitted and also in some cases (Pente), secondary win conditions.

We first decided to try something that would run quickly, so we created a value matrix that would be tracked and updated by the game board. This AI is based on an article from generation5.org, titled *Simple Board Game AI*. The player class that uses this, dubbed Computer, will search for spots on said matrix with the highest value; if there are multiple locations with the highest value, it chooses randomly amongst them. In order to increment the values on the matrix, we started off by creating one update function, now named update1, that, when a move is made, increases the value of all places on the board where that spot has influence. This includes up to four spaces in each of the eight directions (vertically, horizontally, diagonally). This particular update function was written from scratch three times, to fix implementations that did not work quite how they should, and were just not coded well. In the latest implementation, we added threats to this function. It detects when a player creates a threat of three or four tokens that have the ability to create a 5-in-a-row, and increases the value of spots that will block/take advantage of these threats. Then came update2, our defensive update, which removes a token’s influence if an opposing piece blocks it in a particular direction. This second update also needed to be completely rewritten due in incompatibility with new update functionalities. While testing, we came upon the problem that, when a piece’s influence was not blocked, but a 5-in-a-row was no longer possible, the value needed to decrease; that’s what update3, our least-used update is for. Our last update function, update4, was implemented rather late. This update function determines when a threat created by update1 has decreased in value due to one side of the threat being blocked by the opponent, and decreases the value of said threat accordingly. This function should have been implemented within update2, as that is our defensive update, but our current update2 is not compatible with this function and would need to be rewritten yet again.

After implementing the value-matrix player, we decided to implement the Monte Carlo Tree Search algorithm to make another player that might be better than the original. We chose the Monte Carlo algorithm because of its success with other games with large boards, like Go. Our main two sources for researching and implementing the Monte Carlo player were the Monte Carlo Tree Search wikipedia page and the survey: *A Survey of Monte Carlo Tree Search Methods.*

The Monte Carlo Tree Search algorithm takes each available position of a game state and plays several simulated games based on a single position being chosen. After the initial chosen position, the simulated games are played out through random moves by each player (in two player games). The power of the Monte Carlo Tree Search algorithm comes from the data obtained through large amounts of simulations, and the user’s ability to stop the searching and receive the best move the A.I. could find in the given amount of time.

For our implementation, we combined the value system with the Monte Carlo algorithm. Instead of simulating games over every available position in a given game state, the algorithm simulates games over the five positions with the highest values. By reducing the number of positions the Monte Carlo player looked, at we were able to have more simulations over the chosen positions. We also included a cap of 3.5 seconds, after which the algorithm would stop and return the best move it had found so far. However, our implementation was much slower than our initial A.I. and it did not play any better. We believe that the main problem was that the Monte Carlo Player was unable to go through enough simulations. Each move had fewer than 1,000 simulations, where a good number would be from 10,000 - 20,000. Something we think would improve the Monte Carlo player would be to implement threading or parallelization so that each move was being simulated at the same time to increase the number of simulations.

One final look at AI development for Gomoku goes into Alpha-Beta search. Our initial algorithm using a heuristic to determine the value of board positions ran so quickly, it seemed that we could easily run an Alpha-Beta algorithm to achieve a few plies of depth and better our move choice greatly. However, once we took into account the branching factor and the processing that was needed at each point to update the value board, it soon became clear that Alpha-Beta search would not be of much use in its most simple form. Also, because of the need to copy the value board in each successive iteration of the Alpha-Beta search, complexity was sure to become a problem. Writing another heuristic to evaluate the board state would have been similar if not higher complexity. For these reasons and due to time constraints we chose not to implement it into our Gomoku AI.

Our value matrix AI player works surprisingly well. It also can play an entire game in less than one tenth of a second, so it’s lightning fast. After implementation of threat search and more advanced update methods, this Computer player has a nearly impeccable defense. On offense, however, it is not perfect. It will win against most opponents without too much experience with the game. An experienced player will beat the Computer player with careful thought. The Monte Carlo player is not nearly as good as Computer. Since it runs better with more allotted time, this AI take a very long time to decide on a move. Wanting it to run faster, we have it set at a 3.5 seconds time limit, but even with 30 seconds, it’s not as good as the Computer player. With more time, we would have liked to use a selection algorithm for the board to limit a branching factor on an Alpha-Beta search. Also, changing the implementation so that side effects were not incurred on the board of values would have let us use it in a recursive manner inside the Alpha-Beta search function. As it was, copying the board each time consumed too many resources.

Works Cited

Browne, Cameron B., Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I.

Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. "A Survey of Monte Carlo Tree Search Methods." *IEEE Trans. Comput. Intell. AI Games IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012): 1-43. Web.

"Monte Carlo Tree Search" *Wikipedia*. Wikimedia Foundation, n.d. Web. 09 Dec. 2015.

“Simple Board Game AI”

James Matthews

http://www.generation5.org/content/2000/boardai.asp