Developing a Saboteur Game Playing Agent

In playing Saboteur and testing our agent in countless number of games, we have developed an appreciation for the nuances of creating a game-playing agent. We have realized the challenge of developing an agent that accounts for the very many different states of a game and how to reason at each point of the game. We acknowledge that decision making that is intuitive for humans is not so for a computer. With that in mind, we have created a game playing agent for Saboteur to the best of our abilities.

Technical Approach

Our technical approach relies heavily on a few of the methods provided to us, particularly getAllLegalMoves and getHiddenBoard in SaboteurBoardState. We use the information available on the current state of the board, because we prefer to make our moves based on current available information rather than having to predict what the board state would be like after moves by our agent and the non-optimal opponent.

We first start by running breadth first search on the tiles with the entrance as the root of the search tree. Our goal is to find the k-closest (in our case k=8) already placed tiles in terms of Manhattan distance to our current goal tile (one of the hidden objectives).

We then proceed to create a HashMap with the keys as coordinates (e.g. "[6, 5]") and values as legal moves at the given coordinate. Our algorithm iterates over the k-closest placed tiles and retrieves the legal moves that may be placed on its empty neighboring tiles from the HashMap. For example, if the closest placed tile to our goal tile is "[6, 5]", we would explore its neighboring tiles ("[5, 5]", "[6, 4]", "[6, 6]", and "[7, 5]") and the legal moves that may be placed there to construct a path closer to our goal. If the legal move is a SaboteurTile that is not beneficial (tile has a wall in the center of its path, such as tile 1, 2, etc.) in the long term, it is disregarded since it would build a disconnected path. In addition, if the legal move is a SaboteurTile in which the coordinate of its connection in the IntBoard is at a lesser distance to the goal tile than any other edge of the tile, then constructing that tile would effectively increase the Manhattan distance to the goal tile and can thus be disregarded. Effectively, we have pruned out legal moves that are not considered to be good long-term investments towards the goal tile. These remaining pruned legal moves are then sorted based on the Manhattan distance of the move position to the goal tile, and the priority of the tile (we came up with a custom priority of the cards).

As mentioned above, we have come up with a custom priority of the tiles, with the best tiles having the lowest numerical value as priority. For example, tiles with index 0 and 8 have low numerical values which means they have high priorities while tiles 1, 2, and 15 have low priorities.

At the start of every move we check to see if we can update our target tile (the tile we are building towards). At the beginning of the game, the target tile is the middle hidden objective, however as the game progresses and we discover the cards behind the hidden objectives, we must update the goal tile we are building towards. So if we have played a map card we have discovered that the middle hidden objective is just tile number 8, then we have to update our current goal tile to either the left or the right hidden objective depending on the other information in the game.

We also have some logic for special cards such as map, destroy, bonus, and malus cards. This code checks for these special cards prior to searching for a tile to place. In other words, playing a special card has more priority than playing a tile move. The rule we have for malus cards is to play one if we have one in our deck. After, we check for map cards; if we have a map card in our hand and the turn number in the game is less than the Manhattan distance from the closest tile to the goal tile, then we play the map card to adjust our search. The reason for checking for the turn number is to make sure we are not playing the map card when it would be crucial to play a tile to win towards the end of the game (i.e. the turn number is higher). If there are no malus or map cards, we check for destroy cards, and in the case we have a destroy card, we run a similar search algorithm that finds a tile that is deemed worthy to destroy, such as the ones with low priority based on the custom priority we created. If a malus card has been played against our agent, then we attempt to play a bonus card if there exists one in our hand, otherwise we check for map and destroy cards. In the case no special cards can be placed, we drop a card that is thought not to be useful for future moves such as the bad tiles.

To emphasize on the motivations for our technical approach, we wanted to implement an agent that would play as closely to a human player, if not better. We attempted to translate the thought process a human would go through to play a move into an intuitive algorithm. As we played Saboteur ourselves, we realized that we were in fact searching for the tile closest to the goal and building around that tile if possible. Otherwise, we would search for the second closest tile, and so on and so forth. This part of the human game-playing process was reasonable to simulate algorithmically. Using knowledge from previous classes like COMP-250 and general algorithmic knowledge, we decided that implementing this algorithm would be feasible. However, the destroying cards optimally and restoring damage caused by Random Player aspect of the human game-playing process were more difficult to imitate.

