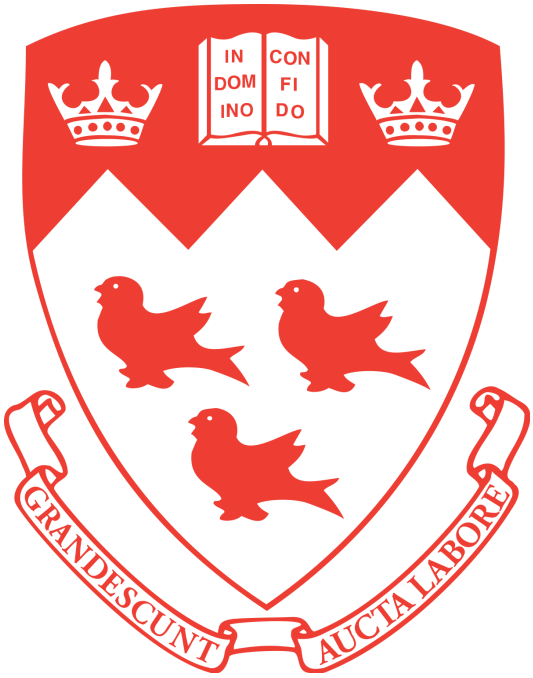
Saboteur Final Report

COMP 424 - Artificial Intelligence



Group 20
Alain Daccache - 260714615
Ketan Rampurkar - 26073283

Motivation, Pros and Cons and Alternatives to Monte Carlo Tree Search

In the literature covering game theory, there are some algorithms that leverage tree search in order to predict the best possible moves in some board games like Chess, Saboteur, Connect Four, Go… We will discuss Minimax, Alpha-Beta, and Monte Carlo Tree Search to shed light on the motivation behind using the latter.

While Monte Carlo Tree Search doesn't require any assumption about the game at all (since it only needs the possible playouts at each state and game-end conditions, and no evaluation function is necessary), we can consider it to be a simple AI agent that exploits randomness, but its naivety can be computationally and memory expensive. This certainly affects the extent to which it can explore and exploit the domain at hand, especially with a 2 seconds timeout and 500 mb of RAM, and thus impacts its decisions.

Other algorithms such as Minimax make use of the evaluation function in order to estimate what the opponent will likely do (rather than using randomness in MCTS). However, it still suffers the same fate of MCTS when it comes to a branching factor that can be big. But an improvement, Alpha-Beta pruning, uses that evaluation function to remove nodes where the player would end up in a worse situation, thus not expanding and exploring those nodes (and subsequent ones) in the first place, saving a potentially high amount of computation (especially with bigger branching factor). With pseudocodes widely available, this algorithm would be easy to implement, but the challenge lies in engineering a sound evaluation function.

Therefore, we decided to use Monte Carlo Tree Search (we did not experiment with another approach beforehand), which uses a universally applicable evaluation function a.k.a randomness. In fact, suppose at a given current state, we play as many random games as possible. Afterwards, if we see that the next move from the current state results in wins 75% of the time, then, without reverse engineering an evaluation function, we can conclude that due to the Law of Large Numbers, there is an inherent advantage to playing that move. Usually, this algorithm can be applied in a purely randomized manner, but knowledge of game specific techniques and rules can further prune out some moves, and thus for the same amount of time, being able to further explore and exploit the tree. Since we estimate the score of a node based on random moves, if we have more iterations, we will have more reliable estimates. Another advantage of using MCTS is that the algorithm can be interrupted at timeout (unlike other algorithms which need to complete) and still yield the most promising move so far.

Monte Carlo Tree Search Theory and Implementation

In our implementation of an AI agent for Saboteur, we have used the Monte Carlo Tree Search algorithm in order to make a decision during each turn of the agent in the game. This game is nondeterministic (replaying the same moves in a second game doesn't guarantee the same outcome, as stochastic outcomes

when drawing a card), but is partially observable (the player cannot observe the other player's card or the deck, but can observe the state of the board).

The structure of the code (data structures + function signatures) was inspired by the article ML | Monte Carlo Tree Search (MCTS) (GeeksForGeeks). The implementation was based on the pseudocode in Monte Carlo Tree Search (SemanticScholar), which will be described below.

