Group 98: Saboteur Game Report

Yanan Wang and Wenwen Xu

April 12th, 2020

Abstract

Saboteur is a mining-themed game. [1] Players are given a hand of cards which are path and action cards. In each turn, the player can play one card from their hand and draw a new one from the deck. Their goal is to reach the treasure by building a tunnel to the nugget before running out cards from the deck. In this project, we designed an agent which played against the random player in a modified version of Saboteur game. We implemented both the Minimax algorithm(with alpha-beta pruning) and the greedy algorithm, and we will discuss their pros and cons in the report.

1. DESCRIPTION&MOTIVATIONS

In order to beat the opponent, our agent would need to build a connected path from the origin to the treasure. This process is achieved by placing the path cards. At the same time, both players can choose to interfere with others by using the action cards. The validity of the card usage is checked by a given method "*getAllLegalMoves*". We would use this method as well as the current board state to decide our next move.

Firstly we will assign each card with a score. This is done through a method, "*EvalMove*". The score of the card is determined by its shape, (ie. path cards with more possible moves will have a higher value) and its functionality (ie. Map cards are given high value). Then we will filter the moves with bad card value out of all legal moves. This process will help to decrease our branching factors in the minimax tree.

With the filtered moves, we proceed to the next step which evaluates the outcome brought by each move. The class "*SaboteurBoardStateCopy"* plays the role of *SaboteurBoardState*'s clone. Its constructor will take a *SaboteurBoardState* object as input, copying its current board, turn player, turn player's hand and try to guess the nugget position, current deck and opponent's hand by inference. Class *SaboteurBoardStateCopy* also retained the method *ProcessMove* and other useful methods in *SaboteurBoardState*. We call *ProcessMove* on the *SaboteurBoardStateCopy* object, and then give the copied board an evaluation based on it's open path's deepest end to the nugget. The greedy approach will immediately return the move which can result in the highest board state value, while the minimax algorithm will expand the current board state, and
return the move which has a higher winning chance taking its opponent into account.

2. EVALUATION FUNCTIONS

Since Saboteur is a card game, we need to evaluate both the board state and the move of a player.

2.1 Move Heuristics

A move is determined by a card and its position. The cards in this game are classified into: 15 kinds of Tiles, Map, Malus, Bonus and Destroy. When they present in the possible moves of a turn, we give the card and its position different scores so that we can prioritize a move which is more likely to connect a longer tunnel for the player himself or restrict the move of the opponent. We assign points as below:

Bonus	10000
Map	500
Malus	100
Open Tiles *0,5,6,7,8,9,10*	20
Closed Tiles *1,2,3,4,11,12,13,14,15*	-2000
Drop	-10
Destroy	0
Postions with row 4	-10
Positions with row 0-3	-100

Figure 1. The card value heuristic

When **Bonus** exists in the possible moves, we are supposed to play it as early as possible to proceed future moves as we know the opponent has used Malus. Then we give **Map** the second highest score because when we reveal a hidden card, we will incline to place future cards towards it or the other two cards (which is calculated in the boardstate heuristics). Moreover, we take into account the position of the card: we favor the card which is placed beneath the entrance. Otherwise, we will return a negative value when it is placed on or above the 4th row. At last, we classify the tiles into two categories based on their patterns. If a tile can block an existing tunnel, such as Tile 1, we deduct points. Otherwise, for tiles like Tile 8, we put more weight on such moves.

2.2 BoardState Heuristics

To evaluate the connectivity of a state, we apply the DFS algorithm on the int board such that we can find the deepest, opening end and the spanning area of 1s (*numOnes*) from the entrance. Basically, the strategy includes exploring more possible paths and digging down to the target as deep as possible. With the deepest end coordinates, we can compute the manhattan distance from it to three hidden targets, if no Map card has been used. If one of them is revealed as a nugget or two of them are revealed as normal Tile 8, we only compute its distance to the nugget. In short, this heuristic helps the agent to make moves toward the nugget as much as possible so that the distance will be shorter and shorter. The distance is represented as *disToThreeHidden.*

The final evaluation function is:

$$f = 10 * (-disToThreeHidden) + numOnes$$

For example, with the board state as Figure 2.a , our move heuristics will give priority to card 1 and 5 in all possible moves. Combined with our board state evaluation function, our agent will choose to place the 5th card in its hand on (6,6) as it will increase the number of ones. It is less helpful to make a move to the left since our agent revealed the left hidden card is not a nugget with using a Map card before this move.

