CS221 Project Final Report
An Intelligent, Artificial Look into Twixt

Bardia Beigi {bardia}, Soroosh Hemmati {shemmati}, Michael Painter {mp703}

Dec. 16, 2016

Introduction

In this project, we explored implementing multiple intelligent agents playing Twixt. Twixt is a board game with a huge state space and branching factor. The rules of the game and its setup also allow for modifying the size of the board which eases implementing and checking different algorithms. In this project, we implemented and used a number of algorithms, ranging from basic minmax agents to pure Monte Carlo-based and Monte Carlo tree search-based agents. These agents were then pitted against each other, checked against a baseline of a simple minmax agent, and evaluated by playing a human expert.

Background on Twixt

Twixt is played on a board comprising a 24×24 grid of holes, without the holes in the four corners (see figure 1). Both players take turns placing their pins and are allowed to place them in any hole in the central 22×22 board. Player 0 can exclusively place pins on the top and bottom-most rows, whereas player 1 has exclusive access to the leftmost and rightmost columns. If two pins of the same color (belonging to the same player) are an L move apart, i.e. a knight's move, then they may be connected using a bridge. The goal of each player is to connect their exclusive regions using a connected number of bridges as shown in figure 1. Note that we have

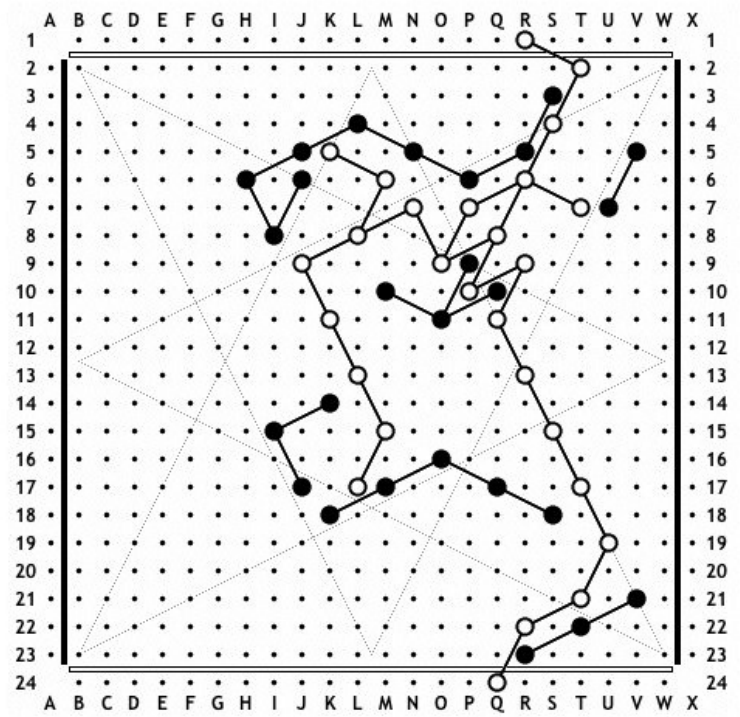

Figure 1: Standard Twixt board. Player 0 (white) has won.

simplified the rules of the game for this project. Particularly, in official rules of the game, each player is allowed to not place a bridge even if their pins are exactly a knight's move away. In addition, a player is allowed to rearrange their bridges during their turn if they see fit. In our model, however, we have taken these options away from the player, i.e., upon placing a new pin,

all newly available bridges will automatically be placed on the table and under no circumstance removed until the end of the game.

Literature Review

There is not a lot of articles written about this game. The few articles available highlight the large state space and branching factor of the game [1, 2, 3]. Twixt is second only to Go in terms of game complexity, and coming up with a reasonable AI agent is a challenging endeavor.

The recent advances in AI (especially AlphaGo [4]) have proven that combining a probabilistic approach (such as Monte Carlo analysis) along with Machine Learning in the form of reinforcement learning or supervised learning could lead to enormous optimizations in games with extremely large state space and branching factor.

More generally, Monte Carlo Tree Search [5] is currently the dominant algorithm used in game playing, and has a lot of literature surrounding it [6, 7], especially in the game of Go [8, 4].

Game Model

Twixt is a zero-sum game, with no immediate reward at each step and only a final reward if the agent wins, like chess. As a result, we have modelled the game similar to the Pacman assignment.

