CSCI 4511W Final Project: Blokus Artificial Intelligence

Will Davies
davie304@umn.edu 5440669

Isaac Marquardt
marqu402@umn.edu 5257274

May 11, 2020

Abstract

Games are an excellent test domain for developing complex artificially intelligent behavior. Two-player games have been an especially fertile testing ground, employing strategies such as Mini-max, alpha-beta pruning, and other common heuristic search techniques. Well-defined evaluation functions and domain specific knowledge both contribute to powerful artificial intelligence capable of besting professional players in the likes of checkers or chess. Recently, developments have been underway to expand this behavior to games with more difficult constraints: namely greater than two players, hard to define evaluation functions, and large depths and/or branching factors. Developments such as AlphaGo Zero have used innovative techniques to solve some of these issues, namely using the Monte-Carlo-Tree-Search algorithm coupled with a neural network to guide its expansion [8]. This article will explore the capabilities of several different AI techniques using the game Blokus as a test framework. Blokus has several interesting challenges it brings to the table, including four game players, a high branching factor (over 4 million possible configurations for the first four plys), and a tricky-to-write evaluation function. These various AI techniques will be pitted against each other to determine which is the most formidable and has the best chance of outplaying a human opponent.

1 Introduction

Humans have played games since the ancient era. Games challenge participants' minds but have also proved to be complex problems that not even a computer capable of billions of calculations per second can solve completely or trivially without certain compromises. The challenge lies in constructing a clever AI that uses its runtime wisely and makes informed decisions to the best of its ability. Blokus fits this criteria perfectly. No computer on Earth could brute force game-winning solutions to Blokus. Even for a game that appears simple with only a 20 x 20 grid and a hand full of pieces, Blokus's game-tree is far larger than a computer can solve in our lifetimes. Instead, artificial intelligence is concerned with playing games like humans do, not trying to brute force solutions but instead using inherent wisdom and the ability to adapt to make quality decisions and outplay an opponent. The pinnacle of AI will have the ability not just to play well, but to see new things and learn for itself in ways the programmer never intended. In this article, we investigate Blokus, a seemingly simple game with many layers of complexity. We investigate how well an artificial intelligence can play. Will the AI be a formidable opponent to a human? Will it succumb to a human's ability to reason, learn, and adapt, or will it crush the measly mortal by thinking dozens of moves ahead? This paper seeks to answer that question, to implement several concepts introduced in CSCI4511W, and to extend them with various modifications suggested by technical journals to make our AI a worthy opponent. We first discuss the mechanics of Blokus. Then we discuss related work and build off of it to implement a solution in our custom Blokus engine. We formulate a hypothesis that our *Progressive History MCTS* will be the strongest opponent. We design an experiment and run several tests pitting different algorithms against each other and discuss the implications of the results. Finally, we end with future work and an appendix containing some relevant code.

1.1 Blokus Mechanics

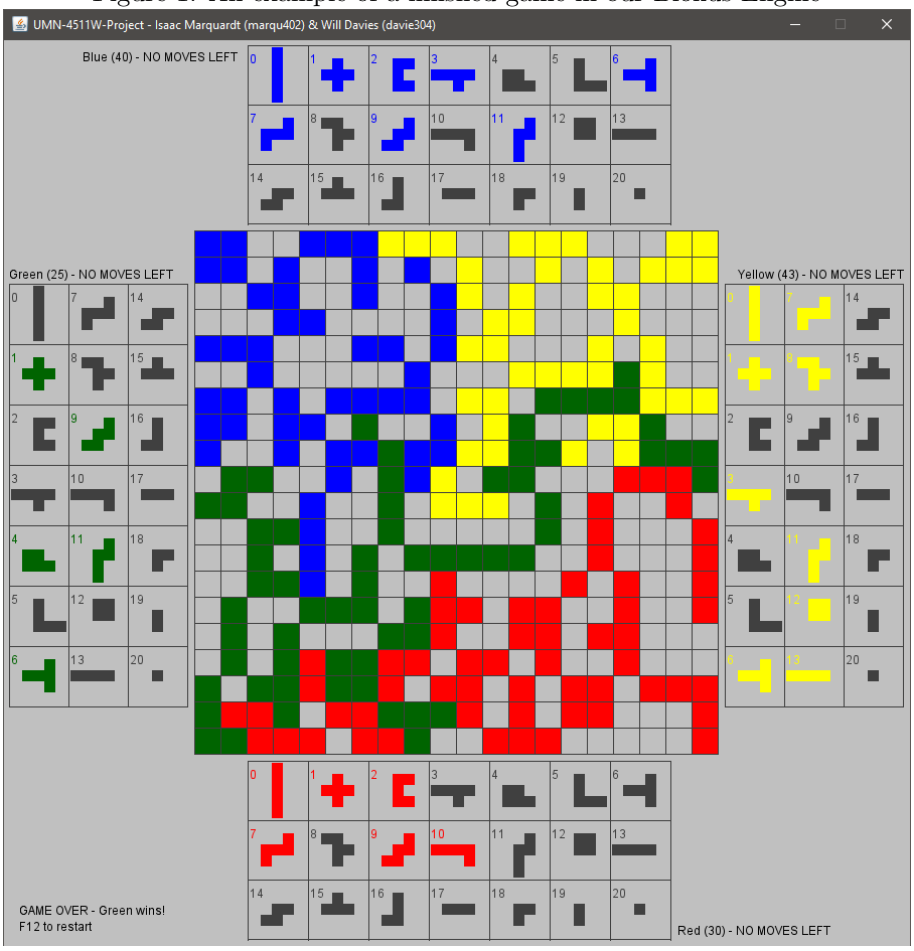
Blokus is a 4 player free-for-all game, played on a 20 x 20 square grid. Each player begins with 21 pieces varying in both size and shape. The size of each piece is between 1 and 5 grid squares large, inclusive. The goal is to place all of one's pieces onto the board, and the end state is when no player is able to place any more of their pieces. Players put their pieces onto the board sequentially, and may flip or rotate the piece in order to place it correctly. There are a few rules for putting down a piece in a valid position. The first rule is that no two pieces may be on the same grid square. The second rule is that two pieces of the same color may not be horizontally or vertically adjacent but they must be diagonally adjacent to at least one already played piece of the same color. The winner is the player with the largest number of occupied squares on the grid when no players are able to place any more pieces. In the case of ties, the person who placed the single piece in the last round - the monomino - wins the game. As a result, a common strategy is to save the single piece for the last move, unless the player needs to place this piece in order to survive. One strategy which takes the importance of the monomino into account is the Barasona strategy, in which the player sets their pieces such that an opponent must use the monomino in order to pass through. A common counter-strategy to the Barasona is to be highly aggressive in the beginning game, such that the player reaches the middle of the board first [4]. One unintentional mechanic in the game is that the first player to move has a higher chance of winning. This is similar to chess, in which the player who uses white wins proportionally more than the participant using black [4].

