An AI Agent for Playing Hex

GEORGE HORRELL, Stanford University, USA
ALI MALIK, Stanford University, USA
ELI ECHT-WILSON, Stanford University , USA

In the past few years, Artificial Intelligence has shown its mettle in all manner of fields. In particular, revolutionary work has been done in the area of abstract strategy games, with AI agents for Chess, Go, and Shogi reaching unprecedented superhuman levels. Beyond providing amusement, game playing provides a valuable opportunity for testing the abilities of high performant artificial intelligence. In this paper, we tackle the two-player abstract strategy game of Hex with the hope of creating a medium level player using enhancements to traditional game playing algorithms such as Minimax and Monte Carlo Tree Search (MCTS). Our findings show, in support of the recent work by DeepMind, that an increasingly sophisticated MCTS is a promising game playing algorithm and with enough complexity, has the potential to achieve superhuman strength in a wide variety of games.

1 INTRODUCTION

1.1 Task scope

Our task for this project was to create an AI Agent capable of playing the game of Hex at the level of an intermediate human Hex player. Hex is a two player zero-sum abstract strategy game in which two players alternate placing hexagonal pieces on a diamond shaped board. Player 1 tries to form a connection between the left and right sides of the board, while player two tries to form a connection between the top and bottom. Hex is an interesting game to develop AI agents for because it has quite interesting mathematical underpinnings that allow complex strategy to emerge out of a game with simple rules. Additionally, Hex is an interesting game to approach because it has a high branch factor (high number of actions for each player to take from a given state), and therefore the state space of the game is difficult for an agent to efficiently explore. Hex has a higher branch factor than Chess, but a lower branch factor than Go, placing it in an interesting category where many different algorithmic approaches towards AI may prove to be successful. Our goal for the project was not only to develop an agent capable of handling this high branch factor and playing the game at a high level, but to explore different algorithmic strategies and see which ones are most successful.

1.2 State and game model

The game model follows the standard paradigm we have used thus far in this class for two player zero sum game models. The game state is represented as a tuple (player, board) which stores the player next to move, as well as a HexGrid object for the current state of the board. The HexGrid efficiently compresses the state of the board into 2 bit vectors: one keeps track of the moves made by player 1 and the other keeps track of the moves made by player 2. Initially HexGrid was backed by a 2-D array with a union-find data structure to enable efficient checking of win condition; however this proved to be too inefficient as copying the large data structures at each turn was too expensive.

At each turn, the set of actions available to either player is the set of empty locations on the board. This is easy to calculate and model, but also presents a significant challenge for us, accounting for the very high branch factor on larger boards, making exploration difficult (discussed in more detail later).

The start state for our game is an empty board, with player 1 taking the first turn. For any state and action taken by a player, the resulting successor state flips control to the opposite player and stores an updated board (the exact same board except with one more tile played, corresponding to whatever action was taken on that turn). The end state of the game is achieved when one player has formed a path between their two sides of the board with their tiles (player 1 is moving left to right, player 2 is moving top to bottom). Note that when the board is completely full, it is mathematically proven by Brouwer Fixed Point Theorem that one of the players has won: therefore, it is not possible to tie a game in Hex (Gale, "Brouwer fixed-point theorem"). The utility for an end state is infinity for the winning player and negative infinity for the losing player.

1.3 Evaluation

We evaluate our model by playing it against other Hex agents as well as humans, and using the agent's win rate to evaluate performance.

First, in order to single out one agent from the many we develop as most advanced, we will play the agents against each other in a tournament. This tournament will play each pairing of agent against one another for some fixed number of games, and the winner(s) of that tournament (in number of total games won) will be used for a further, more detailed evaluation of hyperparameters and performance under different conditions.

Testing our agent against human opponents is difficult, because we do not have a variety of human opponents to test against and each of the members of our team has become quite familiar with the strategy of Hex as well as the weaknesses of each of our AI agents. However, the agent performed well against volunteers at the poster session, proving that it can hold its own against human players.

Lastly, we will evaluate our agents in terms of efficiency, or the amount of time it takes for each agent to make a move. One of the largest challenges of this project has been creating agents that can not only make intelligent moves, but do them in a reasonable amount of time. We will evaluate and discuss the efficiency of each of our algorithms, and how it scales with the size of the board.

2 APPROACH

2.1 Oracle

An oracle for this project would be an expert human Hex player. Despite the development of many successful Hex agents in the AI space, expert humans are still capable of beating agents. The upper bound for this task, therefore, is to create a Hex agent that plays at the level of an expert human Hex player.