- **States:** Intuitively, one might consider the position of the pins and bridges as the state of the game, and that is exactly what we have done. However, due to the huge state-space of the game and to simplify evaluating/assigning a heuristic score to each state, we opted to store the last action taken by the agents as well. This will be explained in more detail later.
- **Actions:** Again, due to our simplification of the rules, once a player decides on the position of their next pin, the program should deterministically decide on which bridges should be placed. As a result, an action is simply the position of a new pin.
- **Start State:** The start state is simply the state with no pins placed on the table (empty grid).
- **End States:** End states are the cases in which either player has won or no player could possibly win (draw). It is a challenging task to code whether a state is in fact a draw if there are still empty holes on the table. As a result, we have decided to let a full table with no winner be a draw state.
- **Evaluation Function:** An evaluation function is highly dependent on the type of agent being used and will be explained appropriately in the following section.

Implementation

The structure of our code is similar to that of the Pacman assignment, with a class containing all the necessary code to describe the grid, states of the game, and all necessary functionality to decide whether an action is allowed, how the board should be updated, and how a successor state should be generated. There are a few minor differences in our game modelling and algorithm implementation, however, which have a major impact on the run time of our code and will be explained in detail here.

- **Score Calculation:** As mentioned before, we have introduced a variable to keep track of the last action each agent took. This is because of the fact that calculating a score over

a state (pins and bridges) can be very computationally expensive if there are a lot of pins and bridges placed on the table. As a result, it would be more efficient to store the score corresponding to every state once a new action has been taken as state-score, and to add the incremental change in the score the next time an agent takes an action.

- **Evaluation Function:** The evaluation function implemented to estimate the likelihood of an advantageous board position is as follows: an advantageous board position is described as the one with the most number of 'linked' columns for the AI agent. A 'linked' column is defined as a column on the board where the longest link either starts at, ends at, or passes. For instance, consider the instance of the game below:

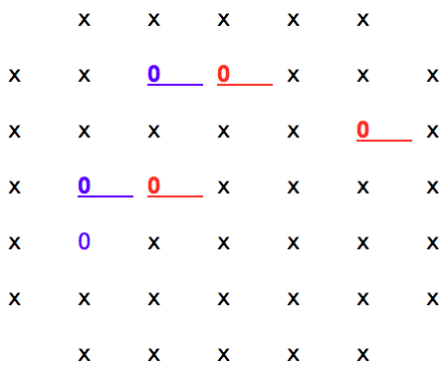

Figure 2: Instance of a Game

The pins for player 0 (human) are denoted by blue O's, and the pins for player 1 (AI agent) are denoted by red O's. Bridges are denoted by underlines on the connecting pins. In this instance, the AI agent (red player) has 4 linked columns which are columns 3, 4, 5, and 6 (from left). Thus, the evaluation function would return the value of 4. In the same instance, the human player (blue) who's going north to south, has 3 linked rows, which are rows 2, 3, and 4 (from top). Please note that row 5 is not considered a linked row as it is not linked to any other blue pins. In this case, the evaluation function of the blue player would return the value 3.
Note that there are many other features to use to calculate the score properly, such as starting close to the center, not placing pins too close to the opponent's pins, creating as many bridges as possible, or creating bi-directional chances. However, the approach explained above proved to be the simplest and most useful for a minimax agent.

Algorithm:

Several agents were developed to play the game on boards of different sizes. This section is devoted to describe each agent in detail as well as comparing and contrasting them against each other:

Minimax Agent

One of the implemented AI agents is a minimax agent that maximizes its score assuming minimization of score from the other player. The implementation of minimax was taken with some significant changes from the Pacman assignment. Due to the large branching factor and state space of Twixt, depths 1 and 2 generally worked well and efficiently on boards of up to size

12×12. Also, it was practically impossible to get a good run time on boards of size larger than 7×7 without pruning. As a result, our minimax agent uses alpha beta pruning. We had to modify the minimax agent to give us the exact depth at which it reached an end state (if at all) to be able to take the winning action at the right time, to ensure victory as soon as possible.

Pure Monte Carlo Agent

Another developed AI agent is the pure Monte Carlo agent. The idea is that an insight about the next best move could be developed if a lot of games are played from the current state as the starting state. The aggregated data will signify the advantageous moves among all the randomly chosen moves. The advantageous moves will be highlighted by their higher probability of leading to victory for the player using this agent.