In a 4 player game of Blokus, a game may last a total of 21 rounds (84 plys, where a ply is one player placing one piece). In finding the number of possible unique states, we will use the relaxation that any piece may be broken up and placed anywhere on the board. The grid contains 400 squares total and each square is in one of 5 states: red, blue, green, yellow pieces, or empty. With this relaxation, once all squares are placed, there are 5^{400} possible states, equivalent to 3.87×10^{279}. In a real game the number of feasible states is likely less than this due to players not being able to lay down all of their pieces. In practice, the number of practicable states is closer to 4.76×10^{156} [2]. For comparison, chess has a theoretical number of legal states in the range of 10^{40} to 10^{50} [18]. In trying to find the branching factor for Blokus, observe that each person on average has 139 possible moves [14]. Therefore, the branching factor for a single move is 139 compared to a branching factor of only 35 in chess [2]. In a game of 4 players, the average branching factor for Blokus is 139^4, equivalent to 2.3×10^8. For chess, the branching factor is 35^2, or 1,225. Note that these branching factors occur near the middle of the game. For Blokus, the branching factor near the end of a game is significantly lower than 139, due to there being few empty areas remaining.

1.2 Blokus Engine

For this experiment, we designed and developed a complete Blokus game engine in Java from scratch, making it easy to tweak the system as needed and easily integrate search algorithms. The engine provides informative visuals indicating the pieces each player holds and a dynamic game board that the player can interact with. The Blokus engine has a built in strategy swapping mechanism to make it easy to set the strategy a particular player should utilize. The engine also includes a built-in benchmarking framework to run hundreds of games dynamically and report win statistics, a feature heavily utilized to produce the experiment results below. The engine also includes a virtual game simulator, allowing search algorithms to easily explore and expand the game tree.

Figure 1: An example of a finished game in our Blokus Engine



The reader is encouraged to download the game for theirself at

https://github.com/wdavies973/UMN-4511W-Project

and attempt to defeat the AI. Download and usage instructions are available on the README at the provided link.

2 Related work

Algorithms such as Depth-First or MCTS are unable to fully explore their search trees before the situation calls for new computation. As a result, the algorithms must be fast enough such that they can find a good solution in a short amount of time [15]. One example of such a program is the Deep Blue algorithm. This machine was a chess engine, and was the first AI to beat the world-chess champion Garry Kasparov. This technology was innovative due to its levels of parallelism, tree search algorithms used, and efficient use of an existing grand-master game database [3]. The use of parallel search is especially interesting, because it indicates that being able to isolate the search processes increases the speed at which the search tree is examined. This in turn would allow for more optimal solutions to be found, as indicated by the success of Deep Blue. The parallel processing as utilized by Deep Blue increases total processing output by minimizing the amount of time the CPU spends waiting for its sub-processes to return with data needed to process other tasks. It does so by providing

all information to each process. As a result, parallel processing computers are capable of producing results by orders of magnitude more quickly than serial processing units [19]. This strategy can prove useful in games such as Blokus or chess. Both have a relatively high branching factor and may be utilized in either a static or dynamic environment. As a result, many concepts that improve the performance of a program in chess will also improve a program's competence in Blokus.

A second circumstance within this environment is that it is multi-agent. In this case, although both chess and Blokus are multi-agent environments, there is an important difference between these two games - chess is a two player game while Blokus is a four player game. While significant research has been performed on activities involving two agents, few studies have been completed on larger multi-player games [7]. In one study, artificial intelligence was used in a game of Diplomacy. This is interesting because similar to Diplomacy, Blokus is ultimately a competitive experience with a few cooperative mechanics built in. Possible approaches one could possibly make to such a game are to be entirely solo, entirely cooperative, or a mix of both. Advantages to being solo are that the player is able to target anyone they choose without fear of betrayal. The disadvantage is the weakness of a single player as compared to a group of players working together [7].

A heuristic is needed for players to measure their performance. One potential heuristic is the sum of the sizes of each shape on the board in which the larger shapes provide more points to the player than the smaller shapes. This is similar to chess, in which each type of piece is worth a differing number of points. The queen, for example, is considered to be worth 9 points, and the pawns are each worth 1 point [1]. This is useful to the AI, because it allows for prioritization of actions in the search for better positioning. The best pieces to remove first in Blokus are the 5 squares, while saving smaller pieces for later when less space is available. Most approaches for using tree search algorithms to solve Blokus make use of this heuristic to some degree. A simple use of this is the greedy search, which simply takes a random piece from the largest available pieces and places it onto the board. A slightly more advanced algorithm that considers this heuristic is depth first limited search. Unlike Greedy-First search, Depth-Limited is able to consider the results of its actions multiple moves beforehand [9]. The advantage of being able to look multiple moves ahead is an increased performance. Due to high branching factors, the disadvantages to such a strategy are increased run-time as well as increased memory usage. Depth search will especially suffer from a high branching factor, due to searching all of the children of a single node before moving onto the next node. As a result, such strategies for games with high branching factors are sub-optimal. In such a game, a dynamic search algorithm is more appropriate.

