Comp 424 - Saboteur Game AI Project Report

Zhiying Tan
260710889
zhiying.tan@mail.mcgill.ca

Yimeng Hu
260795862
yimeng.hu@mail.mcgill.ca

Abstract

This report details the implementation of an AI approached agent for the ***Saboteur*** board game. Various algorithms was explored during the course of the project including the local beam search. However due to the time and computational memory constraints, it was chosen to implement the heuristic minimax search [1] for the project. The performance of the agent was further evaluated against a *RandomPLayer agent*, an agent that chooses its move in a random way. It was examined that the AI agent was able to win on an average of 50% of the time and achieved a best performance of winning 18 times over the 30 games played.

1 Introduction

1.1 Purpose of the Project

The goal of this project is to develop an AI agent for the card game Saboteur. It is a considered as a gold mining game between two or more players each assigned to a role of either *miner* or *Saboteur*. For the sake of simplicity, the game has been modified to include only two players, each plays consecutively aiming to find a path from the *entrance* to the hidden *nugget* out of three hidden objects[3].

1.2 Algorithm Motivation

By the nature of the game, the performance of the agent highly depends on a decision making or a strategy planning algorithm. Therefore our intuition is to use a minimax search[2]. It assumes that there are two players, Max and Min, and assigns a value to every node in a game tree as follows. Terminal nodes are assigned static values that represent the desirability of the position from Max's point of view. Non-terminal nodes can be given the minimax value recursively.

Hence our approach is that at each turn, the agent uses a search tree to simulate the game by generating all possible moves that the opponent could play. The leaves of the tree are then given a value based on a heuristic function that attempts to approximate how well the moves are winning against the opponent player. Details of the algorithm will be discussed in Technical Approach & Theoretical Basis.

1.3 Program Explanation

In order to simulate the game during the search, a ***PlayerBoardState*** object was created independent of the ***Saboteur-BoardState*** object to ensure the simulation would not effect the current board state. The complete search algorithm and necessary tools are stored in the *MyTools* class. Furthermore, since there is a time limit constraining the execution, the search algorithm is provided with the turn start time and halts the execution when the time limit is reached. The search then returns an optimal move among the checked legal moves.

2 Technical Approach & Theoretical Basis

2.1 General Explanation of Algorithm

The main algorithm developed for the game uses a minimax cut-off search with a designed heuristic function. The algorithm expands a search tree of depth 4. At the beginning of the agent's turn, all its possible legal moves is fed in the search. For each iteration, the search simulates two subsequent turns by applying legal moves to a board state and generate children board states at a given node. Each leaf-nodes is then assigned a static value evaluated by a heuristic function. Furthermore, since there is a time limit constraining the execution, the heuristic minimax search is provided with the turn start time and cutoff the execution when the limit is reached. The next few sections will specify in detail how each part of the algorithm is designed.

Due to the complexity of the project, several special cases are handled separately apart from the minimax search. First consider the following, at the start of the agent's turn, if the agent is *blocked* from making a move, the algorithm will return right away an optimal move out of *Map, Destroy or Bonus* three type of cards if possible. Second, if the opponent is close to a win, in other words, if the opponent could find the hidden *nugget* in one move at its next turn, the algorithm will return a *Malus* card or make an appropriate *destroy* move if possible(given agent has a *malus or destroy* card in hand) to prevent the opponent from winning. Beyond these cases, Map cards will be applied at once unless the position of the hidden *nugget* is known or speculated by the agent .

2.2 Minimax search

As introduced in the previous sections, the minimax algorithm uses a search tree of depth four. This allows for a strategy of picking a move based on the simulations of what the opponent might do on the subsequent turns. It was chosen to give each leaf node a static value based on a heuristic function which evaluates how close it is to finding the hidden *nugget* if a given move is processed. Hence the agent will act as the Min player in the minimax search. In other words, at the root of the tree, the best move is picked and returned based on the minimum value among its children board states. Recursively, the value of a node will be determined in the same way by applying minimax search and with each level of the tree alternating minimizing and maximizing.

