CS 221 Final Report: An AI Agent for Blokus Duo

Trey Connelly, Kevin Hu, Kenneth Wang

Stanford University

Project code: https://github.com/treybc/CS221

1 Introduction

Our goal is to create an AI agent for the game Blokus Duo. Blokus is a strategy board game in which players alternate playing polyominoes on a grid such that each of their pieces are connected only by the corners. Currently, there has been little to no effort to solve the game or produce expert AI players, so research into this would be novel.

A sample game. Note that pieces only touch their own color by corners.

2 Related Work

The most prominent agent to date in the field of game playing is AlphaZero, which was able to beat grandmasters in chess, go, and shogi (which is a Japanese variant of chess). In constructing our own AI agent, we referenced a recent 2017 paper by David Silver and his colleagues, titled "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm." The paper describes the high level structure of AlphaZero, which uses reinforcement learning and self-play to "achieve a superhuman level play" within a day, "starting from random play, and given no domain knowledge except the game rules." The AlphaZero engine uses a deep neural network that, given a game state, outputs both a vector of action probabilities given the game state and an estimated utility from the game state. It then learns the parameters of the neural network by using Monte Carlo Tree Search, where the simulations are generated by self-play. In

essence, self-play is a mechanism through which the parameters of the neural network is tuned by letting the player play against itself. If the newer version is able to achieve a 55% win-rate against the previous one, then the player updates to the newest version. AlphaZero started with a random player, but was able to learn to exceed the performance of grandmasters in 24 hours.

Our work did not use self-play and reinforcement learning to generate a high-level player from scratch. That is, every complete iteration through the game does not improve the agent, as it is based on a fixed searching algorithm through the game tree. However, we implemented a similar MCTS algorithm that repeatedly runs simulations until the time-limit per turn is exhausted. It is particularly powerful in the late game where it gets close to exhaustively searching all possible tree branches and makes the best move at the given game state. MCTS is rather weak in the early game because it is not able to explore the majority of the paths in the game tree; and this weakness can be exploited to beat our game agent. To improve the performance of our Blokus player, we could reference the self-play concept from AlphaZero and implement a reinforcement learning algorithm on a similar neural network model.

3 Task Definition

The original game of Blokus is made for four players playing against each other. Since this includes a significant social component (e.g. players teaming up with their friends or against the winning player), we chose to analyze Blokus Duo, a two-player variant on a smaller board. Our goal is to create an AI agent capable of at least matching human-level performance at the game. Since this is a deterministic game, the agent should produce a policy which for each board state (input) produces the optimal action at that state, i.e. places a tile on the board.

This agent involves searching a game tree to find the move with the highest value. Each player in Blokus has 21 pieces, one of each polyomino of length 1 to 5. Since the game ends when both players run out of pieces or are unable to move, the maximum depth of any game is 21, or 42 piece placements (although most games end before all pieces are played, generally after 30-35 piece placements).

The complexity of the problem mostly comes from the branching factor. Each piece can be rotated and reflected before being placed, giving up to eight distinct variants of the same tile, and there are a large number of positions to place each tile. The very first turn, before any pieces has been played, has 414 possible moves. If, for example, the "X" tile is played first, there are 800 possible second moves. The branching stays high for the first 5 turns and then begins to decrease exponentially as the number of pieces in each player's hand decreases and the board fills up, reducing board space available to play.

4 Approach

a. **Baseline**

To review, our baseline is a simple "greedy" agent. Since at the end of the game players are penalized for the number of squares in their unplayed tiles, playing the tile with the highest number of squares is the best short-term move. Thus, the greedy baseline simply chooses a random highest-value playable tile in its hand and plays it in a random valid position. While this maximizes short-term value, it does not seek an advantageous board state or account for adversarial play.

b. **Evaluation Function**

In order to implement depth-limited minimax search, we need a good evaluation function that gives us an approximate utility of a board state. The function takes in a "gameState" and outputs a score for the player agent while taking the opponent into account.

Ideally, the evaluation function should capture the general Blokus strategies that would be used by a human player, allowing the agents to make smart decisions. We will discuss some of the heuristics that we have currently experimented with and some potential directions and ways to improve in the future.