Dynamic strategies are similar to Greedy-First and Depth-First in that they are also tree search algorithms, however unlike Greedy-First and Depth-First, Dynamic strategies change the weighting of their heuristics depending on what stage of the game they are currently in. An example of this is chess, in which the early game is focused primarily upon getting positioning towards the middle of the board, and later more weight is placed upon capturing the opponent's king [11]. For Blokus, such a strategy would be performed by changing the importance of the size of the shape placed, and how this shape interacts with opponent's pieces. In the beginning of the game, the goal is to use larger pieces, and block other players' actions. Towards the end of the game, the focus shifts towards being more tactical about how to increase one's own longevity, rather than trying to hinder opponents' movements [9]. The advantages to this strategy are an increased performance, assuming the transition between weightings is timed correctly. The disadvantages to such a system would be the difficulty of implementing the code for when to transition. In Blokus, this transition could be derived from the number of elapsed moves, however like chess, Blokus games can go at different tempos. If opponents are aggressive, the transition would need to occur more quickly. If players are more defensive, then the optimal point for transition would occur later in the game.

A second strategy to deal with a large branching factor is the use of alpha-beta pruning. This strategy is most useful on games in which there are deep branches. Chess is a perfect example of such a game, because a single game of chess can theoretically last hundreds of turns [10]. Compared to chess, Blokus is a less optimal game for the use of alpha-beta pruning, due to a game lasting a

maximum of only 84 turns and having a four player game tree instead of a two player game tree. This approach would still notably increase the run-time, however, so pruning is still worth considering. The use of alpha-beta pruning eliminates trees known to be suboptimal. The advantages to this involve improvements to both run-time as well as memory use. The extent to these two advantages depend on the degree to which branches are pruned. If for example, a branch is pruned at the first instance of a lower score than other branches, then the run-time and memory space will improve significantly. The cost of such aggressive branch pruning is a decrease in search optimality (if an opponent were to play a non expected move). By removing all subtrees that do not presently have optimal solutions, the search is removing possibly better solutions at a later point in the game. From this, it is seen that for alpha-beta pruning the relation between result optimality and run-time/ run-time is inverse. The more optimal the desired solution is, the slower the resulting run-time [16].

The primary difficulty in Blokus is that many standard search techniques don't apply on four-player game trees. Techniques such as Minimax and alpha-beta pruning need to be modified before they will work on a game like Blokus and even still, techniques like alpha-beta pruning aren't especially effective on a four-player game tree. Chao tested several primitive techniques on a smaller version of Blokus - Blokus Duo - which is a 14 x 14 board for 2 players only. He used several greedy strategies with varying heuristics, such as the number of played pieces or number of available moves, all with limited success [5]. Njissen's thesis on multi-player games provides excellent insight into expanding standard Minimax and other search techniques to multi-player games (more than 2 players). Alpha-beta pruning is still difficult in a game with more than four players and usually isn't a very effective unless a search strategy like best-reply search is used. Njissen's results indicate that Monte-Carlo Tree Search is clearly stronger than Minimax based techniques. Njissen's thesis dives into several enhancements for the MCTS, including $\max^n$, *Progressive History*, and best-reply search playouts [13]. Njissen's and Winand's joint article also provides an excellent overview of MCTS modifications in a multi-player environment. They also introduce a MCTS-Solver which proves theoretic game value positions to aid in their testing. Their advice is largely similar to the former article, but includes a detailed discussion on several different confidence bound modifications such as RAVE and Gibbs sampling which can combine information from multiple simulations together and utilize historical action information [12]. These techniques are the primary techniques used in this experiment.

Other developments include the enhanced iterative-deepening search from Reinefeld and Marsland. This category of techniques uses well known strategies such as A* or iterative-deepening A* to run these algorithms on large game-trees. While this experiment didn't explore any iterative-deepening techniques, many others have experienced success using these types of algorithms [17]. Still, the speed of MCTS and the advantage of not needing a concrete static evaluation function are preferred and are the primary reasons MCTS is used in this experiment.

Interestingly, there hasn't been exhaustive research for games with more than two players and many of the formerly described techniques are only just developing. While researching, we only found one AI for Blokus called Pentobi which also makes use of MCTS and includes an action table for game openings and endings. The field is quite exciting with many new games on the horizon containing challenging problems to solve with new and enhanced artificially intelligence techniques.

3 Approach

As stated above, several unique characteristics make this problem more difficult to solve. Firstly, there are four players instead of two, so many of the default search implementations such as Minimax, alpha-beta pruning, or MCTS will need to be adjusted to work with more players. Nijssen illustrates the difficulty of Minimax based approaches in a four player game, specifically the difficulty and ineffectiveness of pruning on a four player game tree [13]. Secondly, the game has a very high branching factor. While the max depth of any game tree in Blokus is only 84 at most, the game tree can be incredibly wide making it difficult for a search algorithm to inspect a high quantity of moves. Third, the game evaluation function is difficult to write. The area sum of all played pieces may seem like a

good trivial heuristic or static evaluation, but it does not account for position. Tactical positioning in Blokus is important because while most players can easily maintain a score within a few points of the leading score in the early game, position becomes critical at the mid to late game where it will expand the winning power of the player. These criteria make Monte-Carlo Tree Search the obvious best-candidate for a Blokus artificial intelligence. First, MCTS does not need a static evaluation function, but instead can simply use game playout simulations. MCTS simulations will however return the trivial evaluation value described earlier as a result of a playout simulation. This has the advantage of weighting MCTS towards stronger wins that hold a higher point difference from the losers in the game. For these reasons, MCTS is the primary candidate for discussion and experimentation in this article, and several enhancements and variations of MCTS are studied and tested to determine the most effective strategy. Several other algorithms are included to build a context for the quality of MCTS and how it compares to several other techniques. Eight different search algorithms have been implemented and the advantages and drawbacks are outlined below.