This purely probabilistic model works best if the number of iterations of the experiment is relatively high. As a result, for the sizes up to 12×12, the number of iterations was chosen to be $100,000$. Due to the large number of iterations and the large depth of the game tree, this agent takes considerably longer than Minimax to produce the next move. However, its implementation and the idea behind it are very intuitive.

Figure 3 shows the dictionary that the Pure MC agent maintains on every move. At the end of each iteration, it picks the move (pin to be placed at the location of the first element of the tuple) with the maximum number of iterations that ended in victory for the agent (first element in the sorted list).

```
iteration:  2
[((3, 4), 80), ((3, 3), 76), ((4, 4), 74), ((4, 5), 74), ((1, 3), 73), ((3, 2), 73), ((5, 3),
 73), ((4, 2), 72), ((5, 2), 69), ((4, 3), 68), ((2, 3), 67), ((2, 4), 66), ((3, 7), 65), ((3
, 5), 65), ((5, 4), 61), ((4, 7), 60), ((5, 6), 60), ((2, 5), 60), ((1, 7), 58), ((6, 2), 57)
, ((4, 1), 57), ((4, 6), 57), ((2, 2), 56), ((6, 4), 56), ((1, 1), 55), ((6, 5), 55), ((3, 6)
, 54), ((4, 0), 52), ((5, 0), 52), ((2, 7), 52), ((2, 6), 51), ((6, 1), 51), ((5, 7), 51), ((
6, 6), 50), ((1, 2), 49), ((6, 0), 49), ((1, 6), 48), ((1, 5), 48), ((1, 4), 48), ((5, 5), 47
), ((2, 1), 46), ((3, 0), 45), ((5, 1), 43), ((3, 1), 43), ((6, 7), 43), ((6, 3), 43), ((1, 0
), 43), ((2, 0), 42)]
```

Figure 3: Pure Monte Carlo Decision List

As depicted above, the most successful action to take at this state seems to be placing the next pin at location $(3, 4)$ which leads to marginally more victories than the actions $(3, 3)$, $(4, 4)$, and $(4, 5)$. One thing to note here is that this simulation was run for $20,000$ episodes, and the sum of all the victories in the depicted list is $< 5,000$, meaning that by choosing random actions the probability of ending in a draw is quite high, explained by worthless computations due to the random nature of action selection.

Monte Carlo Tree Search Agent

Monte Carlo Tree Search (MCTS) is, as an algorithm, ubiquitous within game-playing AI. The idea is similar to our 'Pure Monte Carlo Agent', but we try to be a little bit smarter.

The algorithm is used, as in the other agents, to make a decision on which action it should take given some game state. Each MCTS iteration consists of 4 steps, over which we build a partial search tree, that grows with each iteration. At each node we store a few additional pieces of meta-data, including the number of times that we have visited the node, and the average value of the game state, as witnessed from this node (see backpropogation). As we progress through the iterations, we hope that our value for each node approximates the value of that state, and hence can be used to make informed decisions on what action to take.

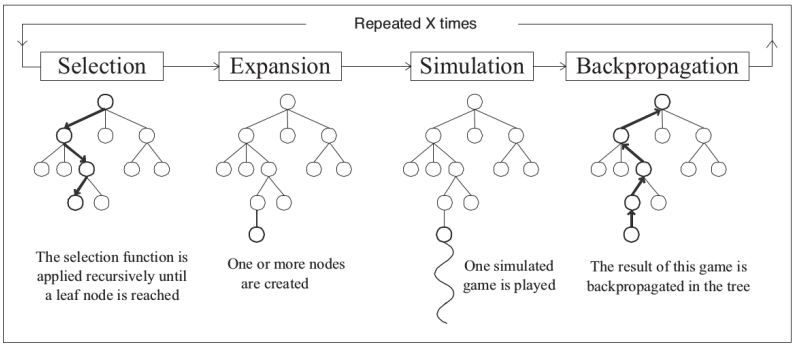

Figure 4: Outline of MCTS (Chaslot et al. 2008 [5]).

In our version of the algorithm, we need two stochastic policies to be provided, the *selection policy* and *simulation policy*. Here, a stochastic policy is a function mapping actions, from some given state, to a weight, that intuitively represents how 'good' that action is.

