Creating a Machine Learning Algorithm to Play the Board Game *Cathedral*

by Sean Cox

Abstract: Cathedral is a two-player pure abstract strategy board game designed by Robert Moore and was created and published in 1979 [1]. The game is similar to the popular Chinese game Go, but with some crucial gameplay and stylistic changes. As in Go, the strategy behind Cathedral is complex, and here we investigate the ability of a machine learning algorithm to play the game.

The Cathedral Board [1]

1. Introduction

Cathedral is played on a 10x10 board of squares, with each player's goal being to capture as much of the board as possible by placing pieces. The two players, red and black, both start with an identical set of uniquely shaped pieces built from 1x1 squares. Players are able to place one piece per turn, with turns alternating between players, starting with red. This continues until a player either runs out of pieces and is declared the winner (rare), or more likely, both players run out of legal moves on the board. In the ladder case, the player whose remaining pieces would have taken up the least area is declared the winner, or there is a tie if these values are equal. Games are usually relatively quick and as such most matches involve playing multiple games, usually with players alternating between using the red and black pieces.

If a player surrounds an open area by forming a contiguous wall surrounding an area (the edge of the board counts for contiguity), they claim this area, and the opponent is unable to place any pieces in the claimed area. If the opponent had a single piece in a newly surrounded area, that piece is picked up and returned to the opponent. If the opponent has more than one piece in the surrounding area, this area cannot be captured. This forms a key part of the strategy of

Cathedral, as capturing territory is key to being able to place as many pieces as possible. It should be noted that a player cannot capture territory in their first move.

There is one special piece in *Cathedral*: the cathedral. This large white piece is placed anywhere on the board at the start of the game by the red player and does not count toward contiguity for either player. The cathedral can be captured and returned to the red player, although this is exceedingly rare as the cathedral is the largest piece and as such it is easy for the red player to stop this from happening by placing a piece next to it.

2. Strategy

2.1. General Strategy

Lending to its relatively niche popularity, there has been little written about optimal strategies in *Cathedral*. There are a few general strategic observations that are clear to anyone who has played the game for a brief amount of time. The most obvious strategy is for players to place their larger pieces first. As the pieces in *Cathedral* are made up of unique shapes, it becomes increasingly hard to fit pieces on the board as it fills up. Consequently, the smaller a piece is, the easier it is to find a spot for it, with the 1x1 tavern piece being the easiest to fit and thus probably best saved for the end of the game. Placing the bigger pieces first also helps to capture territory, as the bigger pieces can more easily contribute to creating large contiguous walls. The official *Cathedral* website lists this as the number one strategy [1].

2.2 Solved Game

One interesting aspect of *Cathedral* strategy is it is claimed to be a solved game, meaning there is a definite strategy for the red player that will always result in a win if carried out correctly. This oversight in the game design was originally discovered by game designer and board game aficionado Tom Lehmann. He proposes a fix for this by making a slight tweak to the rules: instead of the red player placing the cathedral to start the game, the black player instead plays the cathedral along with their first move (after red has already placed their first piece). According to him, this removes the ability for red to force a victory, and overall leads to a more balanced game [2]. We investigate both versions of the game rules.

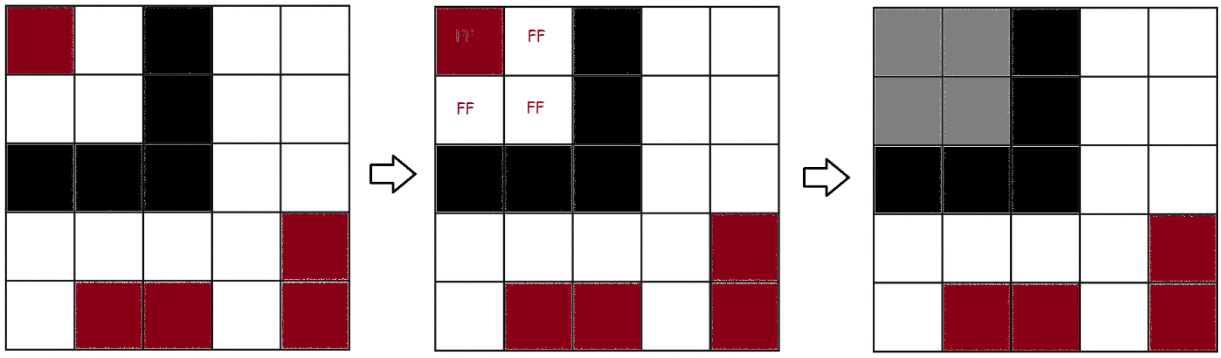
3. Implementation

3.1.The Board

Recreating *Cathedral* in Python is relatively simple. The board is simply a 2d array of numbers and letters, indicating which piece or player controls each square, with positive numbers corresponding to the red player and negative numbers for the black player. For example, an ‘r’ represents territory controlled by the red player, while a ‘-3’ represents the piece with code 3, the bridge, placed by the black player. To implement capturing territory, a modified version of the flood-fill algorithm was used [3]. When a player places a piece, all squares immediately surrounding that piece that are not already controlled by that player (either uncontrolled or controlled by the opponent) are checked with the flood-fill algorithm. In this case, the edges of the board and pieces controlled by the player who just placed a piece are considered the borders of the algorithm. If the explored area contains more than one opposing piece, then the area is not