3.1 Random Strategy

The random strategy provides an excellent baseline for other algorithms to build on. If an algorithm can't outperform random strategy, it wouldn't be useful. The random strategy will play a random valid move each turn. While the obvious advantage is the speed at which the algorithm can play, the algorithm is very poor and can be easily beaten.

3.2 Greedy Strategy

The greedy first strategy is a simple strategy to implement, yet provides significant advantages over the random strategy. First, the run-time complexity of the greedy strategy is $O(n)$. Greedy first goes through the list of pieces and chooses the largest piece that can be validly played. If two pieces of the same size are compared, a random piece is selected. Once a piece is chosen, it is placed into a random valid position on the board. In this case of this implementation for Greedy First search, the lower bound run-time is also $\Omega(n)$, in which n is the number of pieces the player has currently not placed. As a result, this tree search will always loop through all of the player's pieces, rather than stop once it has a 5 square piece.

3.3 Depth-First Limited Strategy

The Depth-Limited Search is an improvement over the greedy strategy, in that it looks multiple moves ahead and determines the total number of points it is able to gain over that span of moves. Due to the branching factor, the maximum depth is set at two, in order to allow the search to explore other primary children before the allotted one second time interval for the search ends. This makes it so that Depth-Limited Search is a modification of the greedy search that looks 2 moves ahead instead of just 1. One important note regarding this algorithm is that it takes the maximum possible number of points, rather than the average number of points gained. Therefore, it is possible for an opponent to block the spot the Depth-First AI would have played the next turn and effectively make the results of the search inaccurate. The run-time of this algorithm is approximately

$$O(n^2) \tag{1}$$

This is given by an upper bound:

$$O(n + (n + children_{n1} + children_{n2} + ... + children_{nn})) \tag{2}$$

This may be simplified as:

$$O(n + (n * children)) \approx O(children * n) \tag{3}$$

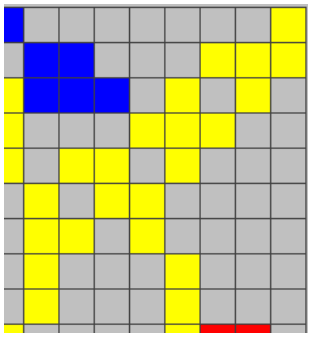
$$O(children * n) \approx O(n^2) \tag{4}$$

This is due to the algorithm searching the nodes of all of its children, as well as the nodes of all of its children's children. In this case, the number of the primary children from the root node given is denoted as "n". The children of the n node are referred to as "children". The children of a specific node may be of a smaller or larger size than the children of other n, but on average they will be referred to as being of size "children". Furthermore, "children" are assumed to have a branching factor similar to "n", so "children" are approximately equal to "n".

3.4 Barasona Strategy

The Barasona strategy is not a standalone strategy, but instead leads another strategy. The Barasona strategy plays moves from a list of pre-programmed Blokus openings, defaulting to another strategy once its opening moves have been played. The Barasona strategy helps address the difficulty of a high branching factor in Blokus. Especially at the beginning of the game MCTS struggles with the large branching factor and can often get a bit lost, leading it to not try to immediately gain invaluable center control or build a strong defense against opponents. This can put the AI at a disadvantage for the late game due to not having far reaching pieces throughout the board. The main advantage to the Barasona opening is how it utilizes the larger pieces in order to create a wall that can only be passed by using a single piece and helps MCTS towards the beginning of the game-tree when the branching factor is incredibly large.

Figure 2: An example of a player using Barasona. The upper right corner corresponds to the starting corner for yellow



3.5 MCTS

Monte-Carlo Tree Search is the obvious candidate for solving a problem like Blokus, but unfortunately the default implementation is designed for a two-player game tree and will need modification. First, MCTS uses a four phase process to analyze game moves:

1. **Selection.** The selection phase will traverse through the game tree and select a node (an action) to explore based off two criteria: how many terminal wins were achieved through this node during simulations and how long its been since it was explored previously. The standard technique is to use the UCB1, or the upper confidence bound. This UCB1 returns the value of a node, and an included constant factor allows the user to tune the exploration / exploitation tradeoff. The UCB1 is defined as follows:

$$v_i = \bar{x}_i + C \times \sqrt{\frac{ln(n_p)}{n_i}} \tag{5}$$

   In this equation, the node i with the highest value v_i will be selected. x_i represents the total score achieved through the node divided by the number of visits to that node, n_i. C is a constant

factor controlling the tradeoff between exploration and exploitation, and n_p defines the number of visits to the parent node.

2. **Expansion.** In the expansion phase more candidate game actions are added to the tree. As mentioned, the Blokus engine includes a game simulator which allows MCTS to easily expand and explore different areas of the game state tree. Discussed further below, it is imperative that the expansion phase is efficient, as MCTS is given a limited compute time and must make wise use of it.

3. **Playout.** The playout phase will randomly playout game actions until a terminal node is reached, returning a win or loss for the root player which will be backpropagated in the next step.

4. **Backpropagation.** The backpropagation phase will take the result of a playout and update scores and visits for each ancestor node along the playout path.

Nijssen's thesis outlines an easy to implement method for extending MCTS to three or more player games. Instead of each playout simulation returning the terminal value of the root player, the simulation will return an array containing the terminal values for each player. Then, this four length array will be backpropagated as normal, but each node will set its score to the respective player score it belongs to [13]. This technique is actually very easy to implement and provides a great starting point for a Blokus AI. Importantly, MCTS can run for a user defined period of time. The longer this time period is, the more nodes MCTS can explore. Nijssen's and Winands describe multiple techniques for improving each of these four MCTS phases to increase the performance [12]. First, Nijssen and Winands compare several selection techniques, from $\max^n$, paranoid, and best-reply search (BRS). Max^n is a modification of Minimax that can be applied to multi-player games of more than two players. According to Nijssen and Winands' testing, $\max^n$ achieved the best results and thus $\max^n$ is used in this experiment. The baseline vanilla version of MCTS uses $\max^n$ for node scores and selection, but uses the default UCB1, expansion, and playout strategies. It should be noted that the speed of functions called during MCTS is important, because faster algorithms will allow MCTS to explore more nodes before its alloted time period is up. Significant work on the Blokus engine has gone into optimizing the playout functions and its support functions such as game action expansion.