2.2 Baseline models

2.2.1 Initial Baseline. Our initial baseline model relied on a very naive heuristic to play the game of Hex. The model ignored the aspect of "defense," and essentially tried to form a single path connecting the two sides of the board as quickly as possible. This naive algorithm reliably beats a random policy, but lacks the sophistication to recognize larger strategic patterns.

2.2.2 Minimax with UCS evaluation function. Our first sophisticated approach involved using a standard minimax algorithm with alpha-beta pruning and a custom evaluation function. Due to the large branch factor of Hex (~ 100 on large boards) and the exponential runtime of minimax ($O(b^{2d})$), minimax can only run at a limited-depth of 1 in reasonable time with the computing power available to us. This means that the evaluation function becomes very important in evaluating different states, because we can only explore a limited fraction of the game tree in a reasonable amount of time, even with optimizations such as AB-pruning.

Our evaluation function approximates the value of a given state by calculating the minimum number of moves needed by each player to win the game. A move in Hex is a good move if it makes progress towards winning the game while simultaneously defending, or preventing the opponent from making progress. This evaluation function captures that idea by minimizing the number of moves needed for the agent to win while maximizing the number of moves needed for the opponent to win:

$$Eval(s) = numPlaysToWin(s, opp) - numPlaysToWin(s, agent)$$

This model starts to display more strategic gameplay patterns and even competes well against Monte Carlo Tree Search Agents (at low iteration levels).

2.3 Improved models

2.3.1 Minimax with modified-beam search. In an attempt to be able to search the game tree at a depth higher than 1, we experimented with a modified-beam search version of minimax, which selectively explores the game tree in more promising areas. The beam search algorithm only explores at most K candidate paths through the game tree, using the evaluation function to determine which candidate paths to continue exploring. At maximizer nodes, it considers all possible actions from the current set of candidates and only continues exploring the best K of them. At minimizer nodes, we consider each candidate path and, for each one, extend it with the action that leads to the worst successor state (according to the evaluation function). This agent has a higher efficiency overhead (because it runs the evaluation function many times for each depth layer, not only at leaves) but can run at a higher depth in a more reasonable amount of time. This higher depth is advantageous, but the non-exhaustive search can sometimes miss good moves and pay a price for it.

2.3.2 Minimax with bridge detection. A final improvement to our minimax approach involved incorporating the idea of bridges into the evaluation function. A bridge is a Hex-specific strategy term referring to a configuration of pieces that a player can treat as a virtual connection. That is, any move made by the opposing player to threaten the connection can be countered by the player who owns the bridge to maintain the connection. Below is an example of the simplest and most common type of bridge:

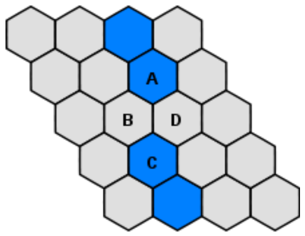

Fig. 1
Bridge pattern are essential to the strategy of Hex. This path is effectively connected because if the opponent takes B, the player can take D and retain the connection.

It is advantageous for a player to form bridges because they can be more productive in claiming space on the board, while not risking the possibility of the opponent cutting off their route towards a win. Our evaluation function takes advantage of this by severely discounting bridges in calculating the number of moves needed to win a game. That is, when a bridge exists, the cost of completing the bridge is a fraction of the cost of an actual move, making it advantageous for the agent to create bridges and stop the opponent from creating bridges.

This extension greatly improved the performance of the minimax agent, allowing it to remain much more competitive with Monte Carlo methods.

2.3.3 Monte Carlo Tree Search. The pitfall of minimax is the fact that it explores the entire state space of the game tree (aside from the branches that are pruned with alpha-beta pruning). With the high branch factor of Hex, this severely cripples the potential of minimax. We decided to explore Monte Carlo Tree Search, a technique that has proven to be effective in similar games with a high branch factor, to overcome this difficulty. Monte Carlo Tree Search effectively prunes the search space by selectively exploring the game tree in places that are more advantageous to the agent.

Monte Carlo Tree Search repeatedly runs for a given number of iterations, and each iteration adds information to the exploration tree. Each iteration consists of 4 phases:

- **Selection**: Select a node(state) in the game tree to learn more about, balancing exploration with exploitation.
- **Expansion**: Expand the selected node to all of its possible children nodes.
- **Evaluation**: Pick a child node and play out a game randomly from that state to the end of the game.
- **Backpropagation**: Update all nodes on the path from the child node to the root with the information from the game playout.

