AI strategy of the Settlers of Catan

R04921049 柯劭珩 B01901192 詹鈞翔

1 Introduction

The Settlers of Catan, a resource gathering and building board game, provides an interesting example of a non-deterministic and non-zero-sum game. Such characteristic lets it become a good resource for experimenting AI strategies. Previous work points out that traditional searching techniques do not work well on it; while the Monte Carlo search is much more powerful in the game. We decide to further investigate the Monte Carlo search techniques, by both optimizing and speeding up the Monte Carlo tree search agent. Furthermore, Sierra et al. also pointed out that initial placements of resources have drastic influence to the game outcome. Thus, with the concept of Bayesian inference, we also experimented and found out the initial placements which hold the maxima likelihood of winning the game. After the improvements, our agent exceeds 90% of winning rate in a two-agent contest against a random agent. We also designed a series of experiments in order to measure the power of every improvement.

2 Background

To sum up *The Settlers of Catan,* it is a resource-gathering and building game. Player gathers resources stochastically (determined by dices) from hexagon tiles nearby their settlements and cities, which is essential to build settlements, roads and cities. Players win victory points from the buildings, and the goal is to reach a designed total of victory points. Cities give double resources and double victory points than settlements, but can only be upgraded from settlements and needs valuable resources to build. Since a player can only build on placements reachable from his roads, it is a locally exploring process. Each player chooses two places to place two settlements, and start from there. There are rules that prevent players build near the other player's properties, and some more complex rules that riches the game. With lots of extensions and versions, *The Settlers of Catan* is a pretty well-known and well-investigated board game.

3 Previous work

They were several research papers on exploiting AI approaches in the game. In a recent term paper in Stanford U.'s AI class, Sierra et al. built a simplified model (and visualization) of the game in Python. They showed that traditional searching techniques, such as the Minimax search and the Alpha-Beta search, do not perform well due to a large branch factor, which limits the search depth of any traditional search method. Since the agents were only allowed to search for several layers, their decisions were generally suboptimal.

However, the found Monte Carlo simulation to be very useful. Applying the Monte Carlo search, their agent can outperform a random agent around 80% of the time. Furthermore, they also pointed out that initial placements of resources have a big effect in the game outcome. It can be argued that those 20% games their agent lost can be blamed on bad initial conditions.

Their work, however, is based on a very simplified game model. The initial placements of resources were random assigned instead of chosen by players. To accelerate the game, each player starts with 4 random settlements and 4 roads, instead of 2 each in the original game, which fortifies the effect of randomness on the game outcome. Each player is strongly limited to only one action per round, be it a trade or a building action. Moreover, their game model allowed resource trading only with the bank but not the other players, and the trading ratio with the bank is erratically set to be 1:1, instead of 4:1 in the real game. This made resource exchanging too easy, which their agent exploited to offset initial advantages. There were some other mistakes regarding the rules, and the subtler part of the game, like the robber, the special outcome of dicing a 7, and the development cards were not implemented.

4 Approach & Methods

With permission from the authors, we directly used – and built on – their game implementation, in order to save time and focus on the AI part of the work. We did several improvements on the previous work. In addition to correcting all the bugs we found, we also generalized the game model to enable one trade and multiple building actions in every round, a step closer to the real game. We also enabled trading between the players, which is the most challenging and interesting part of the real game. To model trades in the real world, we implemented game-theory based decision process into our agents. We also made several optimizations to the Monte Carlo search agent, which speeds up the process. Finally, we attacked the problem of initial conditions, using the concept of Bayesian inference to find out the "best" starting points on the given board layout. This information can be directly used by the agent once we tweak the game model to allow the players to choose starting points manually.

4.1 Generalizing the game model

The original game model was built similar to the famous Pacman project used in AI classes everywhere. The first task is to generalize the model to enable multiple actions per round. However, we cannot encode all possible consequences of actions to the legal actions, since this means the branch factor will be infinite, and will remain remarkably large even we limit the trade to happen once per round. Hence we keep the "one-action-per-round" structure, but do not switch players each round. The current player now loses his right to move only if he chooses to pass. This part, though a considerably change, did not seem to have a large effect on the agents' performance and speed.

