CMPUT 670 – Project Part 4

Taylor Folkersen

2021 April 15

Intro

The main goal of my project was to make progress on a simple Hex player for smaller Hex boards (like 5x5 or 6x6) that an undergraduate student studying game theory could tinker with and learn from. I wanted the player to be well-documented and easy to understand and modify, and for it to make reasonable moves with around five seconds of computation, as I wanted it to be played against by human players. I also wanted to use some existing Hex theory to improve the search process, and for the player to show the user how it arrived at a choice. My code can be found in the repository at `https://github.com/tfolkersen/Hex`.

Monte Carlo Tree Search

There are different methods I could have used for the search process, such as Monte Carlo Tree Search (MCTS) or alpha-beta search, but I used MCTS because it is used by many strong Hex players such as MoHex, MoHex 2.0, YOpt, MIMHex, and Panoramex, as listed by Huang et al. [5]. MCTS also seems most natural for this problem as it is an anytime algorithm, and it is simple to understand, and many improvements to the algorithm can be made depending on the application. I implement MCTS based on the description by Bradberry [2]. MCTS maintains a tree of nodes representing states of the game's state space, and runs simulations on this tree while the algorithm still has time remaining. A simulation uses a policy within the tree called the *tree policy*, which, at each node, chooses the best action for the player at that node according to some metric, usually that player's win percentage from that node. When the simulation reaches a leaf node, it expands the node by adding its children to the tree, and then continues the simulation using the *rollout policy*, typically a policy choosing random moves for both players.

My MCTS Implementation

Equivalent to the win rate metric, the MCTS formulation in my implementation uses a value function mapping nodes to values in the range [-1.0, 1.0], with -1.0 corresponding to a loss for the player, and 1.0 corresponding to a win for the player. The UCT (upper confidence bound for trees) heuristic commonly used by MCTS, and shown below, is the metric that the tree policy uses:

$$h(N') = val(N') + C \cdot \sqrt{\frac{log(vis(N))}{vis(N')}}$$

Where h is the UCT heuristic, $val(x)$ is the value of node x, N is a node and N' is a child node of N, C is the exploration constant, and $vis(x)$ is the number of times node x has been visited. The value of a node is updated using a standard reinforcement learning update for bandit problems:

$$val(N) = val(N) + \frac{1}{vis(N)} \cdot (R - val(N))$$

Where R is the reward, or the outcome of the simulation (-1.0 if the simulation was a loss for the player, and 1.0 if it was a win). Equation from Sutton and Barto [6] (page 32).

This value function metric is equivalent to the win rate metric, but makes it more natural to apply some reinforcement learning theory to the problem. In particular, my player supports discounting the outcome of simulations. A reinforcement learning agent tries to maximize the sum of rewards received from its environment at each time step, but if interaction with the environment does not end, this sum may be infinite, so in this case an agent may try to maximize the sum of its rewards according to a discounting factor γ, as in:

$$R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$$

Where γ is in the range [0, 1). Sutton and Barto [6] (equation 3.8). The only reward received by the player during simulation is the outcome at terminal states (-1.0 or 1.0), so in discounting this outcome, my hope was that the player would favor moves leading to shorter wins, leaving less room for it to make errors in its play. In testing, this does not seem helpful, so the default discounting rate in my player is left at 1.0 (no discounting).

My implementation also uses a transposition table to store previously-seen tree nodes, so that different move sequences leading to the same game state will result in computation using the same node. Additionally, states whose minimax value are known become marked as solved, and a simulation stops once these nodes are reached, and these nodes may help solve the minimax values for predecessor nodes. In practice this only helps the player in states near the end of the game, as the game's state tree is large and a node is only solved when a child node is known to result in a win, or all child nodes are known to result in a loss, from that node player's perspective.

The player also has the option to add a large bonus to the heuristic function for non-visited children to ensure all moves get tried in simulation.