3.6 Progressive Bias MCTS (PB-MCTS)

Chaslot describes an improvement to the selection technique called *Progressive Bias* which adds a heuristic history factor to the UCB1 [6]. The formula is defined as follows:

$$v_i = \bar{x_i} + C \times \sqrt{\frac{ln(n_p)}{n_i}} + \frac{H_i}{n_i + 1} \tag{6}$$

Here, the first component is the same UCB1 as before, but the heuristic of node i, H_i is added. The heuristic used here is the total area of a player's placed pieces at node i weighted with an additional factor of how many open/available corners the player can play at. The total placed area is weighted slightly higher, but using available corners also helps ensure MCTS will look for game states that also have a strong position. This advancement does provide higher quality actions in the game but at the cost of fewer playouts per second due to having to compute a complex heuristic. The *Progressive History* technique can be useful if the heuristic can be computed quickly and there is sufficient domain knowledge, but in the case of Blokus, the heuristic isn't terribly efficient to compute [6].

3.7 Progressive History MCTS (PH-MCTS)

In his thesis, Njissen address the issues with *Progressive Bias*, including the potentially slow compute time and difficulty in problems with hard to define or rarely available domain knowledge. Instead,

Njissen proposes a technique called *Progressive History* which is domain independent, fast to compute, and has an attractive $O(1)$ memory complexity. Again, the *Progressive History* technique is a modification of the UCB1 value used to select nodes. *Progressive History* posits that an action that is successful in one game position is likely successful in other similar game positions as well [13]. To accomplish this, first the total score and plays are recorded for each action for each player when they occur in a Monte-Carlo simulation. These are stored in a table that can return the average score over a run of simulations for a certain action. Then, this value is factored into the modified confidence bound:

$$v_i = \bar{x_i} + C \times \sqrt{\frac{ln(n_p)}{n_i}} + x_a \times \frac{W}{(1 - \bar{x_i})} + 1 \tag{7}$$

Here UCB1 is the same as before, but a new factor is added in, the progressive history. $\bar{x_a}$ represents the average value for an action across several simulations, and W is a weighting factor to define the tradeoff between UCB and an action's history. Per Jnissen's results, a value of 5 was selected for W [13].

3.8 Further enhancements

So far, max^n has addressed and improved the Monte-Carlo tree structure strategy as well as PH-MCTS improving the selection phase. Finally, Njissen dives into optimizations for the playout phase of MCTS. Higher quality and more realistic playouts can result in MCTS producing stronger moves. Several playout strategies are discussed, such as greedy and random strategies. Playout runtime is important because a faster playout function will allow MCTS to explore more nodes. According to Njissen's results, best-reply search (BRS) is a strong technique for running playout simulations. In best-reply search, the search tree is reduced to only the AI player and the strongest opponent. All other weaker players are ignored. The playout simulation will then run through the actions by searching for the best-reply an opponent can make and simulating a game playout through it [12]. This technique seems promising, but was not implemented due to time constraints and would be a interesting technique to study and implement in the future.

4 Experiment Design

The aforementioned Blokus engine is used to benchmark various search algorithms against each other. Each of the players (red, blue, yellow, green) is given a strategy which will be queried at turn start. The strategy will receive information about the complete game state and may expand or simulate playouts on it without affecting the actual game state. When it is ready, the strategy will submit the action it decides on, progressing the game. Many different combinations of search strategies were tested, with the Blokus engine running each match-up through 100 game trials. Importantly, Blokus has a rather significant first-turn advantage, so the first-turn is evenly distributed to players over the 100 game trials, i.e. red will start first for 25 games, yellow for the next 25, and so forth. The random strategy is used as a baseline for experimental results. Each game, the winning player and associated strategy is recorded. As mentioned earlier, the game terminates once all players have exhausted all available moves, with the player holding the least remaining cell area declared as the victor. The experiments are grouped into two categories. Category one will compare two different search algorithms together by assigning two colors to each search algorithm (the algorithms will not work together however). Category two will compare four search algorithms together by assigning each algorithm to its own color.

4.1 Constants

Monte-Carlo Tree Search constants are set to the following:

- MCTS compute time is set to 1000 ms, MCTS can run indefinitely, playing higher quality actions with a higher compute time. In this experiment however, MCTS and its variant are restricted to 1000 ms which puts them on equal footing and minimize the amount of time waiting for the benchmarks to complete.
- The MCTS UCB1 exploration/exploitation tradeoff is set to 0.2, which prioritizes exploitation of promising nodes, this is important because as the branching factor is extremely high, too much time wasted exploring actions may cause the algorithm to find a lot of moves, but not exploit promising nodes to identify quality actions.
- The progressive history tradeoff W is set to 5. As mentioned earlier, progressive history is the tradeoff between UCB1 and the expansion of moves shown to be successful in historical simulations.

The random strategy will play a random valid action each turn. The random strategy is a useful baseline, because any algorithm that does not exceed the performance of the random algorithm is categorically unproductive. The random strategy does not inspect any information at all, which will provide a clear baseline for other algorithms whose wise use of information is directly related to the amount of which it outperforms the random strategy. Most algorithms are benchmarked directly against the random strategy to get a feel for how much they outperform it.