Tree Data Structure

Once it is our agent's turn to make a move, it initializes a tree that it will build throughout the time allotted in order to make a decision. The root node of the tree contains the current state of the board (*State*), as well as important statistics used in Monte Carlo Tree Search, specifically the *Visited Count* and *Win Count*. Each subsequent node will also contain such information. To implement that tree in terms of data structure, each node also contains a reference to its *Parent Node* (implemented as a Node), as well as its *Child Nodes* (implemented as an array of Nodes). Our data structure is defined, and now we can move on to explaining the mechanics of Monte Carlo Tree Search algorithm in our Saboteur context. MCTS keeps repeating these four phases outlined until the time allocated is over (minus some margin of safety), after which it chooses the child node of the root which has the highest visit count, and plays the corresponding move.

Selection Phase

We will cycle through the next four phases until our time allotted is expired (minus a certain margin of safety in order to play the move after the algorithm picks it).

The first step is to perform Selection, which starts at the root node and keeps picking child nodes (thus traversing the tree) until it reaches a leaf node of the tree it built so far. Thus, at first, since the tree only contains the root node so far, that is the one it picks in this step (it stops here). Once we further build the tree (in the Expansion, next phase), it can go down further. Choosing which node to go to next is crucial as it will balance the "exploration-exploitation" tradeoff. To do so, this phase applies the Upper Confidence Bound (applied to trees) - in short, UCT - to each child node of the current, and picks the one that maximizes this value. The UCT is given by the formula below [1], from which the parameters can easily be found using our data structure:

$$UCT = \frac{w_i}{n_i} + c\sqrt{\frac{ln\,(t)}{n_i}}$$

Where the first part of the equation is the exploitation component, and the second part the exploration component, specifically:

- w_i is the total win count after the i'th move [child node] (node.*winScore*)
- n_i is the total simulation count after the i'th move [child node] (node.*visitCount*)
- c is the exploration parameter (usually $\sqrt{2}$)
- t is the total simulation count for the parent node (*node.parentNode.visitCount*)

As we can see, this formula balances between the *exploitation* of deep variants after moves with high average win rate (first component is higher for moves with higher win rate) and the *exploration* of moves with few simulations (second component is higher for moves with lower simulation count).

Once we reach the leaf node (which can be found by using our data structure i.e. **Node.getChildArray().isEmpty()** signifying it doesn't have child nodes), we go to the next phase.

Expansion Phase

Once we reach the leaf node, the agent can observe what are the possible moves in that state, and append to the leaf node, a node for each of those moves (i.e. the board state after making that move). There are countless moves the agent can make in theory, but due to constraints placed by the game, it is limited to some (that are legal and in our agent's hand). Also, with game-specific knowledge (i.e. strategies), we can further prune down that domain.

The possible decisions at a certain turn depend on the state of the board and the cards the player holds. Rules of the games specify the legal moves, thus adding constraints to our search space. By better understanding the game mechanics, we can further prune it out. However, it seemed unfair to prune out some moves that seem not to provide any short-term advantage as this would just convert this algorithm to a greedy one (further domain knowledge is required). Therefore, we decided to reflect those heuristics in our tree by artificially inflating the win rate of such moves when initializing the child nodes, or adding to the UCT formula another fraction, $\frac{b_i}{n_i}$, where b_i is a heuristic score for the i-th move [2]. We performed the latter, and we explain below.

Monte Carlo methods do not perform as well for games with imperfect information such as this one. We thought it would be convenient to have a pre-defined strategy that we would run at first always (before calling MCTS), but after some research, realized we could use those conditions to set a heuristic score to each node of our Monte Carlo Tree during expansion, without necessarily "hardcoding" a strategy, but simply giving some nodes an extra push in order to exploit them (as experience has shown some moves would be better than others).

- For **Malus**, it is a sounder idea to play it towards the end, because at the beginning we are both working towards reaching the hidden tiles, but at the end we want to prevent the opponent from doing so. Plus, it is more likely that the agent would have dropped Bonuses throughout the game because they are not as high of a priority, so it may be scarce of it by the end.
- For **Destroy**, it matters more which card are we destroying, so we set a heuristic score according to that. However, it also matters not to sabotage ourselves, and the path we are building. An improvement here is to destroy the cards that do not allow a path to form between entrance and the nugget (especially if it is found). Currently, we are only considering deadends, but some other cases can arise when other cards are blocking our way that we should destroy so we can legally make a move that is beneficial. This lack of considerations adds a point of failure.
- For **Map**, it would matter way more to play it when the nugget is not revealed yet, and would not have any use otherwise.

