Saboteur Game Modelling Using

Value-Based Opening and Modified Monte Carlo Tree Search

Nghi Huynh 260632588, Krystal Xuejing Pan 260785873

Abstract

Saboteur is a card game played by several players. Players are given a mixed hand of path and action cards and they take turn consecutively. Each player can play a card from their hand or discard it and collect a new one from the deck. The goal of this game is to build a tunnel from the entrance tile to the nugget location which is hidden and fixed on the board. For the simplicity of this project, the board size is restricted to 15x15. Saboteur is classified as a game with imperfect information since the nugget is hidden from players at the beginning. Moreover, player moves are stochastic since the board state are partially determined by chances. This paper proposes an approach to model an Artificial Intelligent agent for Saboteur using a value-based opening followed by a modified version of Monte Carlo tree search. Implementation was done in Java language. To test the performance of our agent, we performed AutoPlay between our agent and a random player. The result is that our agent wins more game and performs much better than a random player.

Introduction

Saboteur is a mining-themed card game, designed by Frederic Moyersoen and published in 2004 by Z-Man Games. Our goal is to build an Artificial Intelligent agent such that it can decide the best move to open a path from start to the nugget location and win the game. We have had a lot of challenges throughout the process since our agent has no knowledge regarding other players' hand and future cards on its hand. Several approaches such as Means-Ends Analysis and Heuristics, Monte Carlo Tree Search and Alpha Beta search have been used to solve for the incomplete information issues (Dion Krisnadi, 2019) In this paper, we propose a new approach to this problem by using a value-based opening followed by a modified version of Monte Carlo tree search (MCTS).

Section I: Approach

After playing and observing multiple games, we decided to separate the program into two phases: opening and MCTS. In the opening phase, we assigned values to different tiles based on its position and its distance toward hidden objectives. Moreover, we also prioritized some moves such as Map, Malus and Destroy based on the current board state. In the MCTS phase, we used an evaluate function and upper bound confidence to generate the best move among all legal moves for our agent.

Section II: Methods

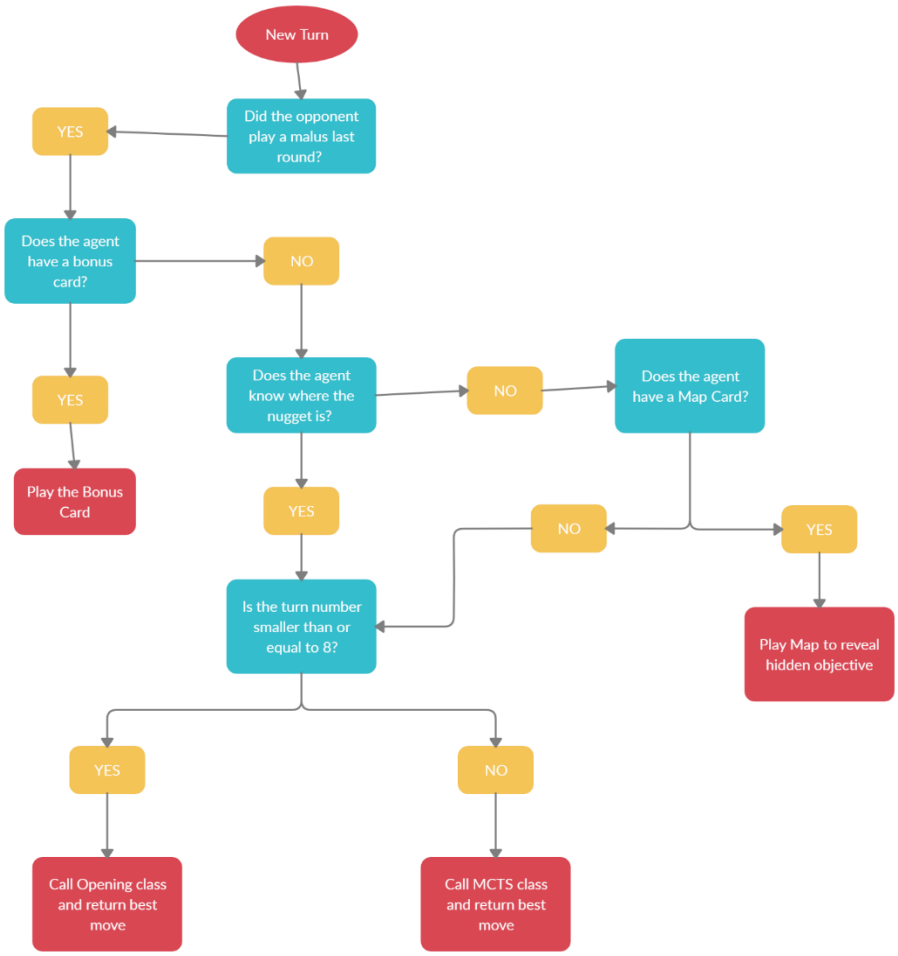
1. Student Player Class

The `StudentPlayer` class is the master class for the student player agent and has the grand structure (Figure 2.1).