Our current evaluation function is a linear predictor consisting of a collection of heuristics, whose weights are assigned and adjusted manually. The first heuristic that we implemented is the current "score." At the end of the game, the score of each player is determined by subtracting the total number of squares in their hand (i.e. an unplayed piece made of 5 squares is worth -5 points, and an unplayed single square piece is worth -1 point). In addition, if a player has no remaining pieces in their hand, they get a bonus of $+15$ points. Instead of calculating the players' scores at the terminating states, we wrote a helper function that enables us to access the theoretical utility of a game state simply by calculating the Blokus score as if the game is finished. Essentially, this encourages our player to play larger pieces on the board whenever possible because the fewer squares remaining in your hand, the better.

Another idea that we attempted to capture is the notion of space. In Blokus, the players are in a sense claiming territories while blocking off one another. We want our agent to be able to expand its territory and move towards open space. Initially, we tested the number of available corners and the number of possible actions as measure of space and freedom. But they can be misleading because you can have a large number of corners, but they are all clumped up and does not actually give you many options. So a heuristic that we came up with is the notion of span. At a given board state, we count the total number of tiles that can be covered by a potential valid move. This gives us a better indication of how much of the board that the player agent can potentially influence in the next move.

In our analysis of the evaluation heuristics, we determined that reducing the opponent's playable space is generally more important than increasing one's own playable space. Intuitively, once the opponent is unable to access an area, or at the end of the game, unable to play anything, then the player can freely maximize their utilization of the remaining space without having to worry about an adversary. This was captured by changing the features from being the difference between the two players to being a feature for each player's span and corner score, and then weighting the opponent's features more heavily (and negative).

A general strategy of Blokus is to move towards the center of the board, especially in the opening stages of the game. To reflect this in our player, we implemented our evaluation function such that it counts the number of the player's tiles relative to the number of the opponent's tiles that lie within the central 6×6 grid. The more you have, the higher the score.

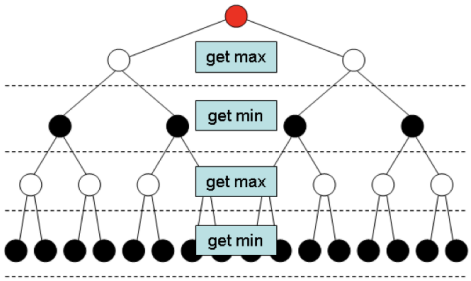
Finally, we know that intuitively, a move is good if it sets you up for future expansion into open space. This is difficult to achieve on a machine because the move itself may not provide the player with the highest score. One way that we experimented with this idea is by writing a heuristics function that counts the amount of empty space that are nearby available corners (where pieces can be placed). While this works sometimes, it is nowhere near perfect, as you can have many empty spaces that are not connected in a way where the player can place blocks. For example, a checkerboard configuration of empty squares would greatly confuse our player agent.

With an abundance of linear factors, it is difficult for us to come up with the optimal weight assignments for the evaluation function to be effective. To begin with, we tinkered with the weights manually using our intuition and observations of the test trials. We could potentially develop a TD-Learning algorithm to learn the weights for the evaluation function, but we decided to move our effort away from tuning the evaluation function and instead toward developing Monte Carlo Tree Search, in which an evaluation function is less important.

c. **Minimax Search**

We implemented a minimax search algorithm with $\alpha - \beta$ pruning in order to slightly reduce the impact of the large branching factor. The search is depth-limited, using the evaluation function from the previous section. Due to the extremely high branching factor in the first few turns, the search runs fairly slow with search depth 1 (i.e. considering a full turn of player and opponent), and cannot run efficiently with a search depth greater than 1 (see results section). This limits the effectiveness of the search, as it can only consider one turn ahead.

$$V_{\text{max,opp}}(s,d) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{max,opp}}(\text{Succ}(s,a),d) & \text{Player}(s) = \text{Agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{max,opp}}(\text{Succ}(s,a),d-1) & \text{Player}(s) = \text{Opp} \end{cases}$$