Pure Monte Carlo Tree Search proves to be effective at playing the game of Hex, as long as it can efficiently run enough iterations in a reasonable amount of time. The tradeoff between efficiency and game performance continues to be an issue, even when using MCTS.

ALGORITHM 1: Monte Carlo Tree Search

```
Function MCTS($s_0$)
    Initialise tree root $v_0$ with $s_0$.;
    repeat
        $v_1 \leftarrow Select(v_0)$;
        $\Delta \leftarrow Simulate(v_1.state)$;
        $Backpropogate(\Delta, v_1)$;
    until time remaining;
    return $BestChild(v_0)$
```

2.3.4 Monte Carlo Tree Search AMAF. Based on our research of related work, we decided to implement an AMAF (all moves as first) extension to Monte Carlo Tree Search which has proved successful in other agents. The extension is based on the idea that a good move is a good move, regardless of when in the playout phase it occurs. Essentially, in the backpropagation phase, not only is the count of the nodes in the selection path modified, but also the count of each node's action is modified. So if a playout involved Player 1 winning with a sequence of moves M_1, M_2, M_3, then for each node v in the selection path, the count of Succ(c, M_i) is also incremented, for each M_i. This modification allows for more data about the game tree to be gathered quickly at the possible expense of accuracy. In practice, AMAF has been shown to be more successful than vanilla MCTS.

After our poster session, we further optimized our implementation of MCTS, rewriting in a more performance oriented language (C++). This sped up search by a factor of ~25x, enabling us to run at a much higher number of iterations, thereby improving the quality of our agent.

3 RESULTS

3.1 Analysis

3.1.1 Finding and optimal policy. After implementing a variety of different approaches for our game player, we tested the implementations against each other in a grand tournament. Each one of our

Policies Tournament					
	Minimax	Beam	MCTS	AMAF	AMAF-Br.
Minimax	-	0.50	0.65	0.25	0.45
Beam	0.50	-	0.70	0.25	0.50
MCTS	0.35	0.30	-	0.10	0.35
AMAF	0.75	0.75	0.90	-	0.90
AMAF-Br.	0.55	0.50	0.65	0.10	-

Table 1
Win rate of row policy vs. column policy. Played 20 games on 9×9 board with MCTS run at 5000 iterations.

policies was pitched against every other policy for 20 games (10 with each starting first) and the best players were picked for further exploration. The results can be seen in Table 1.

The results clearly indicated that Minimax and MCTS-AMAF were the two dominant policies of their respective approaches. This matched our expectations considering these were refinements of basic Minimax and MCTS. One surprising result was that MCTS-AMAF bolstered with a more refined playout that deterministically saved bridges within its random policy (AMAF-Br) actually performed worse than the regular MCTS-AMAF policy.

3.1.2 Minimax vs MCTS. Following the tournament, we selected the best two policies, namely Minimax-Bridge and MCTS-AMAF (hereafter referred to as Minimax and MCTS respectively), to see if we could improve them further and then determine which one performed better than the other when their hyperparameters were optimised. With respect to Minimax, most of our improvements were made at an implementation level by improving efficiency of the code. MCTS on the other hand had two important hyperparameters which we explored to see outcome on game playing ability.

An important hyperparameter affecting the MCTS is the inquisitiveness of the UCT based selection policy. When the algorithm is selecting which node to explore, it must balance exploration vs exploitation by picking promising game states to playout further, but also maintaining the ability to explore unfamiliar states in case a better position lies there. The canonical way to do this, as seen in "Combining online and offline knowledge in UCT" (Gelly and Silver), is to use UCT (Upper Confidence bounds for Trees) to select the child node i with the highest value, v_i, according to the formula:

$$v_i = \frac{w_i}{n_i} + c\sqrt{\frac{\log(p)}{n_i}}$$

where w_i is the number of wins that have been played out from node i, n_i is the number of total games played out from the node, p is the number of games played out from the parent of the node, and c is a constant (often set to $\sqrt{2}$).

Figure 2 shows the results of varying the constant c, called inquisitiveness, to determine the optimal value of 0.1. This intuitively makes sense; too large a value and the player is undirected in its search and does not learn more about good states whereas too greedy an approach leads to myopic play.