4.2 1v1 Results

Table 1: Game win results over 100 games

Strategy 1	**Strategy 2**	**Strategy 1 Wins**	**Strategy 2 Wins**
Random	Greedy	6	94
Random	Depth-First	0	100
Random	MCTS	0	100
Random	PB-MCTS	0	100
Random	PH-MCTS	0	100
Greedy	Depth-First	23	67
MCTS	Greedy	93	7
MCTS	Depth-First	77	23
MCTS	PB-MCTS	42	58
MCTS	PH-MCTS	48	52
MCTS	MCTS w/ Barasona	45	55

4.3 1v1v1v1 Results

Table 2: Game win results over 100 games

Strategy 1, 2, 3, 4 (respectively)	**Strategy 1 Wins**	**Strategy 2 Wins**	**Strategy 2 Wins**	**Strategy 3 Wins**
Random, Greedy, Depth-First, MCTS	0	17	27	56
Depth-First, MCTS, PB-MCTS, PH-MCTS	6	29	29	36

5 Analysis of Results

The results were excellent and came out exactly as expected. Firstly, all strategies except for greedy flawlessly won against the random strategy in all games, as to be expected. Even the greedy strategy only lost to the random strategy in 6 of the 100 games. We expected the following to hold true: $Random < Greedy < DepthFirst < MCTS \approx PBMCTS < PHMCTS$, and the results supported this hypothesis. Greedy outperforms Depth-First 67-23, MCTS outperforms Depth-First 77-23. The 1v1v1v1 results are especially helpful. For *Depth-First versus MCTS versus PB-MCTS versus PH-MCTS*, PH-MCTS holds the lead with 36 wins, MCTS tied with PB-MCTS in second place at 29, and Depth-First at 6. For PH-MCTS we see about a 24% improvement over vanilla MCTS, which is quite pleasing. PB-MCTS as predicted has a major drawback of taking more time to compute the heuristic, so while it is theoretically choosing better moves, it couldn't process as many new moves, cancelling out these effects. The Barasona strategy proved to significantly outplay its non-Barasona counterparts, and we suspect that the Barasona will perform even better against humans by preventing a human from trying to exploit the AI by playing extremely aggressively in the early game while the branching factor is still very large. The MCTS strategies proved to be very fruitful and in human testing, they were worthy adversaries to which we lost frequently. Again, in this experiment, MCTS was run with a compute time of only 1000 ms, which is okay when all MCTS variants have this same runtime, but against a human opponent, the compute time can be increased to strengthen the AI's moves. Several tests were performed to help calibrate the exploration/exploitation tradeoff. Several values were tested, such as 0.2, 2, and 4, but 0.2 proved to be the most successful. This parameter is a bit tricky to play with, and the algorithm still performs well with varying values. According to Njissen, the value of W yielded similar results between values of 0-100, so we left it at 5 which proved to be satisfactory. Overall, the experimental results confirm our hypothesis and in the future we would like to extend PH-MCTS to execute higher quality playouts than the random playout strategy it currently uses.

6 Conclusion

Overall, the project was a massive success. We discussed several results from various technical journals, discussing iterative deepening search as an extension of A*, strategies for extending algorithms to four player games, and several enhancements to MCTS. We succeeded in designing and coding a Blokus game engine from scratch and imbuing it with an advanced AI capable of beating human opponents. We implemented several baseline strategies such as the random and depth-first strategy to compare against our MCTS implementations. We started with a vanilla version of MCTS extended to work with four players and continued to improved its selection strategy testing various selection metrics such as *Progressive Bias* and *Progressive History* of which *Progressive History* proved to be a significant and fruitful improvement to vanilla MCTS. We confirmed our hypothesis about relative algorithmic performance and were delighted to see results that affirmed this. This experiment with AI has proven to be invaluable in introducing us to the world of artificial intelligence and understanding how to architect a solution using modern methodologies and techniques for an unsolved or challenging problem. Understanding how to improve upon an existing algorithm with domain specific knowledge and other techniques has been a massive learning experience for the both of us and will prove useful for building custom solutions in the future.

7 Future Work

In the future, we'd certainly like to explore several more techniques for improving MCTS described in Njissen and Winand's technical papers such as BRS search for Monte-Carlo simulations. Njissen also tested several other algorithms for selection, such as a paranoid and best-reply search method. Paranoid search is a particularly interesting technique, working under the assumption that all other

players have formed a coalition [13]. Several other interesting techniques such as using a neural network to guide MCTS could prove to be quite powerful. Overall, we were very satisfied with the algorithm's performance and it was able to beat us quite frequently, achieving our goal. However, we still were able to exploit the algorithm's flaws occasionally to win over the AI, so it would be fun in the future to increase the AI's capabilities to handily beat humans. Additionally, optimizing the data structures and functions in the game could enable MCTS to run through many more simulations per second, increasing the AI's strength. We also hope to add online functions to the game to play with our friends over the internet and challenge them against our Blokus artificial intelligence.

8 Contributions

8.1 Will's Contributions

- Developed the basic Basic game engine with the following subtasks:
  - Graphics engine to render the game and all the pieces
  - Abstract strategy system to easily implement new algorithmic behavior and assign it to colors.
  - Implemented the following concrete strategies:
    - Human strategy - allows a human to input actions
    - Random strategy - randomly plays any valid action
    - MCTS - Extended Monte-Carlo Tree Search to four players, using $\max^n$ using Njissen's article as a reference [13]
    - PB-MCTS - Implemented *Progressive Bias* MCTS using placed tiles and number of available moves as a heuristic using Njissen's article as a reference [13]
    - PH-MCTS - Implemented *Progressive History* MCTS by cross-referencing good actions in multiple simulations using Njissen's article as a reference [13]
  - Implemented a benchmarking engine to run multiple tests at once making it easier to run experiments
- Designed experiments and implemented a method to distribute which player gets the first turn to ensure accurate testing
- Managed source repository and handled merging pull requests and designed a workflow to integrate code
- Optimized and improved many of the internal algorithms through several of the cited technical articles

8.2 Isaac's Contributions

- Developed the pieces within the Basic game engine, in which the following subcategories were fulfilled:
  - Logical constraints for the placement of a desired piece
  - Mechanics for manipulation of a selected piece by allowing the user to flip or rotate it
  - Creation of a piece class listing all of the available pieces for use
- Implemented the following concrete strategies:
  - Greedy First Search - chooses the largest piece to play in a random position

  - Barasona Opening - Dynamic strategy allowing sub-strategy to take precedence after the fourth turn
  - Depth First Limited Search - Considered best combination of pieces up to two moves ahead
- Assisted running tests and recording results

References

[1] M.C. Beal D.F.; Smith. "Learning Piece Values Using Temporal Differences". In: *ICGA Journal* (Sept. 1997). DOI: 10.3233/ICG-1997-20302. URL: https://content.iospress.com/articles/icga-journal/icg20-3-02.

