COMP 424 Final Project Report

Suleman Malik 260652774 - Parsa Yadollahi 260869949

April 16th 2020

1 Explanation & Motivation of Approach

Our approach with the Saboteur game-playing AI was to keep our decision-making model as simple as possible. This allowed us to keep the code structured, extensible and easy to improve/debug. The program uses various greedy heuristics in order to pick the best move given the board state at each turn. Our program closely resembles A* search however we chose to ignore the cost up to the current tile in our heuristic. We felt the accumulated cost would not help reduce the problem state as the goal states were static and unchanging. Our central heuristic was using the Manhattan distance between a legal move and the goal tile as an indicator for the "goodness" of a move. Additionally, we employed two other heuristics: a heuristic value of each card and a heuristic value of the position on the board on which the tile could be played. This added a layer of robustness to the algorithm which would not be there had we only used the Manhattan distance. At each step, the algorithm would filter out cards based on their heuristic value, further filter the moves leading to the shortest Manhattan distance, and then pick the best position on the board to play depending on the proximity to the goal state.

Our motivation for using this approach was that it implemented a simple yet powerful decision making method. It would be able to pick a move quickly enough and without visiting/simulating a large number of game states. At first, our instinct drove us to use Monte-Carlo tree search or the Minimax algorithm as we assumed they were best suited for game-playing AIs. However, after implementing rudimentary versions of both MCTS and Minimax, we noticed that the branching factor grew very large very quickly. The large branching factor coming from the millions of possible game states highly dampened the speed of our program. Additionally, the overhead of simulating game states added to the complication of using these algorithms, This drove us to stick to a simple decision-making algorithm based on heuristics that we chose and later tuned to optimization.

2 Theoretical Basis of Approach

As outlined above, we decided to base our AI on various heuristics that hopefully gave the best choice of tile placement at each move. At a high level, the three heuristics we considered were: the Manhattan distance to the goal state, the optimality of each card and the best position associated with each legal move.

2.1 Initial move

Our main function *optimalMove* was designed to iterate over the hand at each turn and play the best card first. We decided to play either a Malus or Map card first because it proved more advantageous in our many test runs. Playing the Malus first allowed for our AI to play the best move, uninterrupted by the random player until it played a Bonus card. If no Maluses were present, we played a Map card which narrowed down the goal and problem space. The function was purposefully set to build a path towards $[12, 5]$ if no Maps were present because at the least it brought us closer to the other goal tiles. If a Map was played, it was always played on $[12, 3]$ first. This allowed us to rule out an edge and avoid split paths. If the nugget was not at $[12, 3]$, we could build a more localized route towards $[12, 5]$ and $[12, 7]$. If neither a Map or Malus card were present, we passed control to a second function *decisionHeuristic* to choose and return the optimal move.

2.2 h_1: Optimal card

The first step of filtering the hand involved first playing the best cards. We defined the best cards as the tiles that had straight paths down. The next best cards were the ones that made turns but still did not include dead-ends. Finally, the worst card we defined as the ones with dead-ends and would break the path to our goal state. We immediately filtered these cards out or dropped them if the hand contained only bad cards in hopes of receiving a better card to play.

Figure 1: Optimal cards according to h_1

Figure 2: Sub-optimal cards according to h_1

2.3 h_2: Manhattan distance

After all bad cards that lead to dead-end paths were filtered, the program would play the best card that reduced the Manhattan distance by the most. If the goal tile had already been uncovered, it would shorten the distance between the entrance tile and the goal tile. Otherwise, it would build by default to $[12, 5]$ until a Map was played and the goal was uncovered. Another slight optimization we included was to automatically orient itself to the third tile if the other two were discovered to not contain the gold

nugget. This allowed us to play one less Map card and use that move to potentially move us closer to the goal based on the Manhattan distance.

2.4 h_3: Optimal position

Finally, we ranked each possible position on the board in order to play tiles closer to the goal rather than anywhere on the board. From the list of possible positions, we assigned every spot on the board heuristic value and set optimal cards to be played there first. This allowed the path to easily choose between two moves that might have reduced the Manhattan distance the same amount but would move the path direction in an undesirable direction. Additionally, near the goals, we allowed certain cards to be played first that would keep the path as open as possible and not barricade any entrance to the goal tile. With regard to Destroy cards or dead-end cards played by the random player, we allowed the path to continue being built until a goal tile was reached. After which, we attempted to destroy the bad tile or replace the hole in the path with an optimal card. This ensured that our AI had the last move to complete the path to the goal state instead of the random player but relied heavily on the chance of getting a Destroy card and then an optimal card.