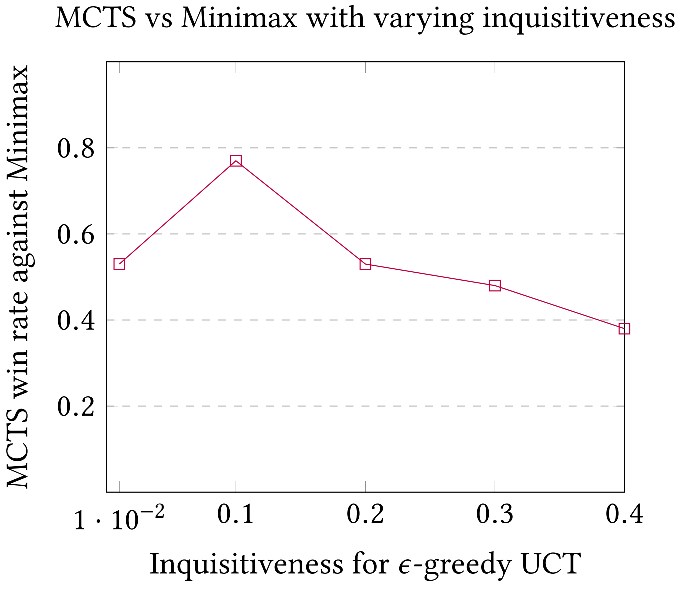

Fig. 2
Effect of varying inquisitiveness of ϵ-greedy exploration in the UCT based selection phase. Played on 11x11 board with 100 games for each data point and 10k iterations for MCTS.

Another hyperparameter we explored was thinking time per move for the MCTS player. As soon as the player's turn starts, MCTS build the game tree from that state, repeatedly exploring and running Monte Carlo playouts to determine what move to make. Intrinsically, one would imagine that the longer the MCTS is allowed to think, the better it performs. However, this has to be balanced with the long time it might take and whether the trade off is worthwhile. Figure 3 shows that around 10-15k iterations of MCTS exploration per move is the optimal thinking time for balancing performance with time.

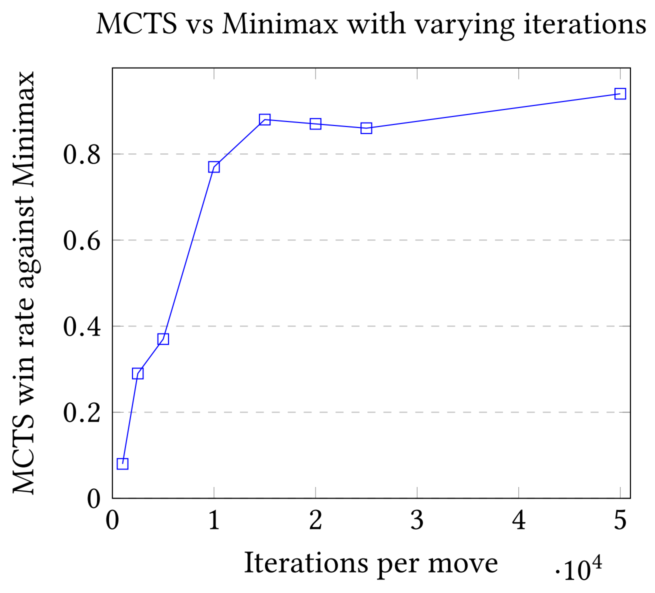

Fig. 3
Effect of varying iterations of Monte-Carlo tree search per move. Played on 11x11 board with 100 games for each data point

With these two hyperparameters determined, we proceeded to test the Minimax policy against the MCTS policy for varying board size. Our hypothesis, corroborated by the data in Figure 4, predicted that MCTS would mostly outperform Minimax on medium sized boards but would start to deteriorate on larger boards. The data showed that there was a severe drop in performance for MCTS on boards bigger than size 11. This makes sense considering the MCTS is still thinking for the same amount of time for each of these boards, whereas the Minimax is using an evaluation function that is almost invariant of the size of the board and is only mildly slower as board size increases. A further area of research would be to increase the efficiency of both these algorithms so that MCTS can run for more iterations and Minimax can run at a greater depth.

