Christopher Zheng
260760794
William Tianqi Xing
260771601

COMP424 Artificial Intelligence
Final Project Report

2020/04/13

1 Introduction

In March 2016, the world was shocked that an algorithm designed by DeepMind won a Go tournament (an ancient Chinese board game) against the Korean grand-master Lee Sedol[1]. It is enlightening to appreciate the artificial intelligence of DeepMind's approach and we are inspired to construct our own intelligent agent to master a common board game. In particular, we expect to implement a set of rules by following which our agent can correctly (and preferably competitively) play the board game, Saboteur modified, with a level comparable to other entry-level agents with artificial intelligence algorithms and techniques.

The modified version of Saboteur that we play is to establish a non-interrupted path towards the final golden nugget, that is, the goal. Given a mixed deck of path cards and other cards of actions, two players take turns sequentially. In his or her turn, one plays a card from his or her hand (or throws it away) and collects a new card if the deck is not yet exhausted. The player who is the first one to construct a fully connected path (no matter how twisted it is) linking the origin and the golden nugget wins.

This report is organized as follows. The detailed explanation of the logic behind our best-so-far agent is presented in Section 2. Next, we introduce and analyze a set of candidate agents aside of the best and conduct a comparative analysis in Section 3. Finally, a brief discussion on future work takes place in Section 4.

2 Mechanism of Our Program

In this section, we introduce the underlying mechanism of our Saboteur player which strictly follows the preset rules that we constructed based on our domain knowledge and observations. Specifically, we elaborate on our best approach in Section 2.1 and the intuition behind it in Section 2.2. Then, a thorough analysis of our agent's merits and drawbacks is presented in Section 2.3.

2.1 Technical Approach

Our method obeys various Saboteur-specific rules and heavily depends on the current board state and cards in check. Hence, sometimes, certain actions are given higher priorities than others are.

To start with, in order to ensure that our agent is currently not constrained in any form, we give *Bonus* the top priority if our agent was cursed by one or more *Malus* cards. If we failed to do so and the opponent played a *Malus*, we consider playing a *Map* card (if we have any) or dropping a relatively useless card. The usefulness of a card is determined by its type, e.g., *Tile*, and (if applicable) sub-type, e.g., *Tile 8*. If we were not impacted by a *Malus* card, we consider playing a valid *Tile* card (if we have any). Determining what type of *Tile* cards to play is complicated. Since we aim to construct a valid tunnel connecting the nugget and the origin, we never play dead-end *Tiles* which cannot make through neither two directions and which we define as *Obstacles* in

Section 3.2. After eliminating the *obstacles*, we use the function can_reach_origin to select *Tile* moves that can trace back to the entrance through the tunnels so that our moves are always valid and, more importantly, meaningful. To a large extent, this avoids wasting time on paths comprising dead-ends or blind alleys. Then, we play a valid *Tile* according to its capacity to connect other *Tiles* (first vertically, then horizontally). For instance, a *Tile 8*, a crossroad, is considered better than a *Tile 0* which contains a vertical road only.

Specifically, the function can_reach_origin checks whether a given *Tile* move is connected with the origin by a direct path. Starting from a tunnel element (denoted by 1 on board) in this *Tile*, the function repeatedly visits one of this *Tile*'s neighbors which is also a tunnel element, and recursively visits further neighboring *Tiles* who are tunnel elements. Once the recursive algorithm reaches the the entrance, it backtracks the path and validates the last *Tile*. We also added a counter that documents the time of returns in this path, which indicates the distance from the current element to the origin.

More significantly, we propose a slightly different strategy when the agent is in the "dangerous zone" on the board, i.e., Line 9 to Line 12. (Line 12 is where the golden nugget and two hidden objectives reside.) In this critical zone, our agent is programmed to take the initiative to the competition. In particular, if we had a *Malus* in hand, the agent utilizes it to prevent the opponent from acting for at least one round. Based on our observations, this strategy appeared to be very effective when our agent combats the random player who is highly unlikely to use a Bonus card even if it has one because of probability. Besides, when the agent reaches Line 11 and 12, we enforce distinct policies of *Tile* selection. For instance, at Line 11, the agent tends to choose *Tiles* that have vertical paths immediately above the nugget or two hidden objectives. On the contrary, at Line 12, the agent is more possible to select *Tiles* with horizontal paths connecting a hidden objective and the nugget or two hidden objectives. Remarkably, if our agent accidentally reached Line 13 due to opponent's move, we try to force our agent back to track via returning to previous lines. Of course as a general precaution, if we did not have any valid *Tiles*, we drop the least useful card; if we ran out of cards, we return a random move to avoid exceptions.