3 Advantages & Disadvantages

3.1 Advantages

The most notable advantage our approach used was the layering of various heuristics. Before each move was chosen, the best card was selected to be played on the best board position leading to the shortest Manhattan distance from the goal state. This way we could ensure to almost always play the optimal move at each iteration. Going through multiple layers of filtering moves and limiting the playing field, we would often end up removing all possible cards from our hand, which would lead us to dropping a card instead of playing a card. This would sometimes end up being the turning point for the game since it would give it advantage and allow it to take the lead. Our agent would also take into account many times where the opponent would try to sabotage our path to the goal state. Our agent would take into consideration if, for example, the opponent would play a Malus card or would block our constructing path towards the goal tile and would take immediate action to overcome it if the right cards were present in it's hand. Though if the opponent would destroy a tile somewhere along the path towards the goal causing the path to be disconnected, our agent would not be able to fix this path and would assume that it is still connected to the entrance tile. This lead to many tied games as our agent would not be able to fix this issue and keep circling around the goal tile or dropping cards until eventually there were no more cards in it's hand.

3.2 Disadvantages

Our function that would calculate the Manhattan distance works well with the tiles themselves and would provide accurate detail on which card to filter out but would lack accuracy when close to the goal tile. The issue is that function would look at the tile itself without regards to the path that the tile leads to. Implying that a if

two cards had the same Manhattan distance to the goal (with regards to the tile), the decision of which card to play was left to the *rankCards* function, which would sometimes pick the wrong card. An advantage that can be shown yet not very critical to the games outcome is the time efficiency to our outputs. Since we do not need to search though a big branching factor of a tree but simply all the possible moves that could be played at that moment. We could quickly return an accurate result.

4 Other Approaches

Our goal for the algorithm was to find a strategy that would maximize the utility. We first noted that the game was deterministic and imperfect. We looked at many search algorithms and decided to proceed with the greedy heuristics outlined above. We looked into algorithms such as Minimax (with and without Alpha-Beta Pruning), Monte-Carlo tree search, and A* search. After discussing the pros and cons of each approach as well as implementing rudimentary versions of each, we decided to proceed with the simplest option; greedy heuristics.

4.1 Minimax

We initially considered Minimax because it seemed like a solid choice for alternating turn games such as Saboteur. Given many other games have had successful AI agents built using Minimax, we began designing and implementing it and envisioned to further optimize it using $\alpha - \beta$ pruning. WE quickly learned how unsustainable the approach became not only because of its large branching factor but because we needed to represent the game as a tree and simulate many thousand possible game states. Additionally, Minimax assumed an optimal opponent therefore it was not well suited to beat the random player. Using Minimax differed steeply from our approach mainly because there was no need for our agent to simulate possible game states. It simply chose the best move given its state.

4.2 Monte-Carlo Tree Search

Next, we attempted to implement MCTS as another possible solution to this problem. Just like Minimax, MCTS presented us with the issues of changing the game representation, the large branching factor and the overhead that comes with simulating games. Ideally, MCTS would have outperformed all other agents but it proved to be both computationally intensive and memory expensive. Again, MCTS starkly compared to our chosen approach as they both tackled the core issue very differently.

4.3 A* Search

Finally A* search, the approach most similar to ours was the last alternative we considered. Classic A* search would require a graph representation but that was not necessary in this scenario given the algorithm;s nature of basing decisions on heuristic values. The one heuristic integral to A* that we did not consider was the accumulated cost up to a certain state. We found this data not useful because of the nature of the game. Other than that, our approach was similar to the mechanics of A*. We assigned

a heuristic value for each playable move and chose the value that greedily optimized our utility (closeness to the goal state).

5 Improvements

In hindsight, there were many optimizations that we would have included to make our agent more robust. We consciously decided not to simulate various game states, however after reducing the state space through iterative filtering using our heuristics, it would have been advantageous to simulate a number of possible scenarios. Because the agent would be confined to making moves in a certain area of the board, simulating a few thousand games would not significantly decrease the processing time of our program and would prove very advantageous. Perhaps running Minimax or MCTS once the problem state was greatly reduced would add another layer of intelligence to our agent.

Another improvement we had began implementing but did not have time to finish was a heuristic to keep track of breaks in the optimal path. This would help rebuilding paths that were impeded by dead-end cards placed there by the random player. Because of time constraints, we simply let the agent complete the path as if it was fully connected until it reached a goal state. Once the nugget tile had been connected, we tried to destroy the dead-end card and replace it with a suitable tile. This could have been approached more intelligently, had we stored the continuous path in a list and played a destroy card and rebuilt the path at the first opportunity. This would remove the reliance on the chance of getting a Destroy card followed by an optimal card to properly rebuild the path.