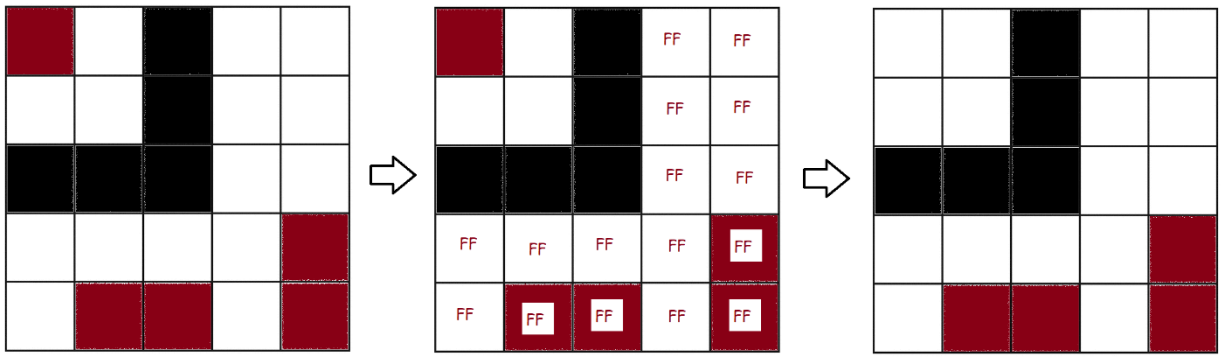
capturable. If not, the list of visited squares is captured by the player. The flood fill algorithm is implemented using a queue.

An example section of the board is shown to illustrate this algorithm. Assuming black just placed the piece shown:



Board Visualization Example 1

When the squares in the upper right corner are checked by the flood fill algorithm, only one red piece is found. Thus, this area is captured by black, and the red piece is returned to the red player.



Board Visualization Example 2

When the flood fill algorithm fills the area to of the rest of the board, it finds two red pieces, meaning this area is not capturable and remains unclaimed.

3.2. Monte Carlo Tree Search

Devising a machine learning algorithm to play *Cathedral* is a similar problem to other games such as Go or chess. In the case of AlphaZero, the best Go playing algorithm, as well as many other chess bots, this is implemented using a Monte-Carlo Tree Search (MCTS) [4]. A MCTS is heuristic search algorithm over a game tree of all possible moves. The algorithm works by exploring the best potential moves by simulating games from those moves, and is split generally into 4 steps: selection, expansion, simulation, and backpropagation.

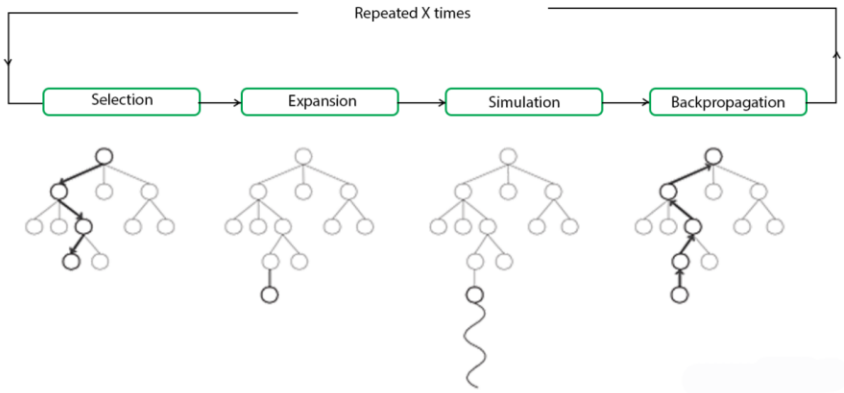
Selection, starting from a parent node, chooses a child node to explore by successively checking child nodes until a leaf node is reached. In this case, a leaf node is one from which the game tree has yet to be simulated from. Child selection is done by with the UCT (Upper Confidence Bound Applied to Trees) algorithm [5]:

$$\frac{w}{n} + C\sqrt{\frac{\ln N}{n}}$$

- w is the number of wins at the current node
- n is the total number of simulations for the node (equivalent to the number of times the node has been visited)
- N is the total number of simulations of the parent node
- C is the exploration parameter, this value determines the exploitation vs exploration tradeoff

This formula forms the basis of how a MCTS works, allowing the algorithm to select for more promising nodes that result in better outcomes.