2.3 Heuristic Function

It is mentioned above that the leaves of the minimax search tree are given a value based on a heuristic function that attempts to approximate how close it is to finding the hidden *nugget* if a given move is processed. As discussed in General Explanation of Algorithm, the *Map, Malus, Drop and Bonus* type of move are being taken care of as special cases apart from the minimax search. Hence the search only returns the optimal move among the *tile* and the *destroy* cards in the agent's *hand*. It will be discussed separately in detail how their heuristic values are determined. One essential feature in the heuristic function is the distance between two tiles or between the tiles and the hidden objects on the board. It is approximated with their Cartesian coordinates[1]. Before detailing the heuristic function into a deeper level, it is worth mentioning that if a given move compose a valid path from the *entrance* to the hidden *nugget* together with the other tiles on the boards after it is processed, then the heuristic functions immediately assigns it a value of 0.

2.3.1 Destroy Card

First consider the moves that *destroys*. Those moves consist of coordinates of the tile on the board which can be destroyed. For each of such moves, an initial heuristic value will be given base on the distance from position of the tile it would destroy to the hidden *nugget*. Two features are checked for the tile it would destroy. It is first examined if the tile is part of a valid path, then it is checked if the tile is a regular *tunnel* (i.e not a broken *tunnel*). If it is, then the heuristic value of the given move would be given a weight of more than 300 since destroying such tile would lower the agent's chance of wining.

Otherwise, the move will be assigned a weight between 0 and 1 to its heuristic value, since except the situations described previously, it is only left with the states where the move is destroying a tile of broken *tunnel* that belongs to a path [2].

[1]Take the two objects with coordiantes(x_i, y_i) and the distance between the two will be calculated as $(x_1 - x_2)^2 + (y_1 - y_2)^2$

[2]Such path is consist of a set of tiles but cannot be counted as a valid path (i.e not a set of ones.)

2.3.2 Tile Card

It is left to check for the moves that are of type *tile*. It is consist of an index which shows the type of *tunnel* it represents and the position of the board where it can be placed. For each of such moves, its heuristic value will be composed of two weighted value: *minDistance* and *moveDistance*. The first value represent the distance from the closest the path to the hidden *nugget* after the given move is processed. The second value serves as the distance from the position the tile would be placed on the board to the hidden *nugget*. Then depending on the simulated board state, these two values will be adjusted accordingly as follow.

First consider the *minDistance*, the main idea of the adjustment is to check how well the card is leading the agent to a win. Given the nature of the game, there are tiles that are consist of a broken *tunnel*[3] . If such tile were to placed in the board, it is less likely for the agent to construct a valid path from the entrance to the hidden *nugget*. In this case, its value of *minDistance* will be given a weight of 10 to 15 depending on the tile. Otherwise, if the given tile creates a valid path together with the other tiles on the board, then its value of *minDistance* will be assigned a weight of 0.01.

Similarly for the *moveDistance*, it is first considered if a given move is consist of a tile with a broken *tunnel*. If it the case, then its value of *moveDistance* will be given a weight of 20 or 4000 depending on if it is able to compose a valid path together with the other tiles on the board. Then consider the case if the given move is set on a position below the hidden *nugget* when there are other more appropriate moves on the rest of the board. If that is the case, the *moveDistance* of the given move will be assigned a weight of 1.5 if the move is consist of the tile with a vertical *tunnel*. Finally, if a given move has a *minDistance* of value less than 2, and it is consist of the tile with a horizontal *tunnel*, its value of *moveDistance* will be weighted 1.5 times higher.

These two values are then combined with weights calculated respectively as explained above in order to reduce the bias in the outcome of the heuristic function.

