COMP424 - Final Project Report

Jiuyue Cai (ID:260770961) & Nico Xia (ID: 260770119)

The purpose of this project is to build an artificial agent that could play Saboteur Game with as higher wining rate as possible. Initially, there are one entrance and 3 hidden objectives spot on the board. A deck of card will randomly assign card to each of the player's hand. Each player has at most 7 cards on its hand. Player needs to determine how to find the objectives with real nugget, how to construct a road from entrance to their goal, and how to hinder opponent's movement or to solve the problem caused by the opponent. According to these rules, we develop an agent with greedy algorithm and target to conquer the random player. We achieve 50% winning rate.

EXPLANATION

The algorithm logic is divided into five parts: Set Goal - Categorize Legal Moves - Test State - Rank Card - Rank Moves. Below are the specific explanations.

Set Goal

Giving a clear position for our AI agent to reach is the most essential step in our logic. On the board, we have three different hidden objectives ({12,3},{12,5},{12,7}) but only one of them is nugget. Hence, at the beginning of every turn we need to look through three hidden positions. We created a boolean nuggetKnown to check if nugget is found and a boolean[] positionRevealed array to store revealing information. We use int[] goal to store our current goal for this turn. Firstly, we set our goal to be in the middle: {12,5}. We only check {12,3} and then {12,5} (Reasons due to *Map* card mechanism, will explain later). We update goal[] to the position ID that is "nugget" and if they are all "hidden", the goal sets to {12,7}. If there is none of the situations above, the goal will be keeping in the middle.

Categorize Legal Moves

Secondly, we create an ArrayList<SaboteurMove> legalMoves to store all legal moves. Then we create seven extra ArrayList: bonusCard, malusCard, tileCardGo, tileCardEnd, destroyCard, mapCard, dropCard for categorization. (However, we did not implement *Destroy* for our AI agent in this game.) It is seen that we differentiate valid *Tile* cards and invalid *Tile* cards in this step. Valid *Tile* cards, 'tileCardGo', are cards with 1 on its center by checking its path, which means we can go through this card from one side to another side. On the other hand, invalid cards, 'tileCardEnd', meaning that if we play this card, we make a dead end for our existing road. To seek a high winning rate, we only play valid *Tile* cards all the time.

Test State

The most essential state for us to make our decision depends on the *Malus* card. When *Malus* is used on a player, this player is blocked to play *Tile* cards until giving a *Bonus* card. Hence, we need to test whether we are "locked" or "unlocked". boardState.getNbMalus() is a function for us to query our locking status and even

opponent's status. We need to input the corresponding player's ID into this function. In case we are not sure about our ID, we used boardState.getTurnPlayer() function. 0 and 1 are IDs for us and the opponent determined by the launch orders. Two different card-playing mechanisms are used according to the current state we are.
If we would like to play a *Malus* card, we need to know our opponent's locking states. It's wasteful if we continue doing *Malus* to our opponent while he is already locked.

Rank Card

If we are "locked", we rank cards in the order: *Bonus, Malus, Map, Drop*.

If we have *Bonus*, we prioritize it in order to become unlocked as quickly as possible. If we cannot solve our dilemma and the opponent is not locked, we use *Malus* to stop him making any useful movement. If not applicable, we use *Map* if we have one and if nuggetKnown is false because we want to specify the correct goal as early as possible. If we have none of the cards above, we drop what we have to leave space for what we need.

If we are "unlocked", we rank cards in the order: *Malus, Map, Valid Tile, Drop*.
For the choices of the first two types of cards, the reasons are illustrated above. If not applicable, we play valid *Tile* cards to pave our way to the goal. Choosing which specific *Tile* card to play is strongly related to where we put it, so the detailed explanation is given in the next part. If none, we drop cards.

Rank Moves

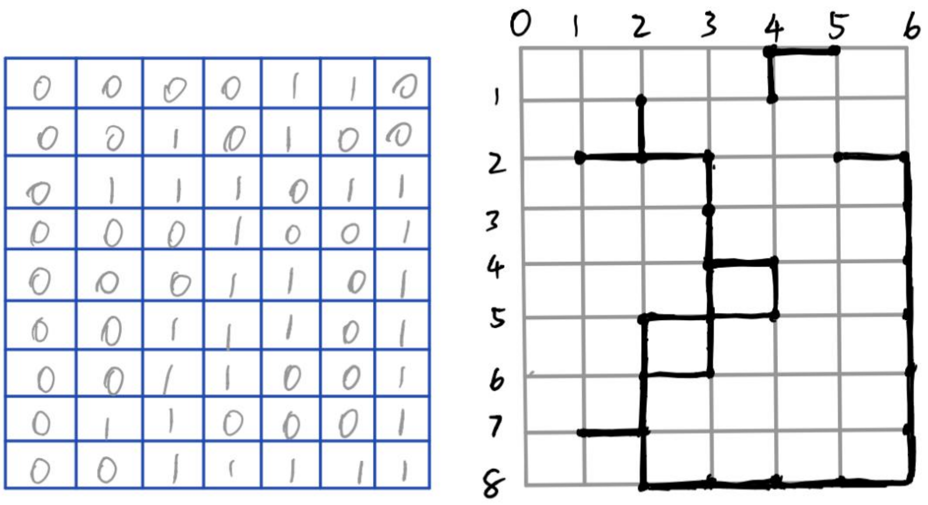
Malus and *Bonus* cards are played directly on players. Thus we only rank the following:

Map - From observation, the position that *Map* moves will take on are ordered from left to right ({12,3} to {12,7}). Firstly, we check if positionRevealed[0] == false (meaning that we haven't checked any hidden position). If so, we choose the first move from all *Map* legal moves. If not applicable, check positionRevealed[0] == true && positionRevealed[1] == false, (representing that we have checked the right and it is not a nugget, we want to check the middle one). Then we choose the second move from all *Map* legal moves. We won't use the third *map* card because we will know the nugget position during goal-setting step.

Tile - This is the essential part of our programming to make our agent get closer to the correct destination. All the decisions are made based on hiddentIntBoard, the game board which uses 1 for road, 0 for wall, -1 for empty space, and tile '8' path for all three hidden objectives regions. Only moves from tileCardGo lists are considered.
Among all legal valid *Tile* moves, we firstly check whether playing this *Tile* at this spot can connect back to the entrance. We copy the whole hiddenIntBoard and temporarily operate the testing move on this board. Taking the center of the testing move's Tile as the start point, taking the center of the entrance card as the end point, and taking the whole copy board as a graph, MyTools.checkpath() uses algorithm which is similar to breadth first search to find any path from the start point to the end.

Moves that passed the path-checking section are proceeded into this distance calculation procedure. The goal tile's center is the goal's coordinate used in this part. We calculate the Manhattan distance between all the '1' spots from the testing move to the goal coordinate. Then we put each tile's index and its corresponding distance as a pair into a priority queue. Worth to notice, each tile's index is stored into the queue multiple times, since each tile contains more than one '1' spot. We sort the elements in the priority queue based on their Manhattan distance in ascending order after we finish traversing all moves. We simply return the first one from the queue, which is the closest move to the goal.

If no element is stored in the priority queue, representing that all the moves are not connected back to the entrance or are useless. We then randomly drop a card to see if we can get something useful in the next turn.



(checkPath algorithm demonstration)

Drop - If we do not have an appropriate move to take, we drop cards in the following order: *Map, Destroy, invalid Tile, valid Tile, (Malus, Bonus)*. If we have already known the nugget position, we drop the Map card first. If none, we drop *Destroy* since we do not use it. Then we drop invalid *Tile* card. Later, we drop *Malus* and *Bonus* cards, although it is theoretically impossible to have the chance to drop them.

For implementation, we firstly loop through the boardState.getCurrentPlayerCards() to find the card which we want to drop, then we possess the index of that card in the current hand. At last, we create a new SaboteurMove(new SaboteurDrop(), index, 0, boardState.getTurnPlayer()) and return this as myMove.

At the very end (outside the big state-testing if/else loop) we let the program randomly drop one card from all legal drop moves for preventing crashes because doing an arbitrary move will possibly block or destroy our well-planned path.

ADVANTAGES AND DISADVANTAGES

Advantages: Firstly, we decide to lock our opponent as often as we can since we prioritize *Malus*. We use this strategy to limit the moves that our opponent can make. This benefits us to pave our own road more freely or to prevent opponents from putting obstacles on our path. Secondly, using Manhattan distance is more closely related to road constructing style in this game. Thirdly, instead of using a famous existing game playing algorithm, our approach is just a Greedy Algorithm. But it works well by closely following and reacting to this game's rules. It adapts the rules well by simply using if-else block as case checking. It saves time and space in most cases, instead of constructing complicated searching trees for decision making. And it is easy to debug, because the functions are organized into each independent and interference-free block. In general, it performs well to beat the random player (we achieved around 50% winning rate in 100 turns).

Disadvantages: we didn't use the *Destroy* card, which means we give up the chance to deconstruct the opponent's road. Also, we cannot conquer the obstacle easily but only choose detouring. Besides, once there is no way connected to the entrance, we definitely lose it. We didn't use invalid tile card, which means we cannot seal the opponent's construction while he approaches the destination. Sometimes we cannot fix our semi-finished path if our opponent destroys parts of it. Since we decide to lock our opponent by *Malus* as often as we can, it seems that we push him to destroy our road with a bigger chance, especially for the random player. Generally, we consider as many cases as we can think of, but we might still miss some unpredictable edge cases. Our agent doesn't make decisions for the whole game by observing and analyzing all potential factors, and it's not clever enough to beat human players or other AI agents.