The next step, expansion, creates a new node from the leaf node reached during the previous step. This node corresponds to a potential next move from the current position. How this move is determined is different depending on implementation. In the case of AlphaZero, a neural network is used. In the case of this implementation, we simply select a random move. Once a new node has been created, the rest of the game is simulated with random moves starting from that node. The result of this simulation (In the case of *Cathedral*: red wins, black wins, or tie) then allows for the final step, backpropagation. Each node records the result of its own simulated end state, as well as the simulations of all the child nodes. This process is carried out by first updating the win counter of the node that has just been simulated, then recursively climbing up the tree to each succeeding parent node, updating their counts as well. This information is then used by the UCT formula to calculate the most promising child node. The entire algorithm can be visualized here:



Monte Carlo Tree Search Visualization [6]

One interesting difference between *Cathedral* and a game like chess is in the endgame. In chess, the potential moves from either position tend to increase after the starting position, with the highest potential amount of moves from a given position usually in the middle to late game. The opposite is true in *Cathedral*, where the number of potential moves from a position consistently decreases as the game goes on and the board fills up. This trend continues until the very end of the game, where the number of available moves is zero. The convergent nature of *Cathedral* leads to a *Cathedral* AI being incredibly powerful towards the end of the game, as it is able to explore all potential ending scenarios and pick the moves that most likely result in a win. We show this in our results.

4. Results

4.1. Playing Against Random Moves

Six game trees were created. Each tree either uses the normal ruleset or the aforementioned modified ruleset, and each tree uses a low, medium, or high value for C, the exploration parameter. The theoretically ideal value for C has been shown to be $\sqrt{2}$ [7], so this was value chosen for the medium trees. In practice, the exact ideal value for C can differ depending on the game, so we investigated both lower and higher values as well. The low and high trees were chosen arbitrarily to have C values of 0.7 and 2.1, respectively. The low-C tree represents a tree which values exploitation over exploration, with a bias towards choosing nodes that have already been shown to be more promising. The high-C tree represents the opposite, biasing exploration over exploitation, and thus is more likely to explore unsimulated nodes. All trees were originally pre-computed from the starting move allowed to explore 2500 nodes and were allowed to explore 25 nodes per game turn. These relatively modest numbers were chosen arbitrarily as to not be overly computationally taxing while still allowing the tree to explore sufficient nodes such that its moves were not essentially random. As we will show, even this low level of depth is more than enough to give the trees a significant advantage.

Trees were initially tested against ‘random’ players, an opposing player who simply makes a random legal move on their turn. We begin with a control simulation, 100 simulated games between a random red and black player. This was done twice, first with the normal ruleset, then with the modified ruleset.

Random Red Moves vs Random Black Moves

	Red Wins	Black Wins	Ties
Original Rules	61	31	7
Modified Rules	36	51	13

These results indicate that under the original rules, red has a significant advantage. This flips under the modified rules, where black has the advantage.

Now we simulate 100 games with each of the trees playing against a random player. For each, 50 games were simulated as red and 50 games as black. First, we look at the results for the game trees under the original rules:

Trees vs Random Moves, Original Rules

	Tree Wins	Random Moves Wins	Ties
Low-C Tree, Normal Rules (C = 0.7)	78	14	8
Medium-C Tree, Normal Rules (C = $\sqrt{2}$)	72	19	9
High-C Tree, Normal Rules (C = 2.1)	72	21	7

And for the modified rules:

Trees vs Random Moves, Modified Rules

	Tree Wins	Random Moves Wins	Ties
Low-C Tree, Modified Rules (C = 0.7)	84	12	4
Medium-C Tree, Modified Rules (C = $\sqrt{2}$)	76	17	7
High-C Tree, Modified Rules (C = 2.1)	73	18	9

Under both rulesets, regardless of whether they were playing as red or black, the trees dominated. It seems the trees with a lower value of C performed slightly better, although this could just be statistical noise. An increased number of simulations could give a clearer idea.

4.2. Trees vs Trees

Having established the strength of the game tree search algorithm in playing *Cathedral*, we simulated games with the trees playing against each other to evaluate which value of C is most promising. It should be noted that these results could change with higher depth trees.

