Comp 424 Final Project Report

Authors: Linda Cai (260720706) , Yilin Jiang (260795889)

Group Number: 93

McGill University

Introduction

Saboteur is a card game designed by Fréderic Moyersoen, and it was published in 2004. It is a multi-player game where players are assigned into roles of goldminers and saboteurs who prevent goldminers from finding gold. It is in the category of card games with incomplete information for each player and non-deterministic outcomes, so it requires an intricate decision-making process for the agent to win the game.

Artificial Intelligence can give the agent the required knowledge of the background, heuristic to optimize moves in order to act intelligently in the game and maximize the win rate against the opponent. So the goal in this project is for us to build an intelligent AI for Saboteur that defeats the random AI while fulfilling the designated requirements of the project (Turn Timeouts, Memory Usage, Multi-Threading, etc.)

We will walk through the project in this report in the following order:

1. Program layout and function, and motivation for the approach
2. Elaboration on the theoretical basis of the approaches chosen, and their relevance from in-class concepts.
3. List of disadvantages and advantages with such approaches, and weakness of the program.
4. Other approaches attempted and reasons why they were not chosen
5. Possible enhancements for the player by AI techniques, etc.

1. Program Layout

Here is a rough description about the classes used in our program:

BoardState: A class describing current board game state, similar to the provided SaboteurBoardState class. It is used to simulate the board state after a move is processed.

AndOrNode: The parent Class of AndNode and OrNode. It contains a method to check if a certain position in the int grid is an open end.

ORNode: An or node that represents the board state after a previously chosen move is processed, not considering which card is drawn from the deck. Compared to AndOrNode, it has additional attributes including SaboteurMove move,a list of constants used in heuristic evaluation function and a list of AndNode successors.

AndNode: A chance node that represents a single possible environment outcome that may exist for a given OrNode. It has the following additional attributes compared to AndOrNode: SaboteurCard dealedCard, double dealedCardProb and a list of OrNode successors.

StudentPlayer: This class contains the chooseMove function and a few helper methods that copy important information from the input SaboteurBoardState param into a new BoardState instance.

2. Technical Approaches and motivation

Before describing technical approaches used, we'll first explain how we relate Saboteur gameplay mechanics to artificial intelligence concepts. We define state as a composition of three parts, the grid with correctly placed saboteur tiles, two sets of cards in both players' hands and the remaining cards in the deck. The action space is defined as the set of all legal moves possible for the current state. The goal state for one player is there's a path from the entrance to the gold tile on the grid and he/she is the player of this turn.

The approach we have applied to tackle this game can be broken down into three crucial parts. The first part is inference of the exact goal tile position from three possible positions. The second one is the design of a good heuristic function. The third one is using chance nodes to handle stochastic outcomes induced by randomness of drawing the next card from the deck.

The exact position of the goal tile is very important as it determines the goal state. By default, if the goal tile is not revealed by the current player, the goal tile is assumed to be the middle tile in the three possible tile positions. In this way, if the guess is wrong, the correction needed to reach the left or right goal tile is relatively small. Inference of goal tile is achieved by using revealed map tiles and detecting if there's a path from entrance to a possible goal tile position but the game has not ended. Map moves have highest priority in all legal moves if exact goal tile position is not found. They are used on the left most possible tile first then middle, then right.

Heuristic function in our game is different for beginning of game and end of game. The way we differentiate beginning of game and end of game is whether any saboteur tile is placed on the row below and including 10. In both settings, map moves are given highest priority if the goal tile is not determined. In open game, we used the following heuristic function:

h = $min_{i,j\in[0,13]}$ {(exists a path from current position to entrance)? w1*Math.abs(i-goalPosX)+w2*Math.abs(j-goalPosY):Integer.Max_Value-+(intBoard[3*i+2][3*j+1]==1)?w3:0} + w4 * (open ends) + w5 * numOfSelfMaluses+ w6 * numOfGoodTilesAbove5 +w7 * numOfOwnedMalusesCard, **where** *w1 = 1000, w2 = 200,w3 = -300, w4 = -50, w5 = 40000, w6 = 100, w7 = -50000*

This heuristic function takes into consideration the distance to goal tile, the smaller the value is, the closer the current state is to the goal state. w2 is smaller than w1 because we want the agent to place tiles as vertically downward as possible. We gave tiles with a open end at the bottom line a bonus of -300 so that if two tiles are in the same row but differ by one column, the one with a open end at the bottom line is preferred. We also gave states with more open ends bonus so that there are more available paths to the goal tile. Besides, we have taken care of maluses by giving 40000 punishment each self malus and -50000 bonus for each malus card in the current player's hand. Lastly, good tiles(0,5_flip,5,6_flip,6,7_flip,7,8,9,9_flip,10) placed above row 5 receive a punishment of 100.