Selection: In the selection phase, we stochastically walk down the partial search tree until we reach a leaf node, according to the aforementioned selection policy.

Expansion: The expansion phase extends the partial search tree by one or more nodes. Our agent expands by creating one node, also by following the selection policy.

Simulation: From each expansion node, we finish playing a game, according to the simulation policy and see which player of the game won. Each action is chosen stochastically according to the policy, and there is no backtracking.

Backpropogation: When we backpropogate, we update data stored in each of the nodes on the path we traversed in the selection phase, as can be seen in figure 4. The values updated are the number of times each node has been visited, and the average value of the node.

Intuitively we see that this agent can act in the same way as our Pure MC agent, if handed uniform policies, and we sample only according to that policy. However, to encourage exploration, when we are following our selection policy π_{sel} from some state s, we sample actions a from

$$p(a|s) \propto \frac{\pi_{\text{sel}}(a;s)}{n_{s'}+1}$$

where $s' = \text{Succ}(s,a)$, the successor state of s from action a, and $n_{s'}$ is the number of times that s' has been visited, as tracked by the partial search tree.

Finally, due to the model's flexibility we can try many policies and see which does best. We very quickly discovered that a policy giving very high weights to any pins that are one or two "knight's moves" away from the agent's own pegs worked extremely well, so much so that it consistently beat all of our other agents, and our resident expert on an 8×8 board.

Results

After developing the agents, some simulations were run to identify the strengths and weaknesses of different agents.

Human Agent vs. Minimax Agent (Baseline)

Initially, we ran a simulation of a human player (blue) versus the AI agent we implemented (red) using the described Minimax agent implementation with depth 1, evaluation function, and a

board with size N=7. The following series of images depict the state of the game as the game progressed.

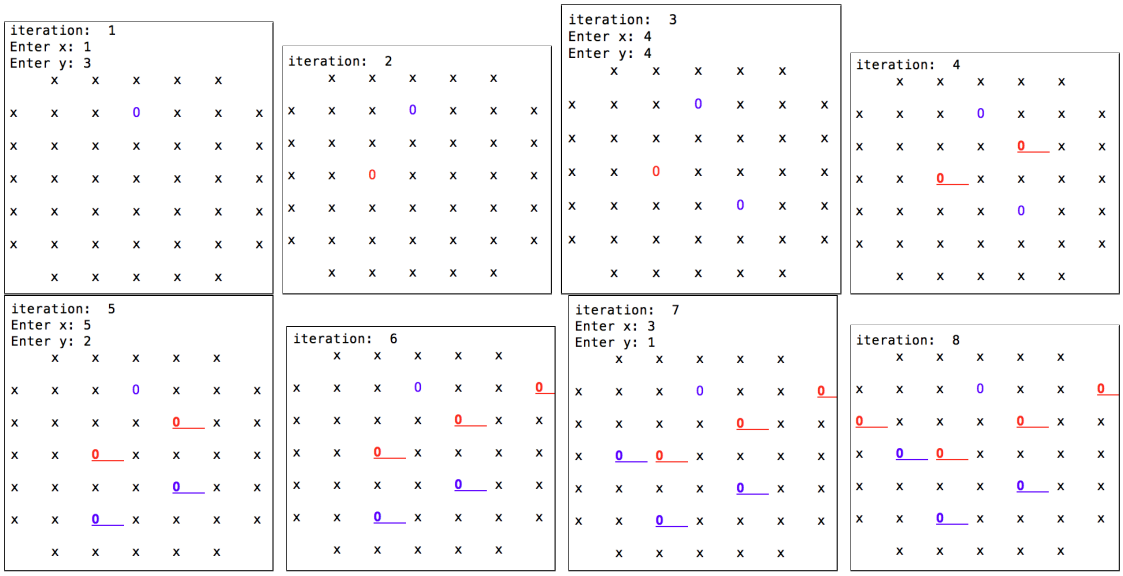

Figure 5: Human vs. Minimax Game

As seen in the images, the AI agent (in red) tries to maximize its width span by constructing wide links across the board. It does well to see the block by the human in iteration 7, to deviate and place the last pin on the first column to win the game. The AI agent was intelligent enough to beat humans new to the game with no prior experience, however it could significantly improve against humans with prior experience.