Attempted Approaches

As a first approach, we tried looping over the list of legal moves and choose the best one based off its Manhattan distance from the position of the move to the target and the tile that was being placed. There was not a huge improvement in the win rate as the criteria for choosing moves was not specific enough. Another approach that we thought of was the Monte Carlo tree search, but based off the methods provided we found it hard for us to implement it as we would have to process moves to go to future board states without it affecting the state of the board on the server. In the end, we chose a greedy approach,

because we were concerned that using a game search tree algorithms such as MCTS and Minimax would have a very large time complexity as the search tree would have a very large branching factor for the possible moves in each turn. This was especially important when given two seconds per move. Finally, we ended up building upon the greedy approach to make what we think is the best move given the information provided.

Pros & Cons of Chosen Approach

We believe that our approach is intuitive and easy to reason about. We utilized the provided getAllLegalMoves method in combination with breadth-first search to explore moves neighboring the tiles that are closest to the goal tile. This approach simply tries to minimize the Manhattan distance of the chosen move and maximize the utility of the tile placed. Our algorithm also has some small edge cases that were included due to logic, such as playing a malus card immediately, dropping the least useful card in a state of malus, etc. We reasoned about Saboteur by playing various games using the GUI and developed the algorithm in accordance with our own human-logic. Our approach would fall under the category of greedy algorithms since we effectively try to maximize our distance from the entrance and minimize our distance to the goal tile simultaneously. In addition, our algorithm has as relatively low time complexity footprint. We run BFS from the entrance of Saboteur to find all the tiles we can reach and keep track of the k-closest coordinates to the goal tile. Iterating over the k-closest coordinates, we explore playing cards in neighboring positions that are viable winning paths and keep these in a list of potential moves. Finally, we sort the list of potential moves by the priority of the tile and the Manhattan distance of the move.

Some downfalls of our algorithm are that some situations are not accounted for, it is difficult to recover from disconnected paths built by Random Player, and we hard coded some specific hand-oriented scenarios. We did not expect how difficult it would be to go up against a Random Player, as it sometimes destroys path at the most inopportune moments and builds tiles that create incomplete paths. Our algorithm does not consider paths that have been corrupted by the Random Player and attempt to fix it, but rather may proceed to build paths towards the goal in other various ways. Given more time, we believe that we could have included some mitigations for these purely random moves. Our agent has some hard-coded situational logic, such as playing a malus card regardless of our hand and the state of the board, playing bonus and map cards as a priority during a state of malus, etc.

Future Improvements

In the future, an improvement we would make to the current approach is an algorithm that would build around or fix disconnected paths to the golden nugget. We want to define what we mean by “disconnected” as a path that doesn’t lead to the golden nugget because either it hasn’t reached the golden nugget on the board or the path to the golden nugget has a tile path that has a wall in the center (we call those *bad* tiles). Often, the reason for which a game ends in a draw is that the path to the nugget has a bad tile. This made the game very

hard to win, especially if there were many points where the path was disconnected. An approach to tackle this would be to build on a different path that is not disconnected.

Another improvement we could make is to incorporate logic regarding the cards played throughout the game. We would keep track of the cards we have played, and the cards on the board to deduce which cards the opponent has. We think it would be especially useful against an optimal agent when trying to have the final move to reach the golden nugget. We want to make sure we have the final move when reaching the golden nugget, and we wouldn't want to gift the path to the opponent, so we would try to sabotage them, by either playing a malus, destroying a tile, or playing a map card.

Other ideas include keeping a history of our moves to detect if our path has been broken and destroying the opponent's path to the nugget if we are malused and they are on the cusp of winning.

An improvement that was on the back of our minds when implementing our solution was to attempt implementing the Monte Carlo tree search, because it would be able to play against an optimal agent and a random agent. We learned early on that playing against a random agent is harder than expected especially in Saboteur where both players are attempting the same goal. In fact, playing against a random agent could be more difficult in some instances than playing against an optimal player as they are completely indifferent to making it to the end goal, making a mess of the board in the process.

Conclusion

Overall, we have learned a lot from this project, as we have explored creating a game playing agent in different ways. We have collaborated between ourselves and peers in the class to hear about their interesting different approaches, and we have had fun along the way.