Next came the task to enable the player trades. Trading actually accounts for the lion's share of the legal actions; the amount of legal building actions is generally around 10 due to a player need to explore to board locally, but since there are 5 different kinds of resources in the game, there are already 5*4=20 possible trades even after fixing the trade partner and the trade ratio. Thus, in a two-player game environment, enabling trading with the other player with a fixed ratio promptly adds another 20 legal actions. In fear of the ballooning branch factor, we decided to fix the trade ratio to be 1:1, which keeps the branch factor at around 50. However, 3:3 trades are also allowed later due to other issues, which will be fully described in section 5.

Since trading between players is an interaction between agents, we need to give any player the right to accept or reject any trade proposal. Henceforth we also implemented decision making methods into agents to solve the problem. In our implementation, the current player can directly trade with the bank (with a ratio of 4:1, thus generally losing resources) or propose a trade to the other player, with the risk to lose the right to make a trade again if the proposal is rejected.

Before we can fully investigate the influence of player trading on the game outcome, we found it slowed down the MCTS agent by much. This leads to our work in optimizing and speeding up the agent, which is described in 4.3.

4.2 Propose/accept the trade

The implementation of a random agent is straightforward. When the random agent has the opportunity to trade, it random chooses from all legal trading actions (including not making any trade at all). When a trade proposal comes, it accepts 50% of the time.

Our MCTS agent, with the expectation of intelligent performance, needs more deliberate approach. First and foremost, we observed that all amounts of resources in any player's hand, are actually public information in *The Settlers of Catan,* since resources came from public stochastic action (the dices) on the game board, and any trading and building actions are public. Assuming one agent knows the type of the other agent, it can correctly forecast the enemy's thinking process in evaluating a trade. This way we can model the player trades into a two-stage game with public information.

Applying the notion of game theory, our agent seeks to maximize the expected utility, in his opportunity to make a trade. The "utility" for a Monte-Carlo-simulation-based agent, if exists, will definitely be the winning likelihood; hence our agent calculates the winning likelihoods after any possible trades. Taking into account that a trade proposal does not get accepted every time, the agent will select the trade proposal that gives a maximum increase in winning likelihood, compared to that in the current game state.

$$\mathrm{BestTrade} = \underset{t\,\in LegalTrades}{\mathrm{argmax}}\ P_{accept}(t)\cdot(Win\%(t) - Win\%(now))$$

Therefore, when facing a random agent, our agent forecasts that any proposal sent to the other agent will be accepted half of the time, and will set all the probabilities in the above equation to 50%. This means our agent will propose a trade to the random agent only if the benefits from it is two times more than the benefits from the best trade with the bank.

If the enemy is a MCTS agent itself, which we didn't experiment with though, our agent should forecast the enemy's decision by actually do the Monte Carlo simulation for the enemy. If the simulation result leads to a rejection, our agent knows that such trade will not be accepted, and will remove it from the consideration by setting the accepting likelihood to 0. If the simulation result leads to an acceptance, it is set to 1. Note that our agent will not propose a trade if it feels that every possible trade will decrease the winning likelihood.

Though we observed that our agent makes reasonable decisions with this method, some strange behavior will happen in extreme cases. This leads to some improvements, which will be discussed in 4.4. Also we acknowledge that this is not the optimal strategy in evaluating trades; this will be discussed in section 6.

4.3 Optimizing and Speeding up the Monte Carlo simulation

The reason behind using Monte Carlo search when attacking *The Settlers of Catan* is that the branch factor is too large for an exact search. Hence, to make decisions in a reasonable amount of time, the MCTS agent could not do lots of random playouts either – the authors set the default to be only 50 random playouts per one decision process. However, after allowing player trades, the branch factor is around 50 itself; how can we obtain appropriate results from doing only 50 random playouts in a tree full of nodes with 50 children?