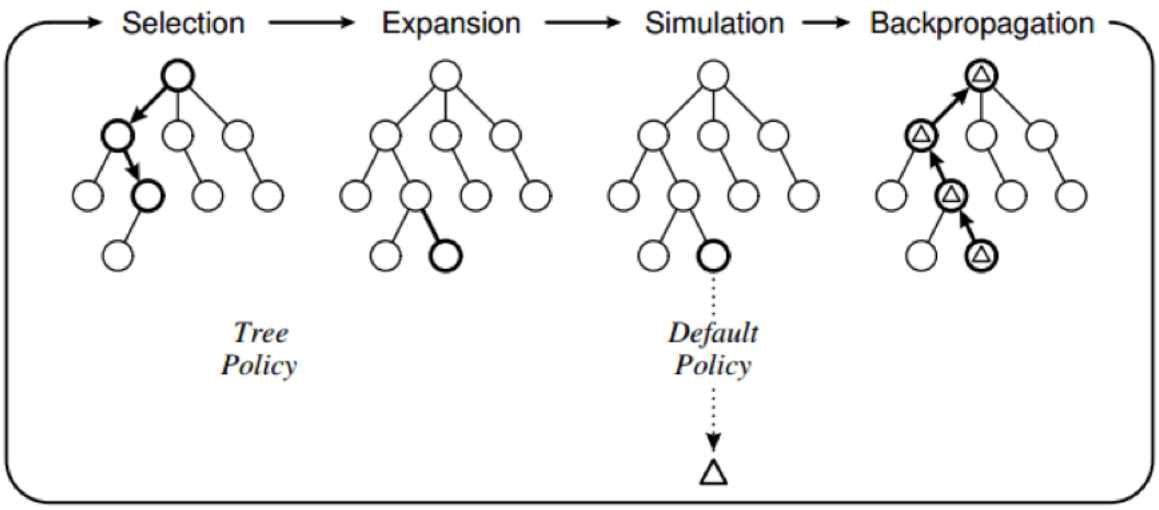


Because we are limited to exploring a single turn, a simple minimax search, even with depth-limiting and alpha-beta pruning, cannot be efficiently used in this problem. As a result, as mentioned in the previous section, rather than attempting to continue developing our minimax algorithm and tune our evaluation function, we turned our focus to Monte Carlo Tree Search.

d. **Monte-Carlo Tree Search**

Due to the extremely high branching factor for the game Blokus, we necessitate the use of approximate methods like Monte-Carlo Tree Search. This algorithm has been very successful in game-playing, producing the strongest AI agents for complicated games like Chess and Go.

Monte-Carlo Tree Search involves an analysis of the most promising moves, mainly by sampling the search space at random and expanding the search tree. There are four steps to the algorithm: selecting a promising leaf node, create a child node from that leaf node, simulating a random playout of the game until the end, and backpropagating the result to all nodes visited. Given enough iterations, the algorithm will converge to the optimal play of minimax.



Monte-Carlo Tree Search does not require an evaluation function to learn how to play the game well, as it repeatedly simulates full games. MCTS can use a "pure" tree search which simulates random playouts for the tree exploration portion of the algorithm, but it can also use a heuristic to drive the exploration playouts. We settled on using a simplified version of the evaluation metrics used in the previous section, specifically the utility score and opponent's corner score, as a heuristic to weight the choices when MCTS explores the tree. This makes MCTS more likely to explore moves which we a priori think will be good, but does not exclude it from searching other moves which may turn out to be better.

In order to balance out exploration and exploitation in our MCTS algorithm, we made use of the

following formula called the Upper Confidence Bounds (UCB):

$$\overline{X_j} + \sqrt{\frac{2 \ln n}{n_j}}$$

At a given state in the game tree, our MCTS agent will choose to play branch j that maximizes the value of the expression above (assuming all branches have been explored in the past). In the expression, $\overline{X_j}$ represents the current average reward from playing branch j; n is the number of plays so far, and n_j is the number of times that the particular branch j has been played so far. The UCB formula makes intuitive sense because $\overline{X_j}$ encourages the agent to exploit actions that leads to a higher reward whereas $\sqrt{\frac{2 \ln n}{n_j}}$ allows the player to explore states that have been visited less frequently.

A further area of development would be to implement a self-play approach combined with Monte-Carlo Tree Search, similar to AlphaZero, which would most likely result in the strongest AI agent for Blokus.