2.2 Motivation

Our algorithm should focus on building tunnels all the way down to the golden nugget that is located in one of the three hidden objectives, since we are aware that both players want to reach the same goal, the nugget. Thus, there is no urgent need to choose between building tunnels or adding *obstacles* to any other directions. In addition, because both players have the same goal, it is worth considering if the opponent would help build useful *Tiles* in the pro-stage game. In other words, two players are either helping out or impeding each other. We did not take consideration of hindering the opponent in our candidate approach in Section 3.1 but we found it quite useful for improving our approach. Therefore, the priority of *Malus* in our agent's move logic was reduced from the top to a more conservative position. Only when the agent reaches the dangerous zone, should we consider using *Malus* cards if possible.

Aside of the straightforward intuition of building valid paths directly to the goal, another motivation of our approach is that it strictly follows simple-to-implement and greedy rules at each step. This imitates the behaviour of a beginner or mediocre Saboteur player who does not know about his or her opponent's hand of cards.

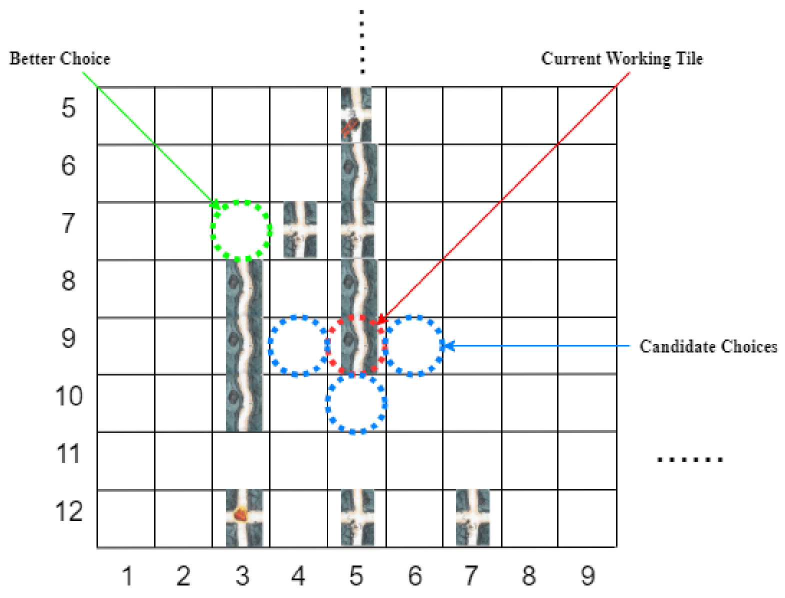

Figure 1: Illustration of Disadvantage 2.

2.3 Advantages and Disadvantages

Advantages:

1. Our agent demonstrates a very stable and reliable performance when competing against players without consistent strategies like the random player, as shown in Section 3.
2. Our agent tends to mostly aim for the goal and hinders its opponent only to a limited extent, which means that the average length of a game is reduced and our agent is very probable to reach the goal if a sufficient number of cards are provided.
3. Our approach is easy-to-implement and intuitive as discussed in Section 2.2.

Disadvantages:

1. Our agent has trouble impeding its opponent except for playing *Malus* cards. (Nonetheless, for the future work, we plan to integrate the Obstacle Generation strategy introduced in 3.2 to strengthen our agent's ability of hindering its opponent.)
2. Due to the fact that the algorithm is working on the lowest entrance-connecting *Tile*, it is difficult for the algorithm to find an alternative path that is potentially better than the current. For instance in Fig.1, it is obviously a better choice for a human player to consider filling block (7,3). However, due to lack of corresponding logic, our agent will stick onto the candidate blocks and miss the better one. Consequently, it will take more steps to reach the revealed goal.