To solve this problem, we decided to omit the possibilities of trades in our Monte Carlo simulations. We modified the code to distinguish between “legal building actions” and “legal trading actions”, and allow only the former to happen in simulation playouts. In other words, our agent assumes that no trades will happen again ever – in its simulations.

This simplification shrinks the branch factor to around 10 again; while it shortens the average elapsed time for a single random playout (since players will only build over and over), the influence of it on the decision making is so small that we cannot observe any differences in most scenarios. Although it did cause issues on some particular cases, we decided that such simplification is worth the trouble, since it allows our agent to make reasonable decisions in only 50 playouts, while we really cannot go beyond 100 with our resources in the game process.

4.4 Optimizing and Speeding up the Monte Carlo search agent

In the process of evaluating possible trade proposals, our agent has done a full round of Monte Carlo simulation on every possible trade scenario, including the one when no trade happens. Whether a trade happens or not, our agent faces the decision of which and where to build next; the Monte Carlo simulation needed will be exactly one of the simulation performed in the previous process! We found that our agent was doing redundant work.

To speed up the decision, we make our agent memorize the best action in the first layer of both the Monte Carlo search trees from the outcome of the proposed trade and from the outcome of nothing happens at all. Thus our agent will know what is the next optimal action no matter the proposed trade is accepted or not. This saves a full round of Monte Carlo simulation’s time, and does not affect the agent’s decisions.

We also found that our trading analysis led to suboptimal actions in extreme situations. Namely, when our MCTS agent believes winning is around the corner, the current estimated winning likelihood will be exactly 100%. Frequently, it will remain at 100% after a possible trade; now our agent cannot distinguish between good trades and bad trades, since they all expect to have no influence on the winning likelihood. To solve this issue, we simply tell our agent to stop trading if the estimated winning likelihood is above 95%. Since our simulation excludes trades, a >95% winning percentage means that we will win without trades most of the time – then why bother to trade?

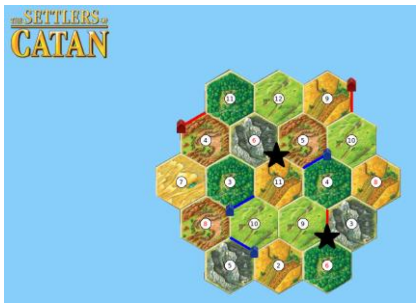
We found that this modification slows down the “must-win” games, since our agent cannot further improve its situation by trades. However, it prevented our agent from doing stupid trades when it is about to win, which was the case before the modification. Similarly, when it is the case that our agent thinks it will lose after any trade, we tell it not to trade. However, this leads to a subtle bug which makes some of the games to be very, very long. This is described in section 6.1.

4.5 Bayesian inference for analyzing initial placement

Both previous work and our early experiments point out that initial conditions are quite important in *The Settlers of Catan*. In some extreme cases, the disadvantage was so large that it is nearly impossible to overcome by our agents, and maybe any agents, since the random player wins after several rounds of play. To measure the true strength of our agent, we need to take away the effect of initial conditions.

We attempted to find out which positions come with the largest likelihood to win the game by the concept of Bayesian inference. According to the real game rules, initially one player can place two settlements on the board. If our agent goes first, then choosing the place most likely to win will be a good strategy; it is also advisable to choose from the remaining options when the enemy goes first.

To analysis the winning rate of all the possible pairs of initial placements, we went by simulating the game (excluding trading as well) with two random agents, with one of them starting at specific initial placement of settlements, while the other starts from random places. We looped through all possible pairs of initial placements, simulating 1000 games for each pair, with a game limited to 600 rounds to prevent endless games to happen. To our surprise, some particular pair of placements led to as high as 990 wins! We also found out that many pairs of initial placements lead to no wins in 1000 tries (though some of the games may result in a tie), meaning that it is impossible to win from those placements.



The two black stars in the figure shows a pair of initial placements that comes with a 99% winning likelihood.