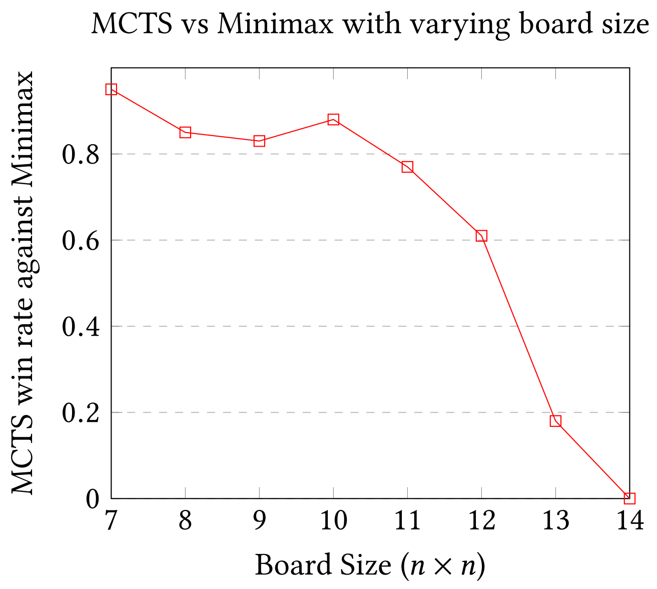

Fig. 4
Effect of board size on policies. MCTS played at 10,000 iterations with 100 games played for each data point.

4 CONCLUSION

Our work thus far has been successful in exploring different approaches towards an artificially intelligent Hex agent, and has shown that Monte Carlo Tree Search with a sufficient number of iterations is the most promising approach. With the correct hyperparameters, MCTS methods outperform minimax and even hold their own against human opponents that are familiar with the game of Hex.

There were, however, limitations to our MCTS game player. Most noticeably, it performed successively worse on larger boards due to the larger search tree and lacked the ability to easily improve further with tweaks to hyperparameters. To overcome this plateau, we identified two main areas of improvement.

Firstly, it was clear that the number of iterations we ran MCTS for played a large role in its game playing ability. Thus, focusing on writing a more efficient, parallelisable implementation of the algorithm would allow large gains in performance. Moreover, we noticed that our testing process was sluggish due to the slow thinking time of MCTS. Improvements to efficiency would allow us to run more tests to determine an optimum configuration for the algorithm.

Secondly, we believe that the MCTS with it's naive UCT based selection and random playouts is inherently limited in the complexities it can capture of a given game. A more sophisticated selection/playout policy would allow for greater information to be gleaned from the Monte Carlo playouts and incorporated into the game player's strategy. In this avenue, the work at DeepMind on AlphaGoZero and AlphaZero have shown phenomenal results using MCTS guided by a convolutional neural network (CNN).

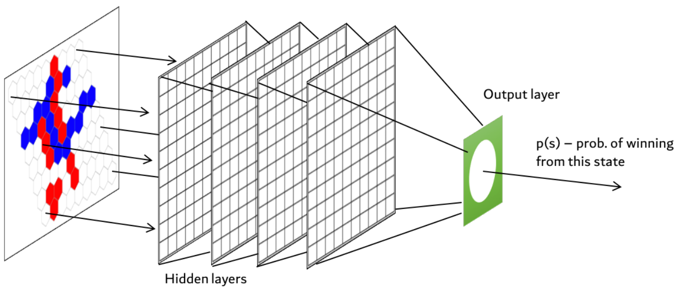

Fig. 5
Neural network used to guide selection/playout phases of MCTS by approximating the probability of winning from a given state.

Our future plans are to incorporate the ideas by DeepMind into our game player by having our selection guided by a neural network. As outlined in Figure 5, the idea is to train a self-playing neural network that takes in a game state and outputs a probability of winning from that state. This predicted probability is used to select which nodes to explore further and the Monte Carlo playout is used as a target value to adjust the neural network's prediction in a reinforcement learning style manner. This improvement will likely make each iteration of MCTS more informative and further improve our agent's performance.

ACKNOWLEDGMENTS

The authors would like to thank Professors Liang and Ermon of Stanford University and our TA, Amani Peddada, for their guidance.

REFERENCES

[1] Silver, David, et al. "Mastering the game of go without human knowledge." Nature 550.7676: 354-359, 2017.
[2] Huang, Shih-Chieh, et al. "MoHex 2.0: a pattern-based MCTS Hex player." International Conference on Computers and Games. Springer, Cham, 2013.
[3] Arneson, Broderick, Ryan B. Hayward, and Philip Henderson. "Monte Carlo tree search in Hex." IEEE Transactions on Computational Intelligence and AI in Games 2.4: 251-258, 2010.
[4] D. P. Helmbold A. Parker-Wood "All-moves-as-first heuristics in Monte-Carlo Go" Proc. Int. Conf. Artif. Intell. pp. 605-610, 2009.
[5] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In ICML '07: Proceedings of the 24th Internatinoal Conference on Machine Learning, pages 273-280. ACM, 2007.
[6] Silver, David et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm." eprint arXiv:1712.01815.
[7] Gale, David. "The game of Hex and the Brouwer fixed-point theorem." The American Mathematical Monthly 86.10 (1979): 818-827.