There are two moves that have priority over all other algorithms. The move that has the highest priority is playing a bonus card when the opponent plays a malus card. The move that has the second highest priority is using the map card when agent hasn't found the gold nugget. When the golden nugget is not revealed, agent checks for the left hidden objective first, then checks for the right objective. If two hidden objectives are revealed, agent knows that the third one is the nugget.(It is important to know where the nugget is for the following algorithm) If it does not know where the nugget is and has a Map card, it plays the card.

If none of the two moves are available, the agent checks for turn numbers to determine which of the two algorithms we employ for this move. If the turn number is smaller than 8, the agent calls the opening class and employs the opening strategy. Otherwise, the agent employs the MCTS algorithm with UCB strategy, which are explained in detail in the following sections.

Figure 2.1: A decision system that our agent uses



2. Opening Class

The agent employs a value-based opening strategy for the first four turns. The goal is to get the agent closer to the center and down. The strategy is to give each legal move different values, and the agent picks the move with the highest value to play.

Positions in row 5 to 8 and column 3 to 7 are each given different values (Figure 2.2). The positions get a higher score when they are center and closer towards the objectives. Each tile we deem functional are given three different values (Figure 2.3). These three different values depend on how beneficial they are in general, and where the current legal move is. The main method, `getBestMove`,

takes a board state and returns the move with the highest value(Algo. 2.4). Each tile is given a value based on the `getCardVal` method (Algo. 2.5).

Figure 2.2: Position Values

	3	4	5	6	7
5	300	400	0	400	300
6	600	700	800	700	600
7	700	800	900	800	700
8	800	900	1000	900	800

Figure 2.3: Three sets of tile values

Tile name	10	7-f	7	9-f	9	5-f	5	8	6	6-f	0
TileValSet1	20	60	0	0	70	0	60	100	90	90	80
TileValSet2	50	70	0	60	80	40	0	100	0	90	60
TileValSet3	50	0	40	60	80	0	70	100	90	0	60

Algo. 2.4

```
Public SaboteurMove getBestMove(SaboteurBoardState boardState){
    For all legal moves
        If the card is Destroy or Malus
            Do nothing
        Else if the card is Map
            If the nugget is found
                Do nothing
            Else if left hidden objective is not revealed
                Play Map on left hidden objective
            Else if right hidden objective is not revealed
                Play Map on right hidden objective
            Else
                Play Map on middle hidden objective
        Else if the card is Tile
            If current move plays between column 3 and 7, row 5 and 8
                Get the position value for the tile move >> posVal
                getCardVal of this tile >> tileVal
                TotalVal = posVal+tileVal//the total value of this tile
    Return the move with the highest total value
```

```
Alg. 2.5
public int getCardVal(String tileName, int ColumnNum)
        If the tile name is on the functional tile list
                If ColumnNum = 5
                        //if the tile in the same column as the nugget
                        Use tileValSet1
                        Return corresponding value
                Else if ColumnNum < 5
                        //if the tile position is to the left of the nugget
                        Use tileValSet2
                        Return corresponding value
                Else
                        //if the tile position is to the right of the nugget
                        Use tileValSet3
                        Return corresponding value
        Else
                Return -1; //not functional tile, no value
```

3. MCTS – IC – E (UCB)

We employed the algorithm of Monte Carlo Tree Search with Informed Cutoff with Evaluation Function with Upper Confidence Bound (MCTS-IC-E with UCB). There are two components: a) the MCTS algorithm with UCB and b) the evaluation function.

a. <u>The MCTS algorithm</u>

The MCTS algorithm we implemented consists of four critical phases: selection, expansion, simulation, and backpropagation. The selection phase starts at the root and selects a board state to move on. The selection is based on the Upper Confidence Bound (UCB) formula:

$$ucb_{i,s_i} = v_i + c\sqrt{\frac{\log N}{n(i)}}$$

where v_i is the value of taking action i in state s_i, n_i is the number of visits at state s_i, N is the total number of visits of the parent node of n_i, and c is a constant for fine tuning. Once computing the UCB for every board state, the one that has the highest UCB value is chosen. Since our agent has no knowledge about the next move or the opponent's move, expanding the board state until it terminates is not possible. So, in the expansion phase, the algorithm expands a board state based on all the legal moves that the agent

currently has instead. To assign value in the simulation phase, we used an evaluate function to evaluate the move and how close it is to the target. Then the back propagation gets the values of Simulation and updates these values to the root (Algo. 3.1).