Hex-specific Player Enhancements

In addition to the aforementioned MCTS features, my player supports simple inferior cell analysis, namely detection and filling in of dead cells and captured cells. As explained by Hayward and Toft [4], dead cells are cells that are not useful to either player, and so can be filled in for either player without changing the outcome of the game, and captured cells are cells that can be filled in for some specific player without changing the outcome of the game. By detecting and filling in these cells, the search space can be greatly reduced by reducing the number of moves to consider, and some states can be solved from fillin. This computation can be costly, so following Arneson, Hayward, and Henderson [1], I only apply inferior cell analysis to a node once it has reached a "knowledge threshold" – a fixed number of visits in simulations. In search, dead cells are filled in for the player's opponent, and captured cells are filled in for the capturing player, and then a "shortcut node" containing the filled in cells is appended to the original node, which is used in subsequent searches. Examples of both kinds of fillin are shown in figure 1. These two patterns are currently the only fillin patterns implemented, as the mechanism for specifying patterns is clumsy; each pattern is recognized by a separate hand crafted function and they are not rotationally invariant. The fillin appears to work, though with this mechanism, each pat-

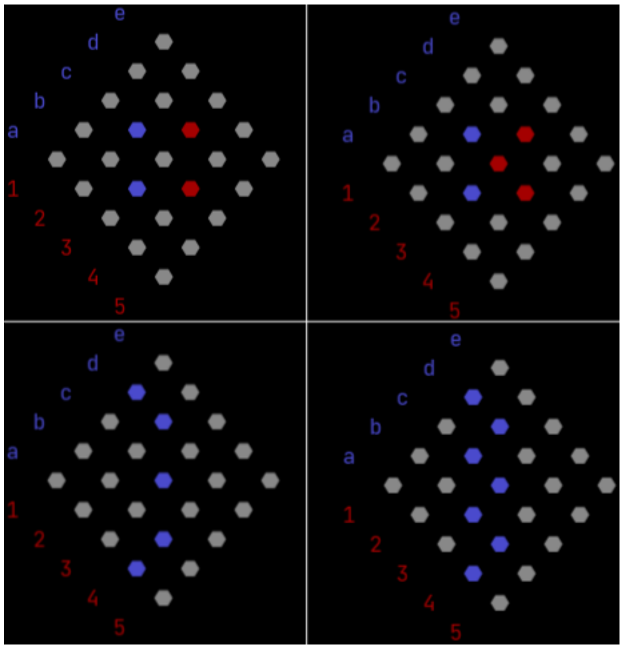

Figure 1: Screenshots of my Hex program. Top left: c3 is a dead cell. Top right: The player fills in c3 for its opponent. Bottom left: c2 and b3 are captured by blue. Bottom right: The player fills in c2 and b3 for itself. Both of these patterns are from Hayward and Toft [4] figure 9.17.

tern has to be separately debugged. Ideally, a pattern should be specifiable by drawing it on a board of arbitrary size, and having the player pre-process all the patterns and their rotations in its initialization.

These filled in cells correspond to legal moves which the player may be forced to make if there are no other moves, so the player is still allowed to make these moves.

Parallelization

Parallelization can be done in a few ways. The authors of Fuego, an MCTS framework used by MoHex, found success using multiple threads working on a single tree using a global lock, or no lock (Enzenberger et al. [3]). My implementation runs on each thread, a separate player with its own memory space, and then combines search results from all threads when choosing the actual move to play. The default *threading* library in Python suffers from global interpreter lock, meaning multiple threads sharing the same memory space will not run in parallel. I instead use the *multiprocessing* library which allows a programmer to run totally separate Python processes and copy data between them. Because an MCTS search tree can become very large, copying trees between these processes in an effort to combine them before resuming search can take several seconds, which dramatically slows down the player and experiments which are necessary to test changes. My compromise is for each thread to keep its tree data to itself, but report values and visit counts of each node corresponding to one of the immediate possible moves back to a master thread, which then combines these values via sums weighted by the visit counts. This is not ideal, as I expect multiple threads working together on one tree to get better value estimates deeper in the tree than multiple threads working on their own trees; still, this method of independent threads seems effective in tests.