In end of the game, we used a slightly different heuristic function:

h = $min_{i,j \in [0,13]}$ {(exists a path from current position to entrance)? w1*Math.abs(i-goalPosX)+w2*Math.abs(j-goalPosY):Integer.Max_Value-+(intBoard[3*i+2][3*j+1]==1)?w3:0} + w4 * (open ends) + w5 * numOfSelfMaluses+ w6 * numOfGoodTilesAbove5 +w7 * numOfOpponentMalus, **where** *w1 = 1000, w2 = 1000,w3 = -1800, w4 = -50, w5 = 40000, w6 = 100, w7 = -50000*

The change in value of constants w2, w3 is to emphasize that horizontal distance and vertical distance in the end of game are equally important. Besides, we changed the last term from owned malus cards to opponent maluses in order to block opponent's tile moves.

The third crucial part is the use of chance nodes. The reason we chose to use chance nodes is that the next card is drawn randomly from the deck. For a given agent move, there are many possible environment outcomes. We use chance nodes to represent these possibilities (chance node is the AndNode class in our program). Chance nodes are used together with depth-limited search to search deeper in the search tree as the long-term heuristic value is different from current state heuristic value. Searching deeper from 1 level to 2 level has improved the winning rate of our agent versus random player from 47% to 60%. The exact implementation is explained in the theoretical basis part.

3. Theoretical Basis

We have applied the following techniques learned in class. The first one is informed search with heuristics. Since the branching factor is huge for the enormous action space with roughly 50~90 actions possible after 15 turns, it is impractical to search through the game tree until goal state is reached. Instead, we used a heuristic function to evaluate how close the resulting state from a processed move is to the actual goal state.

The second one is applying chance nodes to handle stochastic outcomes. The use of chance nodes is illustrated in figure 1. The heuristic value of an ORNode evaluated by the following function:

```
If (curDepth == maxDepth || ORNode board state is goal state) then
    h = current Heuristic Value
Else
    h = $\sum_{a \in AndNode\ descendants} p(a) * max\{heuristic\ value\ of\ a's\ ORNode\ descendants\}$
```

We could have used the expectiminimax to make move decisions based on the opponent moves. However, we didn't implement this as the tree nodes expand too quickly.

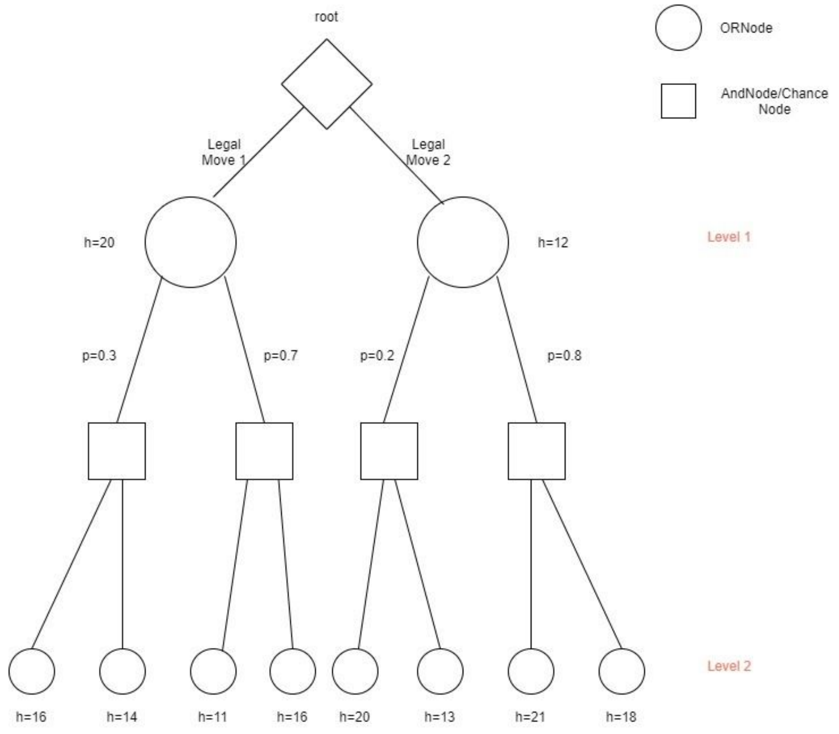

Figure 1: Searching with chance nodes

The third one is depth-limited search. At the start of the project, we used depth-limited search with a cutoff depth of 3 levels from the root ORNodes. However we find out this is extremely time-consuming and usually takes 25 seconds to finish. As a result, for the final version, we used a cutoff depth of 2 level from the root ORNodes and if the program is running out of time, return the current best ORNode's move. It is worth mentioning that we first sort the first level ORNodes in ascending order according to their immediate heuristic values as they have the highest probability to have good expected heuristic value in long turn.