5 Results

a. **Metric and Experiments**

Since there is little existing work on optimizing Blokus and no agents to compare to, our primary metric is evaluating our agents by playing against humans players. Additionally, we can analyze each agent's performance against the baseline agent, as well as against the other agents we developed.

In our experiments, we calculated the average score over 20 trials of each player playing against each other agent. By running multiple experiments of one hundred trials between the baseline agent and itself, we calculated that on average, the first player has a three point advantage over the second player, a common theme in two-player zero-sum games. We are uncertain without further testing whether this handicap value, generated from near-random play, applies exactly to higher-skill play or if the value changes. However, since we know that there is some advantage to going first, we conducted half of our trials with one agent going first, and half with the other agent going first.

For the parameters used in our tests, we set our minimax agent to search depth 1 because of the high branching factor, as previously discussed. MCTS was set to run for 30 seconds per turn, and the baseline agent has no parameters. Trey Connelly was the human tester for all human trials in order to keep the human skill level relatively consistent; he has played Blokus in the past and has an average skill level for a human player. Depth 2 search essentially never finished (we stopped the test after 30 minutes; it had not completed the first turn). Data is averaged over 20 trials with players alternating who goes first and second.

b. **Results**

The results of our trials are summarized in the following table, where positive score indicates an overall victory for the row's player. Note that we did not conduct trials where an agent played itself, as we know these trials would approach an average score of zero over a large enough number of games.

	Human	Baseline	Minimax	MCTS
Human		58	49	-9.3
Baseline	-58		-19.4	-9.0
Minimax	-49	19.4		19.7
MCTS	9.3	9.0	-19.7	

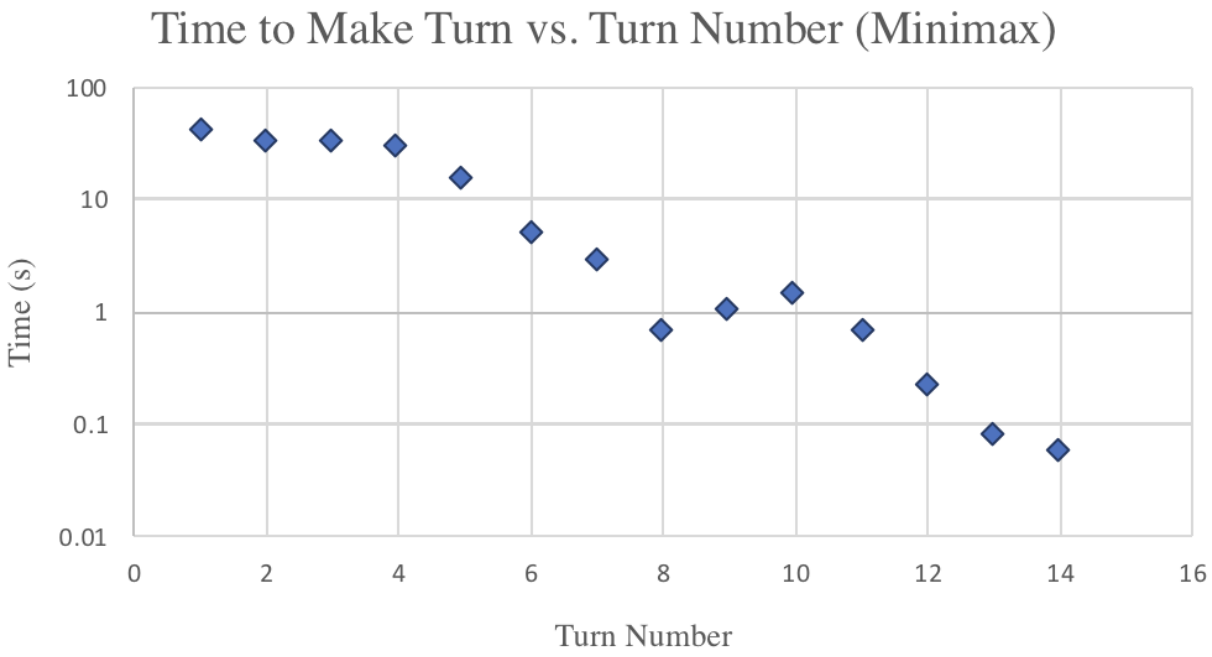
In terms of performance, every player is able to beat the baseline player as expected. Interestingly, minimax performed better than MCTS against non-human players, yet MCTS was the only agent able to beat the human player. The human tester won every game easily against baseline and minimax, while the games against MCTS were generally far closer, with MCTS winning 7 of every 10 games played, with an average score of 9.3 points.