Such analysis tells us which resources are important at the beginning of the game, and what strategies of choosing the initial placements may raise the winning rate of the agent. It came with no surprise that sufficient supplies of lumber and brick is very important, which the original authors had also observed so that they actually guaranteed all players to have at least these two kinds of resources in the beginning. These two resources are needed to build a road, thus making the exploration possible. However, the best initial conditions features balanced supply of all kinds of resources, and a better chance to actually collect them (the numbers on the nearby hexagon tiles closer to 7).

Of course, such analysis also tells us the optimal initial positions on the given map. Even a random agent can win 99% of the time against another, if it begins at the right positions.

5 Experiments and Results

To measure the true strength of our agent, we designed four different experiments: the RR, RG, GR and GG experiments. First we picked up three distinct pairs of initial placements that gives >98% winning likelihood. We call these three pairs of placements the “G positions”, or good positions. In contrast, a “R position” is randomly chosen from all initial placements with lumber and brick guaranteed, just as the default in the original author’s implementation.

Thus, RR means both player start from R positions. The random agent always plays first, so RG means that the random agents starts with R positions, and our MCTS agent starts from G positions. This is the best scenario for our agent. The meanings of GR and GG follows. Note that the random agent “picks” locations first, so three distinct pairs of G positions are provided to guarantee good initial placements even if the random player happens to select on some good positions. Again, to speed up the game simulations, we begin with 3 initial settlements instead of 2, with the third one randomly assigned to the players after the choices of the first two.

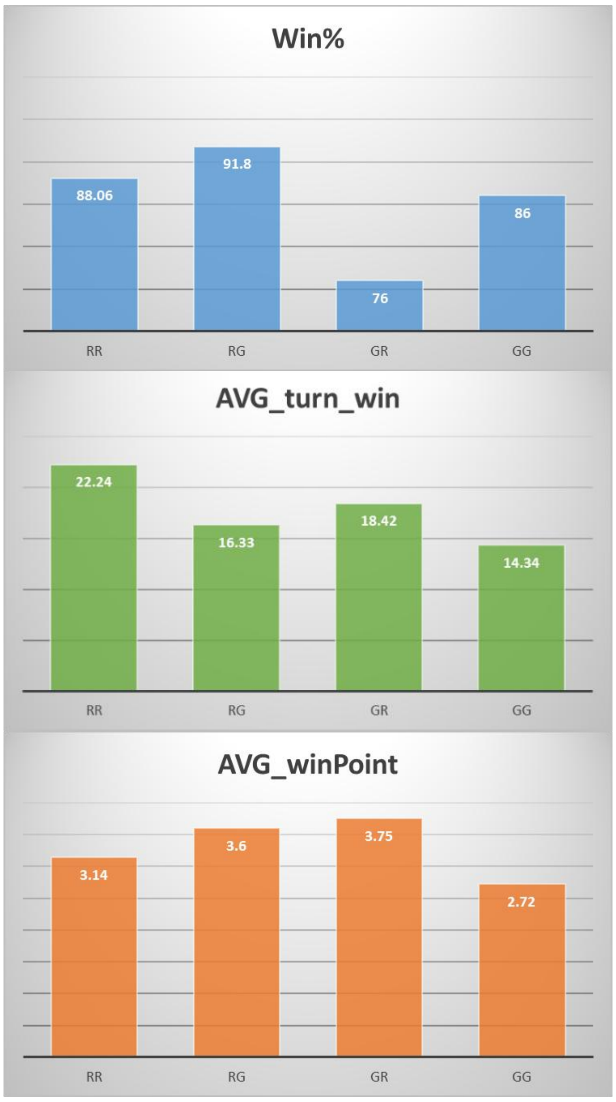
We estimated the overall winning likelihood (Win%), the amount of rounds passed when winning (WinRds), the point gap between agents when winning (WinGap) in all these scenarios. Due to some of the bugs described in the next section, some of the games were discarded, so the total amount of games differ among all scenarios. Last but not least, another round of RR experiments were added, with a time limit for 30 minutes per game. Any game longer than the limit was discarded.