4. Advantages & Disadvantages

Search Tree with chance nodes[1]:

With the presence of uncertainty in the opponent's moves, a search tree with chance nodes is the strategy chosen in this case. The advantage is that we keep track of our move, which is deterministic while keeping assumptions for environment outcomes, which is stochastic. In the game, we must attempt to infer the possible cards in the deck although there's imperfect information for the current player. The more information we have, the more accurate the prediction, and the clever our agent is. We accomplish this part by first getting complete deck composition from SaboteurCard, and then eliminating placed tile cards on the grid from this deck. Besides, we keep track of both the last boardState and current boardState so that we know the last card played by the opponent player if it's a destroy Card or malus card or bonus card. The advantages of using depth-limited depth-first search is that the memory usage is only O(bm) where b is the branching factor, m is the depth.

The disadvantage of using solely chance nodes(AndNode) and ORNodes is that the opponent movement is ignored, making our agent incompetent when playing against human player or another hostile intelligent agent. As mentioned above, we could have used the expectiminimax algorithm but we didn't. This problem is due to another limitation of our approach, huge state space induced by the large branching factor of branching out ORNode parts. This problem may be solved by storing information beforehand and using a look-up table instead so that we can save the time cost of calculating heuristic function in real-time.

There are numerous weaknesses in the program as it is not guaranteed to be an AI that can win all the time. Although the optimality is increased with the implementation of several AI Search Tree and Heuristics, the stochastic nature of the game makes the outcome unpredictable sometimes. Furthermore, although the agent is collecting as much as information as possible to calculate its moves intelligently, the program still misses most information about the opponent. If the opponent's move inference was implemented, it would highly increase the win rate.

5. Other Approaches

In the beginning phase of our implementation of the AI, we chose the Minimax tree's Alpha-Beta Pruning strategy to observe. Besides the known advantages of using this method (e.g. seek optimality at each state, compute utilities at terminal state, etc.), we will briefly talk about why we couldn't proceed to the end with this AI approach. Firstly, we quickly realized that our opponent, the random AI agent, does not necessarily always choose the optimal path for itself. In fact, it could choose a not-so-smart move, along with the stochastic nature of the distributed cards from the deck. The main factor, uncertainty, was the obstacle that we attempted to solve through running multiple simulations. However, not only does it requires a huge memory space to store all of our belief states and expensive time complexity, the ultimate result may not even be completely accurate or useful. This caused us to reject this structure and began to genuinely consider a structure that acknowledges randomness of the environment while keeping track of our own deterministic actions.

6. Possible Enhancements

One of the approaches that we later realized could be useful is the Bayesian Network [2] that we learned in class. The use of probability theory in the prediction of the opponent's moves is a great method to sharpen the agent's precision of assumption and accuracy of the decision through mathematical calculations. For instance, the conditional probability of certain cards given that the opponent previously picked certain cards can greatly relate to inferring the pattern of opponent's moves, so that the agent can better counteract against each case. Specifically, we can use the past moves of the opponent as the observed sample, and then use parameter estimation and finally perform structure learning. To compute the likelihood of certain moves, we need to collect sufficient statistics for the probability model, which means that perhaps the more rounds the game passes by, the agent may be more accurate in its prediction. Nevertheless, this strategy highly depends on the randomness of the opponent AI. If the opponent is absolutely random in its moves without any form of strategy, then inferring its moves may cause a challenge to our agent and consumes its turn's time.

If the above probabilities model can be implemented correctly and there is indeed a pattern observed in the opponent's moves, then in the later game (e.g. after 100 rounds…), we could combine Bayesian network inference with Alpha-Beta pruning[3] strategy. This is a very popular strategy for games that involve usually two players and uses the Minimax Tree[3] to simulate the game and search for an optimized path. By searching for the best move to play at each stage, it clearly enhances the optimality of our agent in the game.

Conclusion

Overall, this project led us to reflect on the subjects we learned in class and apply them to Artificial Intelligence. Such an application of concrete knowledge into practical solutions enabled us to discover the meaningful relevance of machine intelligence with the simulation of human activities.

References

1. (Class Note) Lecture 7 – Uncertainty, Non-Deterministic actions: AND-OR Search, page 22.
2. (Class Note) Lecture 14, 15, 16 – Bayesian Network, & Learning Bayesian Network, page 6, 7.
3. (Class Note) Lecture 8 – Games, page 20