c. **Further Analysis**

- As a qualitative analysis of play styles, baseline and minimax agents are very predictable, and thus easy for a human player to counter. The evaluation function we developed appears to be mostly useful in the early game, where blocking opponents pieces is more important, and these agents tend to play much more poorly in the late game. For example, they perform poorly at determining how best to fill an area that has been "captured" (i.e. that the opponent cannot reach) in order to maximize piece placement.

  On the other hand, MCTS will sometimes start the game playing inefficiently, as it has run relatively few simulations, but as the game progresses, it has seen more states and can run further simulations much faster, since the branching factor and the number of turns left in the game both decrease. As a result, its play approaches optimal as the game progresses. MCTS's play is hard to predict as a human, and the agent is adept at taking advantage of mistakes and risky moves on the part of the human player.

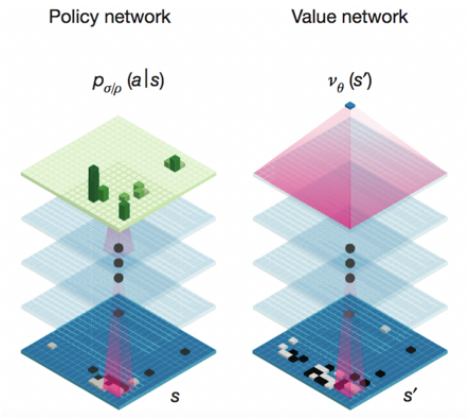
  We theorize that the minimax agent was able to beat MCTS because it performed much better in the first few turns while MCTS was still functioning fairly poorly. Minimax's aggressive strategy of highly prioritizing blocking the opponent was able to seal out the game at the start, while the human player's more balanced strategy of attempting to secure regions of the board gave MCTS more time to improve.
- We also measured the time efficiency of the minimax search algorithm. A human player can generally make a good move with 30 seconds, so MCTS was set to a constant 30 second turns. The following graph displays average turn time of depth-one minimax search (with alpha-beta pruning) for each turn over all trials. Note that, on the logarithmic scale of the graph, runtime decreases roughly exponentially, reflecting the similar decrease in the branching factor of the search tree.

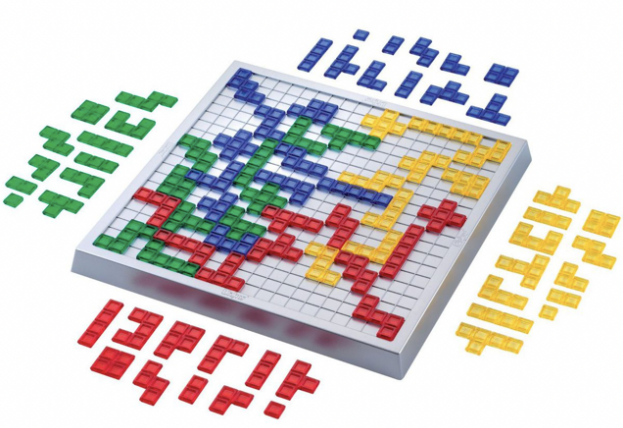


Minimax search with a search depth of 2 performed far worse. In our initial trial of depth-2 search, on turn one the search took longer than an hour, and on turn 10 the search took over 10 minutes. Since these values were so far out of the acceptable bounds, we decided there was no need to further analyze minimax search for any depth higher than 1.

6 Conclusion

We managed to create a player that outperforms humans consistently, namely the MCTS agent. In the process, we encountered challenges due to having high branching factor and time constraints. The MCTS agent is clearly not the optimal agent, since it did not perform well against minimax, but there is a large amount of room for development.





In the future, one possible avenue of exploration is to combine the current model with neural networks and self-play, following the model of the successful AlphaZero. Once Blokus Duo is better solved, we could also potentially extend the project to the traditional, four-player Blokus, analyzing how the social component affects the performance of our agent and trying to tune it to account for this.