Minimax Agent vs. Pure Monte Carlo Agent

Subsequently, we ran a simulation between the Minimax agent (blue) and the Pure Monte Carlo agent (red) on an 8×8 board. The Minimax agent is setup with depth 2 (higher than the last example), and the Pure Monte Carlo agent is setup to run $50,000$ iterations of the game starting at the current state and pick the next action with the highest probability of success.

As one can note, the Minimax agent gets off to a bad start by placing the first pin in their home territory, whereas the Pure MC agent places the first pin nicely in the middle. From that point onward, the two agents play optimally by trying to maintain their respective links and make them longer. The Pure MC agent always has the upper hand in this game though, and will finally win the match.

This trend repeated many times among these two agents with the Pure MC agent being superior with the chosen depth and number of iterations. We could further improve the success rate of the Minimax agent by increasing the depth of the tree it uses to determine the outcome of the move and improving the evaluation function it uses to approximate the result of the game. We could also further improve the success rate of the Pure MC agent by increasing the number of episodes it runs per move. However, doing any optimization on either of the agents has a trade off, meaning the time needed for the agent to generate the next move would increase quickly.

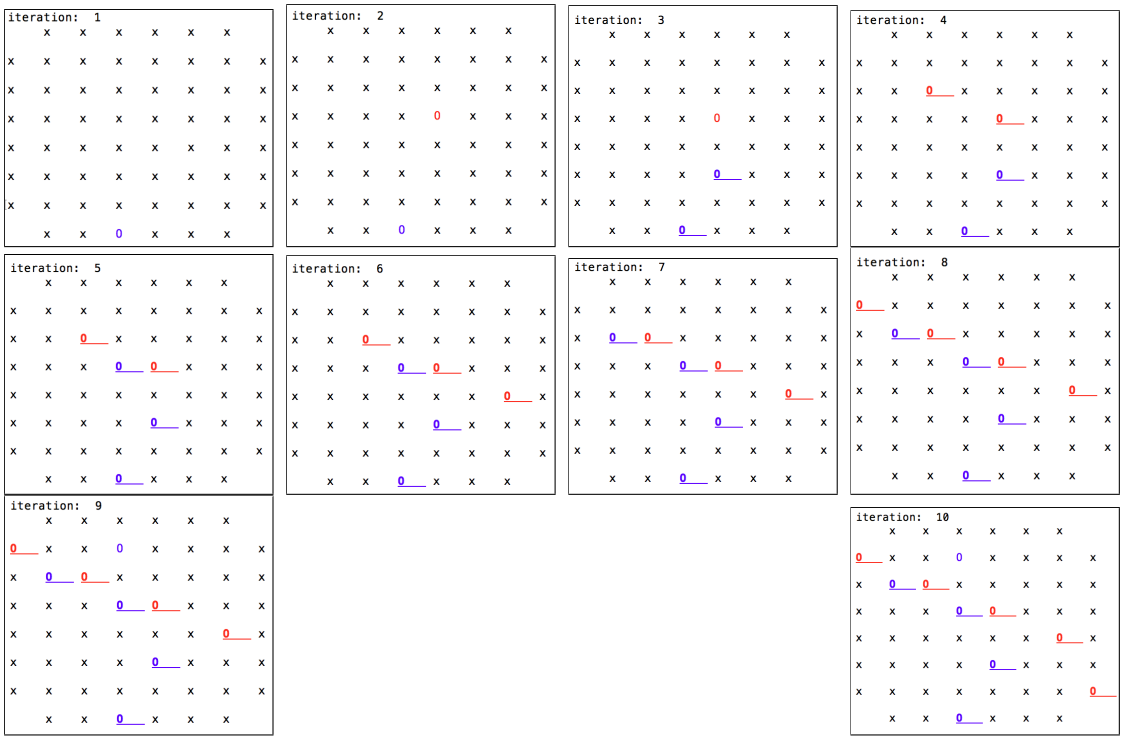

Figure 6: Minimax vs. Pure Monte Carlo Game

Monte Carlo Tree Search Agent vs. Pure MC Agent

Here we have pitched an MCTS Agent armed with 2000 iterations and the policy described prior against the Pure MC agent armed with 10000 iterations. The MCTS Agent is playing as blue, trying to go from top to bottom and Pure MC as red, going from left to right. After iteration 5, we see that the MCTS agent has blocked off the Pure MC agent, leaving it with very few advantageous moves to make, and which given that it searches randomly, is unlikely to find.