2.4 Time Constraint

The time constraint is handled by halting the execution if the restriction is violated. It is done within the minimax search algorithm by checking the time at the beginning of each node minimaxization and throwing an exception if the time limit is reached. The exception is thrown all the way up to the function call from where it is caught and returns the optimal move from the last completed minimax search.

3 Discussion and Conclusion

The performance of the implemented agent was evaluated opposed to a *RandomPlayer* agent, an agent that chooses its move in a random fashion. By the nature of the game, there is a fixed number of cards of different types and hence finite number of turns. This implies that it is involved a *luck* factor in winning the game. Therefore it is expected that our agent is not guaranteed a win on every game played. However, despite what has been said, the AI agent was able to gain a win on an average of 50% of the time and achieves 18 wins over 30 games played on its best performance.

3.1 Other Approaches

During the course of the projects, Other approaches were explored. However due to the time and computational memory constraints, they were discarded. This section will discuss in detail one of the inspected algorithms, local beam search algorithm, as well as the reasons why they were abandoned.

3.1.1 Local Beam Search

The local beam search is a heuristic search algorithm that explores a graph by expanding the most promising node in a limited set. It uses breadth-first search to build its search tree. For this project, the classical local beam search has been modified to cooperate with the minimax algorithm.

The algorithm conducts an initial minimax search among all possible legal mvoes with a cut-off depth level of 4. It returns back a set of k best optimal moves selected based on the heuristic function. The search halts if any one of the successors reached a goal state, otherwise it proceed with the next minimax search among the k moves and returns back $\frac{k}{2}$ optimal choices. The search goes on repeatedly until k = 1 and it will be the most optimal move.

[3]For example, tile:2 ,3, 4, 11, 12, 13, 14, 15 or their flipped version

Clearly its complexity is much greater than the minimax search presented in Technical Approach & Theoretical Basis and it eventually violates the time constraints with little improvement on the overall performance. Hence it was not chosen to be the best algorithm for the agent in the contest.

3.2 Advantage and Weakness

One obvious weakness of the algorithm is that it expand no further than depth three due to the time constraint. The depth limited minimax algorithm results in a more biased outcome. Furthermore, given the specialty of the cards, the algorithm only considers the *tile* and *destroy* type of cards. The cards of *Malus*, *Bonus*, *Drop* or *Map* type are dealt separately outside of the search algorithm. As a result, the minimax search algorithm may lack a certain degree of complexity. However the strong side of the algorithm is reflected by its time complexity. The agent was able to make a decision at each turn within 0.1 second and managed to win a game on an average of 50% of the time. The best performance it achieved was winning 18 times over 30 games played against an *RandomPlayer* agent.

3.3 Future Improvements

There are several improvements could be done regarding the two weaknesses discussed in the previous section. First, It is possible to expand deeper in the search tree by simulating more subsequent turns. Moreover, it is likely that the algorithm could include the cards of type *Malus, Bonus, Drop or Map* into consideration for future improvements. This way the complexity of the algorithm could be raised and reduce the bias in the outcome. Furthermore, taking the time and computational memory constraints in mind, an alpha-beta pruning could be added to reduce the size of the tree that is expanding.

In addition to what is set forth, inspired by AlphaGo[4], the game agent built on the neural networks and Monte Carto tree search, developed by Google DeepMind, it is worth trying to combine the minimax search algorithm with machine learning and extensive training with large scale of data. It is possible to let the agent play against the *RandomPlayer* agent repeatedly in order for the model to learn and to improve. The goal will be to obtain a well trained model optimizing the best move of the agent given a board state.

References

[1] Michael Buro. Improving heuristic mini-max search by supervised learning. *Artificial Intelligence*, 134(1-2):85–99, 2002.

[2] Murray S Campbell and T. Anthony Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20(4):347–367, 1983.

[3] Adam Trischler; Project TA: Pierre Orhan Course Instructor: Jackie Cheung. Comp-424–saboteur-rules. winter 2020.

[4] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.