Figure 2.a

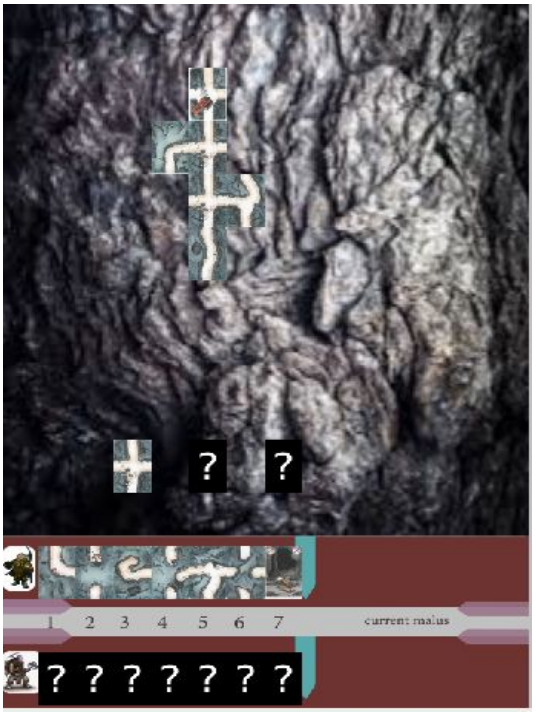

Figure 2.b

3. ALGORITHMS

3.1 Minimax Algorithm

Minimax is a backtracking algorithm for finding the optimal strategy, it will minimize the possible loss in the worst case. [2] There are 2 players present in minimax, one is the max player which tries to maximize its scores while the min player tries to minimize its scores. Starting with the current board state, it will expand the search tree by switching between max and min players. It will stop until terminal states have been reached, which happens when there's a winner found, the hand is empty or it reaches to our depth limit.

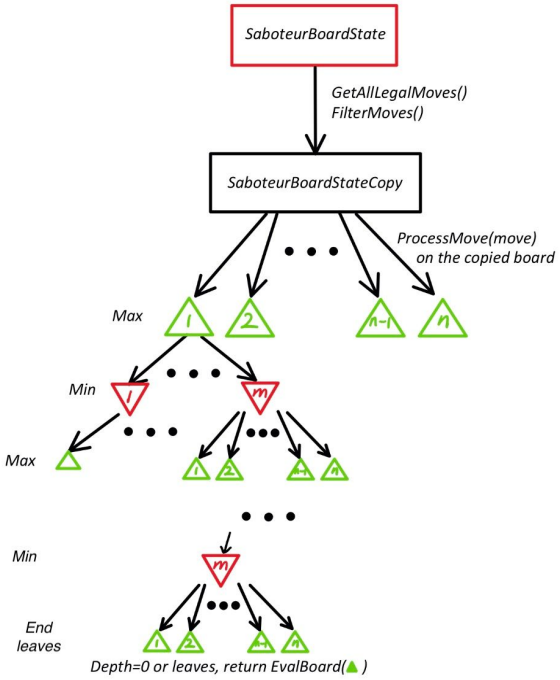

Figure 3. Minimax tree used in the Saboteur game. The min player(*RandomPlayer*) at the second last row will choose the leaf that generates the least value. The max player(*Agent*) on the third row will choose the child which generates the highest value. Our *ChooseMove* will return the move which has the highest value processed by the minimax algorithm.

3.1.1 Alpha-Beta Pruning

Alpha-Beta pruning is an optimization technique which helps to prune the tree in the minimax, since it's super expensive to construct a complete search tree. It will add 2 more parameters, [α, β] , where they are the best values that max player and min player currently can guarantee at this level or above. [3] Their initial value will be [-∞, +∞] respectively. It will prune when α >= β has been found.

3.2 Greedy Algorithm

Greedy algorithm is an intuitive algorithm that is widely used in optimization problems. It simply takes the optimal choice at each step with the intent of finding a global optimal. [4] In our case, we will choose the move which results in the highest board state value.

4. RESULTS

4.1 Minimax Algorithm

4.1.1 Results of Running 100 Rounds.

Number of Lose	Number of Win	Number of Draw	Winning Rate
0	*0*	*100*	*0%*

Figure 4. Minimax result with depth 4, branching factor=10. Branching factor is the maximum possible number of children allowed for each node.

4.1.2 Result Analysis

Using minimax can be a good choice when we have perfect information about the game. For example, applying minimax to the game Go or Tic-Tac-Toe. These games are what we categorized as *deterministic* games. They have no randomness associated with them and each action will lead to a clear outcome. [5] Minimax can lead to a good winning chance under this setting. As shown in the table, minimax is having a poor winning rate result. The poor performance is caused by the following reasons :

1. Non-deterministic: In the Saboteur game, randomness is involved in drawing the card from the deck. As the opponent's hand is unknown to us, we can't get the perfect information and then do the correct inference. Our current approach is to 'guess the opponent's hand' by removing the used tiles that appeared on the board. However, we have no ways to obtain the usage of Map, Malus, Drops cards. We can never predict the opponent's hand accurately.