We simulated 50 games between each tree, with each tree playing 25 games as red and 25 as black. First, the trees with the normal rules:

Trees vs Trees, Normal Rules (Win-Loss-Tie)

	Low-C Tree, Normal Rules (C = 0.7)	Medium-C Tree, Normal Rules (C = $\sqrt{2}$)	High-C Tree, Normal Rules (C = 2.1)
Low-C Tree, Normal Rules (C = 0.7)	-	29-17-4	22-25-3
Medium-C Tree, Normal Rules (C = $\sqrt{2}$)	17-29-4	-	28-15-7
High-C Tree, Normal Rules (C = 2.1)	25-22-3	15-28-7	-

There does not appear to be any clear best tree under these rules. The low-C tree had a slight advantage over the medium-C tree, and the medium-C tree slightly outplayed the high-C tree, but these disparities might disappear if more simulations were run, or if the simulations were run with higher depth. To give another metric to weigh the relative skill difference of the trees, we also calculated an Elo rating for each tree, with each tree starting at 1000 Elo:

Tree Elo Ratings, Normal Rules (Initial Elo = 1000, k = 20)

	Elo
Low-C Tree, Normal Rules (C = 0.7)	1009.32
Medium-C Tree, Normal Rules (C = $\sqrt{2}$)	1007.09
High-C Tree, Normal Rules (C = 2.1)	983.58

These ratings indicate that all trees have relatively equivalent skill levels.

Now the trees with modified rules:

Trees vs Trees, Modified Rules (Win-Loss-Tie)

	Low-C Tree, Modified Rules (C = 0.7)	Medium-C Tree, Modified Rules (C = $\sqrt{2}$)	High-C Tree, Modified Rules (C = 2.1)
Low-C Tree, Modified Rules (C = 0.7)	-	27-22-1	26-21-3
Medium-C Tree, Modified Rules (C = $\sqrt{2}$)	22-27-1	-	20-24-6
High-C Tree, Modified Rules (C = 2.1)	21-26-3	24-20-6	-

And their Elo ratings:

Tree Elo Ratings, Modified Rules (Initial Elo = 1000, k = 20)

	Elo
Low-C Tree, Modified Rules (C = 0.7)	1028.32
Medium-C Tree, Modified Rules (C = $\sqrt{2}$)	982.09
High-C Tree, Modified Rules (C = 2.1)	989.58

Again, the results indicate that all trees have relatively equivalent skill levels.

4.3 Understanding the Results

The games played between trees do not indicate any significant advantage for varying values of C. More, higher depth simulations must be run to empirically validate which value of C is best.

The results of the random move control group indicate that, while the modified rules do make the game fairer, they tip the scales a little too far, giving black the advantage. It seems that even though the cathedral does not count towards either player's score, the ability to place the cathedral concurrently with another piece can give the player a significant positioning advantage. Another variant of the game potentially worth investigation is one where red places the cathedral first, but black is then allowed to place their first piece. This would give red the advantage of placing the largest piece where they see fit, while equalizing this advantage by giving black the first move with their own pieces.

The machine learning algorithm used here was unable to discover the hypothesized 'solve' of *Cathedral* when starting with the red pieces, although this is likely due to the relatively low initial depth of the pre-computed trees. Just considering the first two moves of the game, there are over 350,000 possible first/second move combinations to consider. This combinatorial explosion of the tree was computationally intractable with the resources we had access to. It is entirely possible a higher-depth tree could organically discover the strategy that guarantees a win when playing with the red pieces.

5. Potential Improvements

There are potential improvements to this algorithm. The clearest area of improvement is in the expansion strategy, which, under this algorithm, simply picks a random next move to simulate. More advanced chess and go playing algorithms like AlphaZero instead use a neural network at the expansion step. The neural network is trained using reinforcement learning and can evaluate the strength of any given position, allowing for better moves to be explored [4]. In this case, the chosen expansion strategy of picking random moves instead of a more intelligent strategy such as a neural network is because training a neural network to be able to evaluate the board state is a difficult task and beyond the scope of this project.

Additionally, there may be other methods of representing the game board or simulating the game that are more computationally efficient, and thus would free up computing resources to allow for the tree to have increased depth, improving the algorithm strength.

6. Conclusions

We have shown that a machine learning algorithm can be used to effectively play the board game *Cathedral.* Further work would likely lead to a better performing algorithm.

All code is available at https://github.com/SeanRCox/cathedral-ml

References

[1] "Cathedral Game," [Online]. Available: https://www.cathedral-game.co.nz/about-how-to-play.html. [Accessed 10 June 2024].

[2] T. Lehmann, "Tom Lehmann," [Online]. Available: https://sites.google.com/site/ptlehmann/gaming/house-rules/cathedral-fix?authuser=0. [Accessed 10 June 2024].

[3] "Geeks For Geeks," 9 September 2023. [Online]. Available: https://www.geeksforgeeks.org/flood-fill-algorithm/#. [Accessed 10 June 2024].

[4] D. Silver, T. Hubert and J. Schrittwieser, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *DeepMind,* p. 19, 2017.

[5] L. Kocsis and C. Szepesvari, "Bandit based Monte-Carlo Planning," 2006.

[6] "Monte Carlo Tree Search (MCTS)," 23 May 2023. [Online]. Available: https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/. [Accessed 10 June 2024].

[7] P. Auer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Machine Learning,* p. 22, 2002.