[2] J. Burmeister and J. Wiles. "The challenge of Go as a domain for AI research: a comparison between Go and chess". In: *Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems. ANZIIS-95*. 1995, pp. 181–186.

[3] Joseph Campbell Murray; Hoane. "Deep Blue". In: 134 (1-2 Jan. 2002), pp. 57–83.

[4] Chin Chao. "Blokus Game Solver". In: (Dec. 2018). URL: https://digitalcommons.calpoly.edu/cpesp/290/.

[5] Chin Chao. "Blokus Game Solver". In: (2018).

[6] Guillaume Chaslot et al. "Progressive Strategies for Monte-Carlo Tree Search". In: *New Mathematics and Natural Computation* 04 (Nov. 2008), pp. 343–357. DOI: 10.1142/S1793005708001094.

[7] Fredrik Håård. *Multi-Agent Diplomacy : Tactical Planning using Cooperative Distributed Problem Solving*. 2004.

[8] Sean D. Holcomb et al. "Overview on DeepMind and Its AlphaGo Zero AI". In: ICBDE '18 (2018), pp. 67–71. DOI: 10.1145/3206157.3206174. URL: https://doi.org/10.1145/3206157.3206174.

[9] A. Jahanshahi, M. K. Taram, and N. Eskandari. "Blokus Duo game on FPGA". In: *The 17th CSI International Symposium on Computer Architecture Digital Systems (CADS 2013)*. 2013, pp. 149–152.

[10] Daniel Just Time; Berg. Vol. 5. Random House Puzzles and Games, 2003.

[11] T. Kunz et al. "Dynamic chess: Strategic planning for robot motion". In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 3796–3803.

[12] J Pim AM Nijssen and Mark HM Winands. "Enhancements for multi-player Monte-Carlo tree search". In: *International Conference on Computers and Games*. Springer. 2010, pp. 238–249.

[13] JAM Nijssen and Mark HM Winands. "Monte-Carlo tree search for the game of Scotland Yard". In: *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*. IEEE. 2011, pp. 158–165.

[14] Joseph Nijssen. *Monte-Carlo Tree Search for Multi-Player Games*. Dec. 2013. URL: https://project.dke.maastrichtuniversity.nl/games/files/phd/Nijssen_thesis.pdf.

[15] Martin Pesendorfer and Philipp Schmidt-Dengler. *Identification and Estimation of Dynamic Games*. Working Paper 9726. National Bureau of Economic Research, May 2003. DOI: 10.3386/w9726. URL: http://www.nber.org/papers/w9726.

[16] Aske Plaat et al. "A New Paradigm for Minimax Search". In: *CoRR* abs/1404.1515 (2014). arXiv: 1404.1515. URL: http://arxiv.org/abs/1404.1515.

[17] A. Reinefeld and T. A. Marsland. "Enhanced iterative-deepening search". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16.7 (1994), pp. 701–710.

[18] Stefan steinerberger. "On the number of positions in chess without promotion". In: *SpringerLink* 44 (Nov. 2014), pp. 761–767. DOI: https://doi.org/10.1007/s00182-014-0453-7. URL: https://link.springer.com/article/10.1007/s00182-014-0453-7#citeas.

[19] Promod Vohra. "Parallel Processing Computers". In: *The Journal of Epsilon Pi Tau* 18.2 (1992), pp. 29–36. ISSN: 08879532. URL: http://www.jstor.org/stable/43603594.

9 Appendix

9.1 Base MCTS Code

```
public class MCTSStrategy implements Strategy {

    // the number of seconds the strategy is allowed to work for
    private static final int COMPUTE_TIME_MS = 1000;

    @Override
    public void turnStarted(BlockingQueue<Action> submit, SimulatedNode root) {
        long start = System.nanoTime();

        root.expand();

        while((System.nanoTime() - start) / 1_000_000 < COMPUTE_TIME_MS) {
            double[] result = MCTS(root);
            root.update(result);
        }

        double maxScore = Double.MIN_VALUE;
        SimulatedNode bestChild = null;

        int inded = 0;

        for(SimulatedNode child : root.getChildren()) {
            if(child.getVisits() == 0) {
                System.out.println(inded + " / " + root.getChildren().size() + " were expanded");
                break;
            }

            if(child.getAverageScore() > maxScore) {
                maxScore = child.getAverageScore();
                bestChild = child;
            }

            inded++;
        }

        if(bestChild == null) {
            throw new IllegalStateException("An error occurred" + root.getChildren().size());
        }

        submit.add(bestChild.getAction());
    }

    private double[] MCTS(SimulatedNode node) {
        SimulatedNode bestChild = null;
        double maxUCB1 = Double.MIN_VALUE;

        if(node.getChildren().size() == 0) {
            return node.getScore();
```

```
        } else {
            for(SimulatedNode child : node.getChildren()) {
                double ucb1 = child.getUCB1();

                if(ucb1 > maxUCB1) {
                    maxUCB1 = ucb1;
                    bestChild = child;
                }
            }
        }

        double[] result;

        if(bestChild.getVisits() == 0) {
            bestChild.expand();
            result = bestChild.playout();
        } else {
            result = MCTS(bestChild);
        }
        bestChild.update(result);
        return result;
    }

}
```