2. Partial search tree: The results obtained in the tables used tree depth=4, branching factor=10. These values are the highest values which guarantee to generate the moves in 2 seconds. Given a large branching factor and depth limit will cause a big game tree. Thus this partial search tree will bring bad results due to incomplete information.
3. Opponent's heuristic: When we expand our game tree, we assume our opponent takes the same heuristic as our agent. This is a bad assumption because random players are not optimizing anything and may have unpredicted strategies.

4.2 Greedy Algorithm

4.2.1 Results of Running 100 Rounds.

Number of Lose	Number of Win	Number of Draw	Winning Rate
14	*54*	*32*	*54%*

Figure 5. Greedy approach has a winning rate 54%

4.2.2 Result Analysis

Greedy algorithms have several advantages [6]:

1. Simplicity: It's easy to come up with and code up greedy algorithms.
2. Efficiency: Greedy algorithms can often be implemented more efficiently than other algorithms. We don't need to build a search tree as in minimax.

The disadvantage associated with greedy search is also very clear:

1. It fails to find a global optimal solution: The highest value move may not always lead to an optimal solution. For example, when a destroy move and a cross-tile move are present at the same time, *evalBoard* will give more marks to the outcome brought by the cross-tile move. This may not be optimal as sometimes destruction can give a better result.
2. No foreshadowing: It only depends on the moves it made so far, but is not aware of the outcome brought by the choices. Suppose we are just 2 tiles away from the nugget. If we use a tile that is the cross-tile one, our opponent will have more chances to win as our cross-tile maximizes its moves.

In general, greedy algorithms yield better results and we use it as our final approach.

4.3 Heuristic-inherited Errors & Weakness

Instead of the error brought by the drawbacks of the algorithm, there are also some errors inherited in our heuristic which affects the results algorithm, there are also some errors inherited in our heuristic which affects the results.

4.3.1 Manhattan Distance

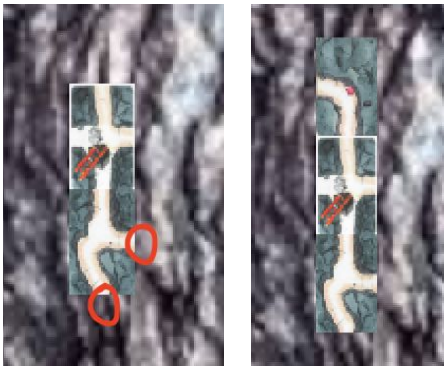

Figure 6 .Incorrect placement of tiles.

In figure 6, instead of placing the tile at the bottom, our agent is placing it on the top. Since Manhattan distance is simply the difference between x and y. If our nugget is on the rightmost. The 2 red dots circled in the graph will result in the same distance to the nugget, then our agent cannot decide the best move and will place the tiles randomly.

4.3.2 Card's Heuristics

To decrease our branching factors in the game tree, we are evaluating each card with our own inference(i.e cards with more open ends are prioritized). This biased strategy may work when played against a *RandomPlayer,* but it may fail when played against more intelligent agents.

5. DISCUSSION & FURTHER IMPROVEMENTS

Both greedy and Minimax algorithms rely heavily on a reasonable evaluation function. During our implementation, we have encountered some challenges to design useful and well-representative features given a large number of board states and move choices. In the future, we may focus on developing a more robust and Saboteur-specific evaluation strategy to improve performance. In addition, we managed to filter moves during the game tree expansion because minimax is very computationally expensive even with alpha-beta pruning. Therefore, it is also worthwhile to design better criteria to select more potential moves to speed up our implementation.

What we have not attempted but witnesses great success in AI game playing is deep learning algorithms. Reinforcement learning will make the agent learn better by associating different stimuli and movements with reward and punishment policies.

In conclusion, we have proposed our automatic game playing agent which is able to achieve around 60% winning rate against a random player. With our thorough analysis of our algorithms pros and cons, we now have a clearer picture of improving our agent's performance and optimize our simulating process.

REFERENCE

[1] Saboteur Board Game (2004). https://boardgamegeek.com/boardgame/9220/saboteur

[2] Chua Hock-Chuan (2012) : Case study with tic-tac-toe : part2 AI. https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaGame_TicTacToe_AI.html

[3] Julio Cesar Aguilar Jimenez. (2018). Search Heuristics for Isolation. http://ajulio.com/assets/documents/Adversarial_Game.pdf

[4] Karleigh Moore, Jimin Khim, and Eli Ross. (2016): Greedy algorithms https://brilliant.org/wiki/greedy-algorithm/

[5] Baeldung. (2018). Introduction to Minimax Algorithm. https://www.baeldung.com/java-minimax-algorithm

[6] Greedy Algorithm. Page 7 https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/lectures/13/Small13.pdf