3 Comparative Studies

Throughout our work, we have tested various approaches which are useful for building the ultimate version of our agent. This section summarizes these approaches, their strengths and

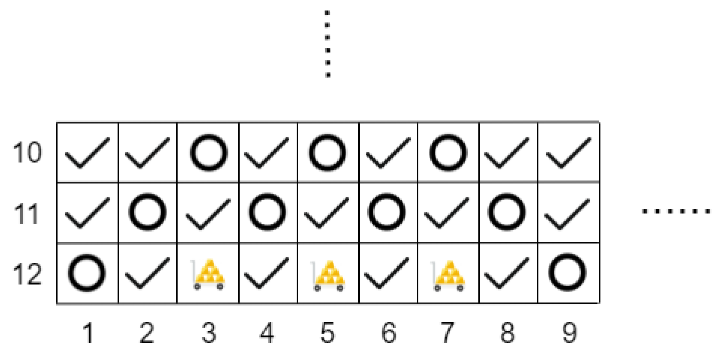

Figure 2: Illustration of positions to place good tiles and obstacles.

drawbacks. A summary of our experiment result is presented in Table 1 at the end of the report.

3.1 Consolidate Step By Step Search Downward

The main idea of this approach is simply trying to dig down as much as possible while blocking the opponent as often as possible. The algorithm is greedy and emphasizes the priority of vertical tiles, i.e., *Tile* 0, 6, and 8. The preferred order of cards is *Malus*, *Map*, *Bonus*, *Vertical Tiles*, *Horizontal-turning Tiles* (*Tile* 5, 7, 9, and 10), *Destroy*, and *Dropping Dead-end Tiles* (*Tile* 1 to 5 and 11 to 15). The essential logic of the algorithm is to first use *Malus*, if we have it, to mute the opponent. The next step is to check if there is a *Map* available in hand, and use it accordingly. Then, the algorithm tests and removes the *Malus* that were blocking us. The algorithm then finds the *Tile* on board with the deepest vertical position(s) and connected to the entrance with a direct tunnel. Starting from this *Tile*, new *Tiles* are being added following the card-using order until a complete path from the entrance to the goal is achieved.

Nonetheless, this algorithm is very case-specific; detailed specifications of if-else syntax and recursive helper methods lead to a large portion of unessential calculations during run time. More importantly, the logic of case specification could be really complex, and missing a single case could lead to unexpected logical errors that are difficult to be spotted. Practically, this algorithm performed quite poorly in our experiments.

3.2 Obstacle Generation

Obstacle generation method is implemented very similarly to our best-so-far agent except for its opponent-hindering policy. When this agent of *obstacles* steps inside the dangerous zone, its vigilance is triggered and becomes quite careful with each step. If it was two steps (measured in Manhattan distance) away from either a nugget or a hidden objective, it is afraid that if it placed a constructive *Tile* at the current position, its opponent may take advantage of that *Tile* and then directly reach the nugget. Therefore, it is always safe to place an impeditive *Tile*, i.e., a dead end, if the agent was two steps away. A detailed illustration Fig.2 is provided, in which checks denote places for constructive *Tiles*, circles for *obstacles* and trolleys for the nugget and hidden objectives.

The feature of this method is obvious. Both pro and con are that obstacle generation method is in the forever loop of hindering its opponent rather than actively searching for the nugget itself. It is unlikely for this agent to lose nor to win.

3.3 Monte Carlo Tree Search

Monte Carlo tree search (MCTS, [2]) is a heuristic search algorithm of decision processes and it is frequently employed in game playing, e.g. Go, in 2006. This approach, however, did not produce promising results in our experiments.

The main problem with MCTS is that it tries to obtain certain information through simulating random games and, based on which, decides the best move for each round. This logic is seemingly flawless but does not fit in our scenario. In this adapted version of Saboteur, almost every random game ends up with a draw, thus all the obtained data is very skewed (because entries contain draws only) and is actually useless. To tackle this "self-loop", we also tried to make game simulations without random moves, but at that point, we were basically implementing an algorithm to solve the simulation problem, whereas directly using this algorithm to play the game does seem much more intuitive.