5.1 Results

Scenario	**RR**	**RG**	**GR**	**GG**	**RR-L**
Games Played	67	49	50	50	100
Wins	59	45	38	43	79
Loses	8	4	12	7	21
Win%	88.06%	91.8%	76%	86%	79%
WinRds	22.24	16.33	18.42	14.34	20.66
WinGap	3.14	3.6	3.75	2.72	3.16

RR-L stands for RR with limit.

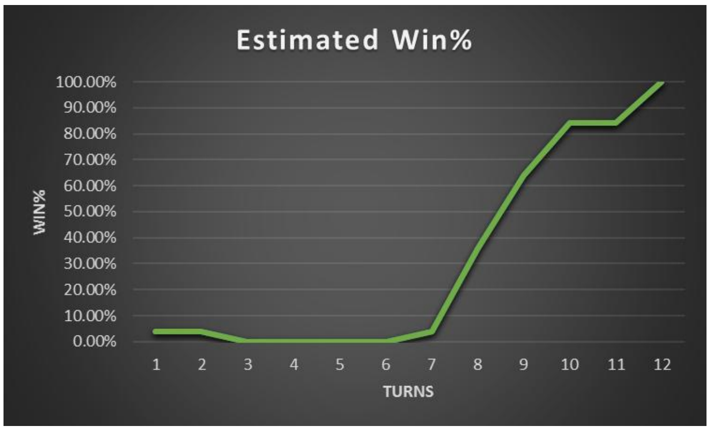
Figures that compare the first 4 scenarios are shown in the right.

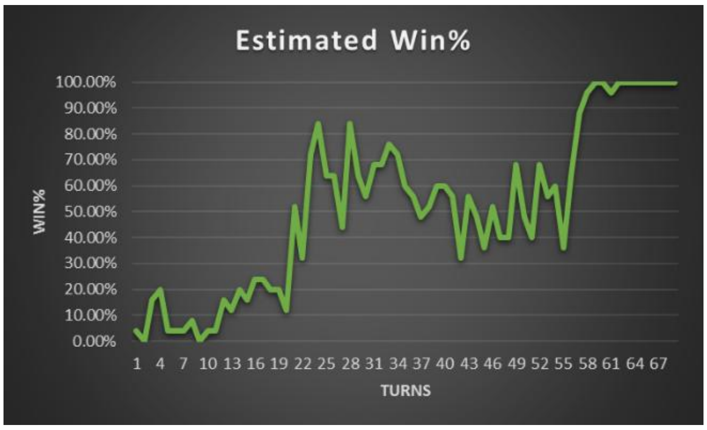


5.2 Implications

First, we found that our MCTS agent can win around 85%~90% of the time with “fair” initial placements, i.e., in RR and GG scenarios. It is remarkable that our agent still wins 76% of the time in the worst scenario, in which the random agent is given the best initial conditions, but our agent needs to start elsewhere. This shows that the Monte Carlo search is good enough to overcome some very bad initial disadvantages.

To our surprise, our agent only wins 91% of the time in the best scenario, in which we expect close to 100% winning likelihood. This, however, could be related to that there is still one settlement random placed for every player. This leads us to speculate that our agent could have won almost all of the games if we start from only 2 settlements.





We also found that the game ends faster when any player moved from R positions to G positions. While this is expected with regard to our agent, the game becomes shorter even when the random agent starts in better places. We believe that when the random agent starts in better positions, it will collect more resources, thus increasing the possibility of obtaining something that our agent needs. Then it is more likely that our agent exploits the randomness of the random agent and trade successfully with it.

We observed that the game tends to be more close in GG games, since both player start from good positions. The random agent can gain points easily even with random moves. The biggest gap, surprisingly, is in the GR games; note that this only takes into account games our agent won. We believe that this is due to our agent losing more close games in this scenario.

Finally, we observed that the winning likelihood of our agent decreases when we put a limit on the game length. This shows that our agent is more likely to win the longer games, in which no players can easily win. Generally, more intelligent behavior is needed to win in this kind of games, so the limit handicapped our agent a bit.