OTHER APPROACHES

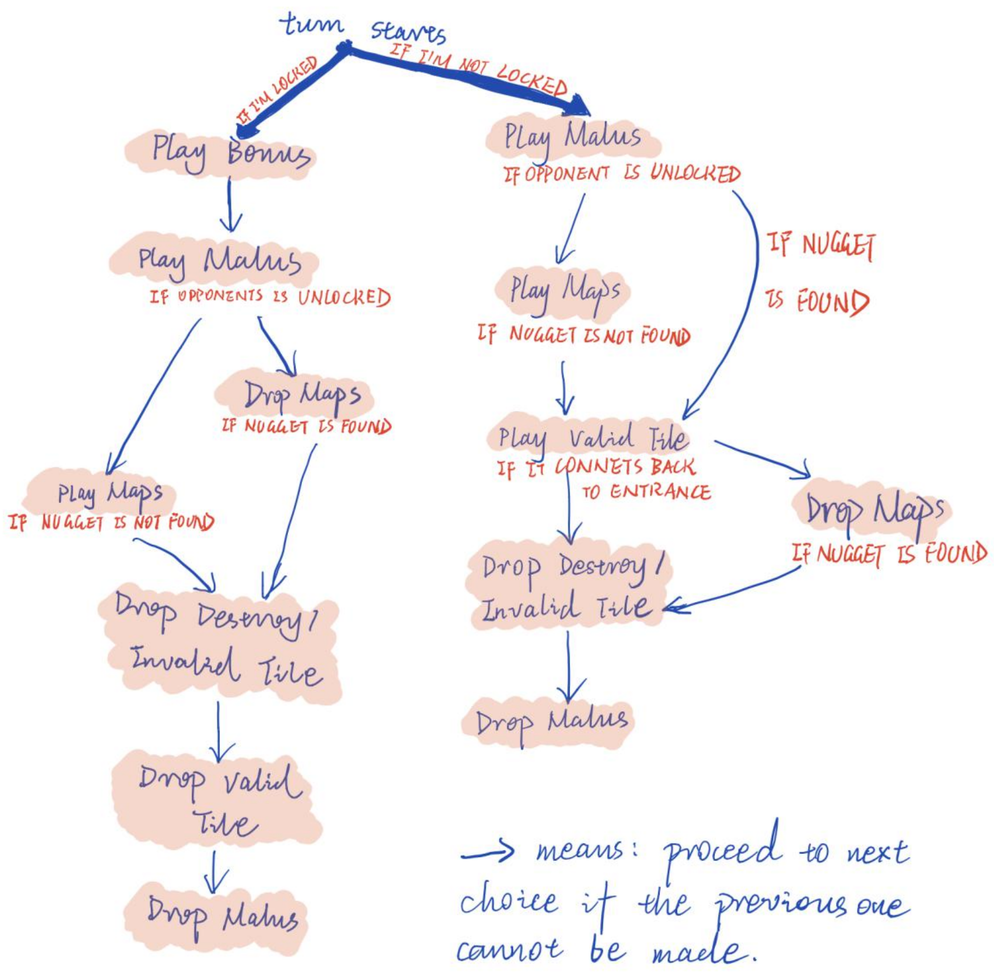
Primarily, in the *Tile* selection parts, instead of calculating each '1' spot from the testing tile to the goal position, we tried simply using the center spot of each tile. This approach saves more spaces and time, but the winning rate is not as ideal as the one we currently choose. The reason might be that this approach losts information on each tile's edges. Moreover, switching to our current approach, we originally consider testing whether those spots are connected to the open end from the result of the last turn or become the new open end for next turn. We only calculate Manhattan distance for those '1' becoming new open end. But the logic behind gets much more complicated while the board is developing, and the winning rate doesn't change whether we do this additional tracking.

Meanwhile, we also tried to add two more constraints: only tiles that contain vertical paths can be added on row 11(e.g.: Tile 0), and only tiles that contain horizontal paths can be added on row 12(e.g.: Tile 10). Vertical road helps us to eliminate the turning away cases while approaching the destination. Horizontal road helps us to easily approach the real nugget if our original destination is reached without revealing nugget. But we failed to work it out before the deadline. Although we are not sure if it is correct to add many constraints to the agent, we think it is still worthy to try. Because this approach takes future steps into account instead of just focusing on the current cases.

IMPROVEMENTS

If we still follow the big frame of Greedy Algorithm, we need to find a proper way of using *Destroy* cards. We probably should track the road which we are currently working on, so we can fix the hole if our opponent just destroys one tile of it. We can also temporarily put testing moves on board, traverse among the board to get the position of the closest ‘1’, and take advantage of this ‘1’ to our destination.

We believe that the Greedy Algorithm is not the best choice since it is not comprehensive enough. Selected moves might be most optimal for now but not potentially for the future. We can keep part of our scheme as greedy, like the part of *Bonus*, *Malus*, and *Map*, but change the *Tile* selection part into Min-Max algorithm or other search algorithm with analyzing of our opponent’s potential movement.



(Logics Flowchart)

(Notes: Italic - Card Type, Underlined - codes that appeared in our java files)