4 Discussion

We summarize three aspects of possible future improvements or new strategies of our agent.

Minimax algorithm ([4]) is a decision-making rule of minimizing potential loss incurred by worst-case scenarios and of maximizing potential gains. It is often deployed in both game theory and artificial intelligence. Particularly, researchers have utilized this algorithm in extensive board game playing [3]. We, however, have encountered some challenges when implementing Minimax in Saboteur. In this case, defining a proper utility function is tricky given a large number of successful states, intermediates, and types of possible cards. Future work may focus on developing a robust Saboteur-specific utility evaluation scheme.

It is worthy to further investigate the effect of the obstacle generation algorithm mentioned in Section 3.2. We believe that further expansion of the dangerous zone could give the agent a chance to manipulate its series of final steps ahead of time, and enhance the winning rate. Nevertheless, more complex and intelligent logic of planning will be needed simultaneously.

The third aspect of possible improvements is adding a "bottleneck" restriction to the *Tile*-building logic. Specifically, three potential goal positions are deterministic. It may be useful to restrict the algorithm in certain way such as forbidding adding useful *Tiles* outside the x-coordinate range of the three goal positions (i.e. x = 3 to x = 7). These restricted areas seem to be less advantageous in a game with a size-limiting deck because we are obligated to use good and constructive *Tiles* in the most efficient positions before using up all the cards in deck. However, due to the quite narrow corridor for *Tile*-building and a severe consequence of blocking, if the opponent gave a dead-end in our agent's planned path after we apply the "bottleneck" restriction, the priority of horizontal *Tiles* should be cut, and the logic of using *Destroy* should be enhanced accordingly.

5 Conclusion

In conclusion, in this work, we have proposed our best-so-far automatic Saboteur-playing agent which outperforms several other players and achieves steadily better performance against the random player. Based on thorough comparative analysis, we conclude that all solutions have their pros and cons, and an optimal solution should be very case-specific.

Table 1: Experiment Results (Appendix)

	Agent vs Random		Agent vs CSBS		Agent vs Obstacle		Agent vs MCTS	
	Agent: 1	Agent: 0	Agent: 1	Agent: 0	Agent: 1	Agent: 0	Agent: 1	Agent: 0
1	W	W	D	D	D	D	W	D
2	D	W	D	W	D	D	W	W
3	D	W	W	D	D	W	W	W
4	W	D	D	D	W	L	W	W
5	D	W	D	D	D	D	D	W
6	W	D	D	D	L	D	W	D
7	W	W	D	L	D	W	W	W
8	W	L	L	D	D	D	W	W
9	W	W	W	D	W	W	D	W
10	D	W	W	D	D	D	W	W
11	W	W	D	W	W	D	W	W
12	D	W	D	D	D	W	W	W
13	D	W	W	D	D	W	W	W
14	W	D	D	D	W	D	W	D
15	W	W	D	D	D	D	W	W
16	W	W	D	W	W	D	W	W
17	D	D	D	W	D	W	L	W
18	W	W	W	D	D	D	D	W
19	W	D	D	D	L	D	D	W
20	D	W	D	W	D	D	W	W
#Win	12	14	5	5	5	6	15	17
#Lose	0	1	1	1	2	1	1	0
#Draw	8	5	14	14	13	13	4	3
Win %	0.6	0.7	0.25	0.25	0.25	0.3	0.75	0.85

Bibliography

[1] S. Borowiec. Alphago seals 4-1 victory over go grandmaster lee sedol. *The Guardian*, 15, 2016.

[2] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

[3] D. Kalles and P. Kanellopoulos. A minimax tutor for learning to play a board game. In *18th European Conference on Artificial Intelligence, workshop on Artificial Intelligence in Games, Patras, Greece*, pages 10–14, 2008.

[4] M. Nikulin. Hazewinkel, michiel, encyclopaedia of mathematics: an updated and annotated translation of the soviet" mathematical encyclopaedia. *Reidel Sold and distributed in the USA and Canada by Kluwer Academic Publishers*, 2001.