Evaluation

Evaluation of this player is difficult because playing it against existing players is unlikely to give any useful results. My player is imperfect on small boards while existing strong players can produce perfect play on these boards, and are much better equipped for handling larger boards, so I would expect my player to always lose to these strong players. The way I have evaluated progress in this project is by comparing performance of the player after each feature was implemented to previous versions of the player.

There are many different versions of the player in the repository, so a summary of features of the relevant ones is given below:

Player	Inferior Cell Analysis	Parallelization	Transposition Tables
MT-FasterPlayer3	✓	✓	✓
MT-FasterPlayer2	✗	✓	✓
FasterPlayer3	✓	✗	✓
FasterPlayer2	✗	✗	✓
BasicPlayer2	✗	✗	✓
RandomPlayer	✗	✗	✗

RandomPlayer makes random moves. BasicPlayer2 is a simple MCTS player. FasterPlayer2 is another MCTS player that is more optimized. FasterPlayer3 implements simple inferior cell analysis. MT-FasterPlayer2 and MT-FasterPlayer3 are multithreaded players running FasterPlayer2 and FasterPlayer3 respectively on their threads, as described in the parallelization section above.

Some experimental data for these players:

Player 1	Player 1 wins	Player 2	Player 2 wins
BasicPlayer2	99	RandomPlayer	1
FasterPlayer2	74	BasicPlayer2	26
FasterPlayer3	46	FasterPlayer2	54
FasterPlayer3	60	BasicPlayer2	40
MT-FasterPlayer3	86	FasterPlayer3	14

Each row is the result of 100 games with each player getting 1 second of search per move, and each game is played on a 6x6 board with a random player starting first. The "swap rule" whereby the first player makes a move and the second player chooses whether to play as black or white, was not used. Multithreaded players were given 4 threads.

BasicPlayer2 beats RandomPlayer almost always. FasterPlayer2 is more optimized than BasicPlayer2 and can do more simulations in the allowed time, so it wins against BasicPlayer2 most of the time. FasterPlayer3 tends to complete fewer simulations than FasterPlayer2, likely because FasterPlayer3 does not have enough inferior cell patterns to benefit from the inferior cell analysis. Running 4 threads of FasterPlayer3 produced better results than 1 thread of FasterPlayer3.

In playing against MT-FasterPlayer2 with 8 threads and 6 seconds of search on a 7x7 board, I found the player to make reasonable moves that a human might make.

Future Work

There are still features I would like to have implemented. Patterns for inferior cell analysis should be easier to specify, and should be invariant to rotation. More patterns need to be added for the inferior cell analysis to really benefit performance, but until patterns are made easier to specify, designing patterns is prone to error. The player should also respond to its opponent playing in its captured cells. Ideally the project should be rewritten in a language like C++ to allow for better performance and more flexible parallelization, like the lock-free parallelization used by MoHex. I also played with other ideas inspired by reinforcement learning theory from Sutton and Barto [6], like bootstrapping values of states from children visited in simulation in an attempt to speed up convergence of value functions, and an update based on the Q-learning update, where the value of a state is moved towards that of the best successor state, rather than the one actually visited in simulation; these ideas seemed to result in a worse player, but I think exploring them more could be interesting.

References

[1] Broderick Arneson, Ryan B Hayward, and Philip Henderson. "Monte Carlo tree search in Hex". In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.4 (2010), pp. 251–258.

[2] Jeff Bradberry. *Introduction to Monte Carlo Tree Search.* Sept. 7, 2015. URL: `https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/` (visited on 03/25/2021).

[3] Markus Enzenberger et al. "Fuego—an open-source framework for board games and Go engine based on Monte Carlo tree search". In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.4 (2010), pp. 259–270.

[4] Ryan B Hayward and Bjarne Toft. *Hex: The Full Story.* CRC Press, 2019.

[5] Shih-Chieh Huang et al. "MoHex 2.0: a pattern-based MCTS Hex player". In: *International Conference on Computers and Games.* Springer. 2013, pp. 60–71.

[6] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.