Playing a few matches, we found that giving the Pure MC agent given around a 10-fold increase in iterations over the MCTS agent led to a close match. When we allowed MCTS to have a few more iterations, it tended to win if it started first or second. However, we did also notice that either MCTS or Pure MC, given enough iterations, would play near optimally on an 8×8 board, and so for higher numbers of iterations, the winner depended only on who got to move first.

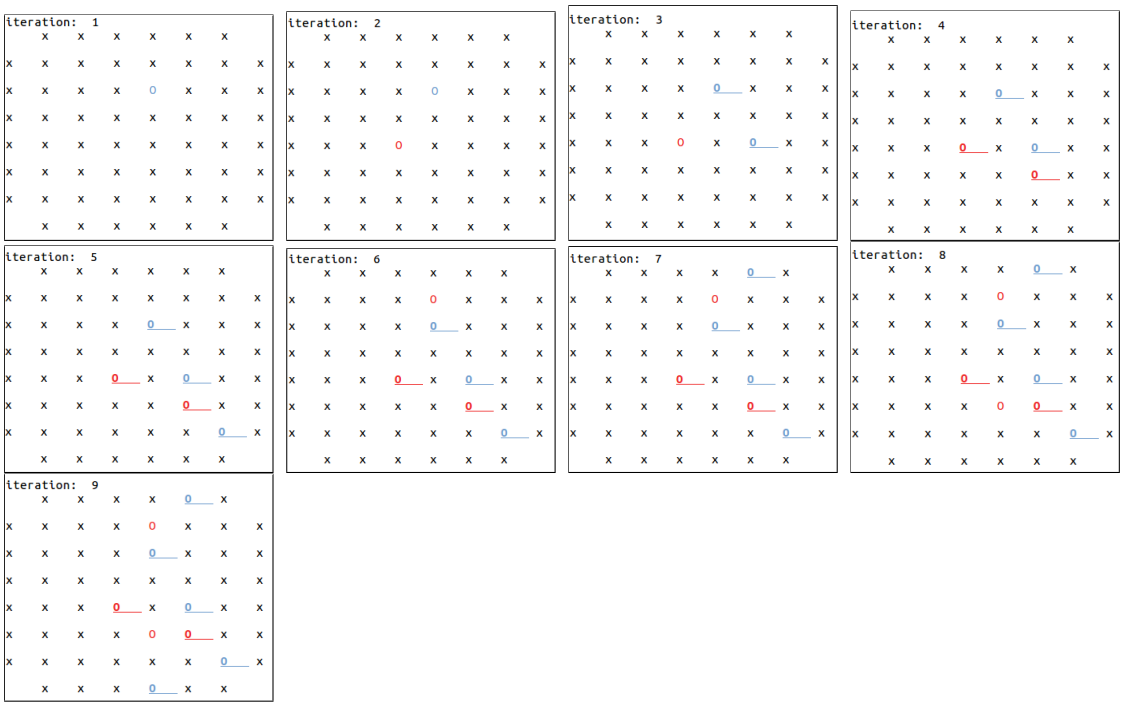

Figure 7: Monte Carlo Tree Search vs. Pure Monte Carlo Game

MCTS Agent vs. 'Resident Expert'

One of the aims of our project was to beat one of our team members (or our expert, as we will now refer to him), which given that our agents managed to play near optimally on an 8×8 board we achieved, but again, it depended on who got to move first. Below is an example of one of the matches where our MCTS agent managed to beat our expert.

When playing the MCTS agent against our expert on a 12×12 board (too large/long to fit in this report), the agent gave our expert a "tough match", which is more praise than any of the rest of our team got when playing him.