Algo.3.1: MCTS algorithm:

```
Function MCTS (boardState, tree):
        For i=0, … ,15: //repeat 15 times
                state <- selection (tree)
                expansion (state, tree)
                simulation (tree)
                backpropagation (tree)
        end for loop
        return bestAction (state, tree)
```

b. Evaluation Function

The evaluate method in EvaluationFunction class evaluates all the legal moves from a given board. It employs a similar concept to the Opening class, with some added details. It has the same three sets of values to functional tiles, and three different sets of values for all 15 x 15 positions in the board based on where the nugget is. The closer the position is to the nugget, the larger the value is for it.

There is an added conditional statement regarding Destroy cards. If Destroy card destroys a non-functional tile in critical position, it is considered a potential move. If the move of Destroy is in between row 11 and 13, column 3 and 7, and the tile it can destroy is NOT in our functional tile set, agent deems this move significant and gives it a value of 10.

Another new conditional statement is for Malus cards. If the turn number is smaller than 10, Malus card move is considered insignificant. A turn number grows, the value for Malus card gets larger and larger. Thus, agent is more likely to play malus towards the end of the game.

For legal moves of Tiles, the basic evaluation process is similar to the Opening. The values assigned to legal moves of functional tiles is the sum of a tile value and a position value, both depending on where

the objective nugget is. Agent also checks if the current tiles played is connected to the starting tile. If it is, it is given a bonus of 100; if it is not, the value is decreased by 10.

The evaluate method returns an integer array of evaluated values corresponding to the order of all legal moves, which is used by the MCTS method.

Section III: Advantages and Disadvantages

The advantage of this approach when compared with pure MCTS or random player is that it is fast and efficient but not required lots of memory. Moreover, by separating the opening and the MCTS phase, our agent improves its win rate up to approximately 42% while reducing its draw rate to 52% compared to older versions which were only 30% for win rate and more than 60% for draw rate (Table 1,2). However, the disadvantages are that there are some tiles with the same values after the simulation phase in the MCTS and these values generate the same value when calculating UCB; thus, it results in sub-optimal moves for our agent. Moreover, the expansion phase only expands a board state based on all legal moves but not on termination state, it results in a biased evaluation for that state.

Table 1: **Our agent player plays against a random player in our current version (based on 1000 games on each test case)**

Test case	Random player win rate	Agent player win rate	Win ratio
Agent player vs Random player	8.7%	42.8%	1 : 4.92
Random player vs Agent player	7.4%	40.6%	1 : 5.47

Table 2: Our agent player plays against a random player in our previous version (based on 1000 games on each test case)

Test case	Random player win rate	Agent player win rate	Win ratio
Agent player vs Random player	7.0%	33.9%	1 : 4.84
Random player vs Agent player	7.2%	31.9%	1 : 4.43

Section IV: Potential Improvement

There are many possible improvements to ameliorate agent's win rate. The evaluation function is critical to the performance of the agent, and the model we are using are flawed. The values assigned to position and tiles are highly subjective, resulting in less rational behaviors. Also, many moves sum to the same value, leaving the MCTS algorithms uncertain. A potential improvement would be using priority queue with custom Comparator class, so that all moves are in preferences order.

Another possible improvement is considering the opponent's move. In our algorithm, we only take in consideration of the current board state. However, if we could track the opponent's move, at the end of the game, we will be able to utilize Malus and Destroy more effectively. Thus, preventing the opponent to build the last tile to the gold nugget.

The final potential improvement is to use a different algorithm than MCTS. MCTS is effective for non-deterministic, stochastic and imperfect games. However, once the nugget is revealed in Saboteur, the game changes into a perfect information game. Random rollouts are not beneficial in this case. An improved algorithm would be the means-ends analysis, which limits the search space by iteratively evaluating the differences between current state and goal state and apply the operator at each difference, which reduces the differences.

References

Baier, Hendrik, and Mark H.M. Winands. "Monte-Carlo Tree Search and Minimax Hybrids."

Baier, Hendrik, and Mark H.M. Winands. "MCTS-Minimax Hybrids with State Evaluations." *Journal of Artificial Intelligence Research*, vol. 62.

Krisnadi, Dion, et al. "Decision System Modelling for 'Saboteur' Using Heuristics and Means-Ends Analysis."