9.2 Game simulation code

```
// represents a state as the result of an action being performed
public class SimulatedNode {

    // exploration / exploitation tradeoff
    public static final double TRADEOFF = .2;
    // determines influence of progressive history
    public static final double W = 5;

    private static final Random random = new Random();

    private SimulatedNode parent;
    private ArrayList<SimulatedNode> children;

    // A map from player id to pieces that the player may no longer
    // use throughout the course of this simulation
    private HashMap<Integer, HashSet<Integer>> excludedPieces;

    // state of the game at this node, which will be in turn
    // altered by any of the nodes in "children"
    private final Color[][] grid;

    // All players in the game
    private final Player[] players;

    // The action that was applied to this game state
```

```
private Action action;

// Helper variables that other algorithms may make use of
private double score, visits;

// Which player made the action leading to this state?
private final int player;

public static SimulatedNode CREATE_ROOT(Color[][] grid, Player[] players, int player) {
    // NORMALIZE THE PLAYER ID, this is a special case for the root, because the root
    // has no direct ancestor, so it will look like whatever player is before the current,
    // but not action applies and the previous move doesn't technically exist
    return new SimulatedNode(grid, players, Math.floorMod(player - 1, players.length));
}

// This is only for the special case node at the root, use CREATE_ROOT to make it
private SimulatedNode(Color[][] grid, Player[] players, int player) {
    this.players = players;

    // CREATE EXCLUDED PIECES
    this.excludedPieces = new HashMap<>();
    for(int i = 0; i < players.length; i++) {
        this.excludedPieces.put(i, new HashSet<>());
    }

    // COPY THE BOARD STATE
    this.grid = new Color[Grid.HEIGHT_CELLS][Grid.WIDTH_CELLS];

    for(int row = 0; row < Grid.HEIGHT_CELLS; row++) {
        for(int col = 0; col < Grid.WIDTH_CELLS; col++) {
            this.grid[row][col] = grid[row][col];
        }
    }

    // SET THE PLAYER THAT MADE THE MOVE
    this.player = player;
}

/*
 * Initializes the simulated node, this will copy the game position
 * and apply the action immediately on the grid
 *
 * @param player = the player who is performing Action
 * @param action = the action to apply to the game state
 * @precondition - action is a valid move that can be made on the grid
 */
public SimulatedNode(Color[][] grid, Player[] players, int player, HashMap<Integer, HashSet<Inteq
    this(grid, players, player);

    this.parent = parent;
```

```
    if(action == null) {
        throw new IllegalArgumentException("Action may not be null");
    }

    // COPY THE HASHSETS
    this.excludedPieces = new HashMap<>();
    for(int playerId : excludedPieces.keySet()) {
        this.excludedPieces.put(playerId, new HashSet<>(excludedPieces.get(playerId)));
    }

    // APPLY THE ACTION
    action.perform(true, this.grid);

    this.action = action;

    // BAN THE PIECE FOR FUTURE USE FROM THIS PLAYER
    HashSet<Integer> excluded = this.excludedPieces.get(player);
    excluded.add(action.piece.getKind());
}

// SimulatedNodes are lazily expanded - they are only expanded on demand,
// this will populate the children with available moves
public ArrayList<SimulatedNode> expand() {
    if(children != null) {
        return children;
    }

    /*
     * Note: if there are no nodes that can be expanded, try expanded three more times
     * for the other players, otherwise, set the children to an empty array
     */
    // for the player array
    // bottom is 0, right is 1, top is 2, left is 3
    int nextPlayer = (player + 1) % players.length;

    ArrayList<Action> childActions = new ArrayList<>();

    for(int i = 0; i < players.length; i++) {
        nextPlayer = (nextPlayer + i) % players.length;

        childActions = players[nextPlayer].getAllActionsExcluding(grid, excludedPieces.get(nextPl

        if(childActions.size() > 0) {
            break;
        }
    }

    children = new ArrayList<>();

    if(childActions.size() > 0) {
        // nextPlayer was the player that succeeded in finding at least 1 move to play
```

```
        for(Action action : childActions) {
            children.add(new SimulatedNode(grid, players, nextPlayer, excludedPieces, this, actio
        }

    }
    return children;
}

/*
 * Randomly play out this game state by successively applying
 * actions from "children", make sure grid is reset afterwords though.
 *
 * @precondition = this node itself has already been expanded
 * @constraint = cannot affect the state of the SimulatedNode, should be callable multiple times
 */
public double[] playout() {
    if(children == null) {
        //throw new IllegalCallerException("Please call expand() before calling playout()");
    }

    SimulatedNode currentNode = this;

    while(true) {
        ArrayList<SimulatedNode> childNodes = currentNode.children;

        if(childNodes.size() == 0) {
            // PLAYOUT GAME OVER
            return countScores(currentNode.grid);
        } else {
            currentNode = childNodes.get(random.nextInt(childNodes.size()));
            currentNode.expand();
        }
    }
}

private double[] countScores(Color[][] grid) {
    double[] scores = new double[players.length];

    for(int row = 0; row < Grid.HEIGHT_CELLS; row++) {
        for(int col = 0; col < Grid.HEIGHT_CELLS; col++) {
            if(players[0].getColor().equals(grid[row][col])) {
                scores[0]++;
            } else if(players[1].getColor().equals(grid[row][col])) {
                scores[1]++;
            } else if(players[2].getColor().equals(grid[row][col])) {
                scores[2]++;
            } else if(players[3].getColor().equals(grid[row][col])) {
                scores[3]++;
            }
        }
    }
```

```
        return scores;
    }

    public double getUCB1() {
        if(visits == 0) {
            return Double.MAX_VALUE;
        }

        return (score / visits) + TRADEOFF * Math.sqrt(Math.log(parent.visits) / visits);
    }

    public double getProgressiveBias() {
        if(visits == 0) {
            return Double.MAX_VALUE;
        }

        return getUCB1() + (getPlayerScore() + 0.2 * children.size()) / (visits + 1);
    }

    public double getProgressiveHistory(double xa) {
        if(visits == 0) {
            return Double.MAX_VALUE;
        }

        double relativeHistoryScore = W / ((1 - (score / visits)) * visits + 1);

        return getUCB1() + xa * relativeHistoryScore;
    }


}
```