- For **Bonus**, if it is a legal move, then we are under Malus and we should play it anyway. We haven't considered the tradeoff between playing Bonus or Map in that turn.
- For a **Tile**, we assign a heuristic $Score = \frac{Priority}{Distance} * Orientation$ that takes into account variables that in a sense complete one another:

Priority: we split them into three categories of priority i.e. the more versatile ones (priority 1), the deadends (priority 3), and the rest (at priority 2). However, priorities of tiles (and cards in general) shouldn't be static and depend on other variables such as the state of the game, so that can be an improvement for the future. For instance, playing the cross card (tile 8) might be preserved for the end as it is a more versatile card. However, we have reached a tradeoff: (1) preserving them in the beginning but being less likely to reach the hidden cards area (since the Random would be building a random path, so the versatile moves we are saving are not allowing it to play as many move), or (2) using them to reach that area, but having less versatile cards in the end, leading to outcomes of whether the random player has the card that fits, or drawing paths around the hidden tiles, ending in a draw.

Distance: using only priority, we can have favorable moves but at unfavorable positions, so we should also use the distance of the tile to the golden nugget (or middle hidden tile if not found yet). The less the distance, the better the overall score (inversely proportional). A better way would be computing the Manhattan distance (knowing our cards and estimating the cards left to draw using knowledge of the cards played) from all possible paths and picking (and sticking to) the least OR the one that opens the most paths (testing needed).

Orientation: Many moves can have the same distance to the golden nugget, but some would not fit as well as others to create a path from the entrance, while some would. This heuristic considers whether the open end of a tile is in the rectangle formed by the extremities of the entrance and hidden tiles (this heuristic would return 1), in order not to give preference to tiles placed in another direction (this heuristic would return 0).

Simulation Phase

In this phase, we pick a random node from those created in the previous phase. We can do so by picking a random index for the *ChildArray* of the select node of the first phase. Then, we simulate a playout by alternating turns between the agent and the opponent. We can pick a random legal move given the board state, and do so until the game is over (either at a win, draw, or loss).

At first, it seemed intuitive that the simulation phase of MCTS would not yield promising results. With a high-branching factor up to (and including) the endgame and picking random moves, the simulation would end in a lot of draws, backpropagating results into nodes without giving either some competitive edge. We can produce more realistic playouts by picking smarter moves than random, with a better knowledge of the game (using our heuristic score for (a) this move in (b) this board state). However, this was effectively reducing the number of iterations of our algorithm for a specific move, as it would require sorting after each move in Simulation. Another improvement would be running many random simulations concurrently in order to reach a more realistic estimate of the outcome of playing a specific move.

Backpropagation Phase

The result of the playout will be returned by the previous phase, and can be used to start backpropagation. In this phase, the algorithm traverses the tree upwards from the current node to the root, while incrementing the visit count for the visited nodes, as well as the win score for each (if the player for that position has won the playout). Our code was inspired by a snippet from Monte Carlo Tree Search for Tic-Tac-Toe Game (Baeldung).

By adding a heuristic, we modified the original Backpropagation in order to propagate a score proportional to each node's heuristic, which has effectively improved the performance of our agent by 9% (to 49%). Some improvements definitely need to be made in order to more fairly compute the UCT: fine tuning of the heuristic scores, the exploration parameters, and the backpropagation score, in order to more fairly balance exploitation and exploration.

Improvements can also be made by executing threads concurrently, described in [3]: parallel execution of many playouts from one leaf of the game tree, or building independent trees in parallel, or parallel building of the same tree (using either global mutex, local mutexes, or non-blocking synchronization). Phases that need to access or modify shared memory (such as selection, expansion, and backpropagation) will need to be handled in a way (locks, mutexes, spinlock…) to ensure consistent data among the threads.

References

[1] Kocsis, Levente; Szepesvári, Csaba (2006). "Bandit based Monte-Carlo Planning". In Fürnkranz, Johannes; Scheffer, Tobias; Spiliopoulou, Myra (eds.). *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18–22, 2006, Proceedings*. Lecture Notes in Computer Science. **4212**. Springer. pp. 282–293. CiteSeerX 10.1.1.102.1296. doi:10.1007/11871842_29. ISBN 3-540-45375-X.

[2] G.M.J.B. Chaslot; M.H.M. Winands; J.W.H.M. Uiterwijk; H.J. van den Herik; B. Bouzy (2008). "Progressive Strategies for Monte-Carlo Tree Search" (PDF). *New Mathematics and Natural Computation*. **4** (3): 343–359. doi:10.1142/s1793005708001094.

[3] Guillaume M.J-B. Chaslot, Mark H.M. Winands, Jaap van den Herik (2008). "Parallel Monte-Carlo Tree Search"(PDF). *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 – October 1, 2008. Proceedings*. H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, Mark H. M. Winands (eds.). Springer. pp. 60–71. ISBN 978-3-540-87607-6.