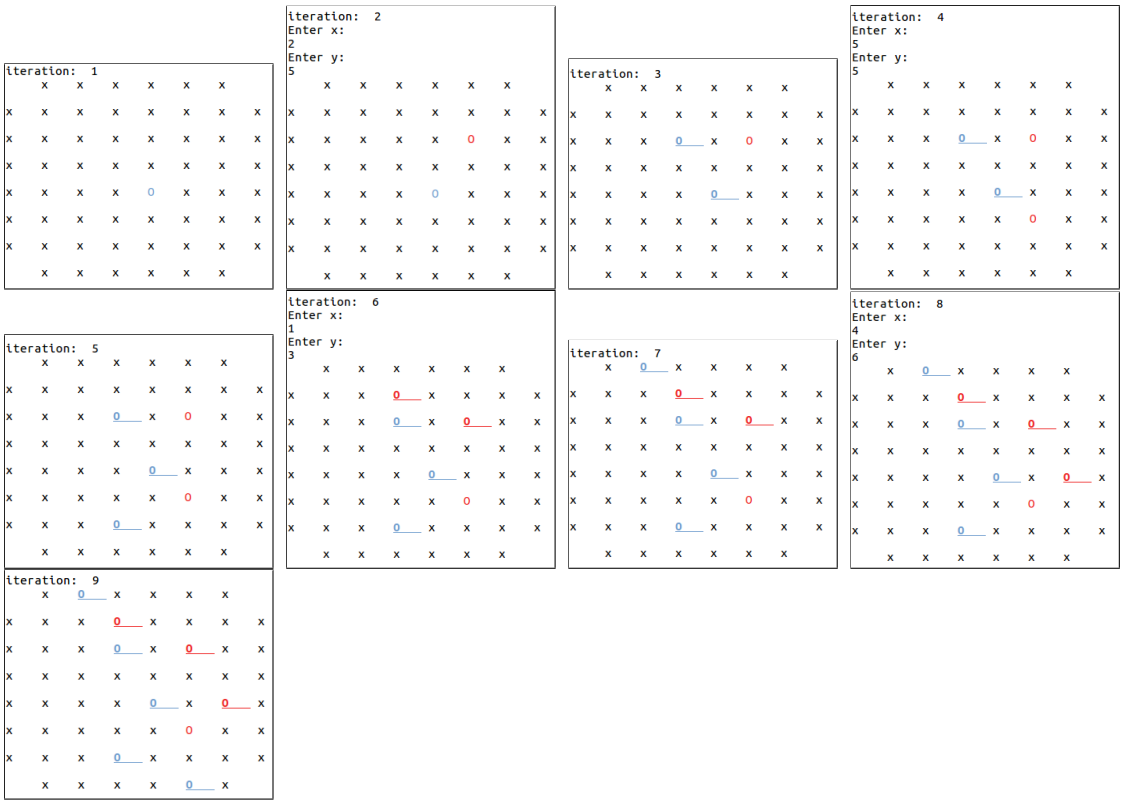

Figure 8: Monte Carlo Tree Search vs. Human Game

Areas for Potential Future Work

If given time we would have liked to look further into policies to be used with MCTS. Literature commonly uses the UCB1 policy (which yields the UCT algorithm when combined with MCTS) [7, 6], and has provable convergence properties to optimal play.

Another idea that could be pursued is to learn some policy using machine learning. However, this presents a rather difficult problem, because as we could find, no data set of twixt moves exists. Therefore the main challenge in proceeding here would be that supervised learning isn't achievable without collection of a data set, and reinforcement learning would likely take too long to converge unless 'warm started' with a function learned from some data set. Indeed, this approach of warm starting a Q-function for reinforcement learning was used in AlphaGo [4]. One potential idea for solving this problem is to try and formulate some representation of a Q-function that is parameterized by the size of the board. If we managed to formulate such a function, we might hope to learn a good one, using episodes from our other agents on smaller boards. We would then hope that this function still performs decently on larger sized boards, or may at least provide some decent start for reinforcement learning in the absence of training data.

References

[1] Édouard Bonnet, Florian Jamain, and Abdallah Saffidine. Havannah and twixt are pspace-complete. In *International Conference on Computers and Games*, pages 175–186. Springer, 2013.

[2] Jos Uiterwijk and Kevin Moesker. Mathematical modelling in twixt. *Logic and the Simulation of Interaction and Reasoning*, 29, 2009.

[3] James R Huber. Computer game playing: the game of twixt. 1983.

[4] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[5] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

[6] Cameron B Browne, 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 Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[7] Edward J Powley, Daniel Whitehouse, and Peter I Cowling. Bandits all the way down: Ucb1 as a simulation policy in monte carlo tree search. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[8] Markus Enzenberger, Martin Muller, Broderick Arneson, and Richard Segal. Fuego—an open-source framework for board games and go engine based on monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.