We have also recorded the current estimated winning likelihood that our MCTS agent calculates. Two of them in the games are shown below, with both of them being RR games. The first game shows a quicker resolving one, and how our MCTS agent found a way to win after starting poorly, thinking it is going to lose. The second game shows that randomness still plays a role after the initialization: after our MCTS increases its winning likelihood to around 80%, randomness in resources awarding can lower it to slightly less than 40%. Our agent still overcame that and won finally.

5.3 Comparisons to the original model

Aside from corrections of bugs, the biggest differences between our model and the original model are enabling player trades and enabling multiple actions in each round. We believe that enabling player trades swings the game more toward our agent's favor, since a random agent is totally stupid regarding to trades. Meanwhile our MCTS agent can to some extent correctly evaluate possible trades and decide from them. Indeed, our MCTS agent achieves a higher winning rate then the authors'. It is not clear how multiple actions in each round affects the game, which may need further testament.

6 Issues & discussion

In this section we discuss some problems that we face during the project and how we solve them (or tried to). It is interesting that, while simplifying the game model makes the game easier to solve, but also causes some problems and bugs during the simulation. Thus, how to simplify the game model without losing the characteristic of the game and let the game preform as usual is an important task.

6.1 Will the game be stuck in some cases?

One of the bugs we fixed from the previous work is the limitation of amounts in settlements, cities and roads, which was not implemented in the original authors' work. After the correction, it became impossible for any player to win without building a city, which needs particular types of resources, like ore, that is not given to be collectable by players. In the scenario when both players have reached the maximum number of settlements and roads but remain without reach of any ore resources, the only possible way for some player to win is to trade with the bank. The random player is not intelligent enough to do this, so our agent needs to win to prevent the game from stucking.

However, building a city needs multiple ore resources. This means that winning is impossible even after one trade with the bank, since a trade only nets one unit of ore. Since the simulations exclude trades, our agent does not see it can win after such trade, and will simply give up since no one wins in all simulations. This causes a stuck-game bug: while a normal game takes around 5 to 10 minutes, some of the games remain stuck after 3 hours of simulation.

We have not totally solved this bug other than simply suspend the game after 30 minutes of play. In theory, this bug can be solved by relaxing the trading ratios in each round from 4:1 / 1:1 to 12:3 / 3:3 also, since at most 3 units of any resources suffice to build a city. However, we only relaxed it to 3:3, 1:1 with agent and 4:1 with bank, since it is very difficult to any player to store 12 units of any resources. Our implementation cannot allow three different types of trade happening in one round.

In the stuck-game cases above, when all the trade holds a 0% winning rate, our agent will pass until some type of the resources accumulate to 3 units. If the enemy has many the needed resources, such 3:3 trade will raise the winning rate to be something positive, thus letting the game go out from the stuck point. However this does not solve the scenarios when both agents do not have some common types of resources.

6.2 Trade-off between choosing maxima root value or maxima children value

When using multiple Monte Carlo trees for decision making, the best way to exploit it is actually not looking for the max value for each root (which is the method we use), but the max value among each root's best children. The reason is that we can reasonably expect that we still have the right to do building actions, no matter if a trade is completed. Hence we can actually play the best action after the trade phase, which corresponds to not the winning likelihood given by one whole Monte Carlo tree, but the maximum values in each tree.

However, since there are not sufficient total tries (default only 50 random playouts for one tree), the tree is frequently not fully expanded, especially in cases while some action is clearly better (like building settlements and cities), and small sample size affects the values of the root's children extremely. We can find 100%-winning rate children in every trees, which doesn't tell us much since it may be only one win in one playout. Thus the better decision method does not work.

One way to solve this problem is to merge these MC trees into one larger tree, using the first layer to represent different trades. This is probably the proper way to do it, but there are technical problems in implementation regarding to how we handle trades. Thus we settle for only using the total value of the root to finish the decision process in reasonable time, while such trade-off does not lead to a significant effect on Monte Carlo agent.

6.3 Unbalanced search tree

With highly expected winning rate, it may be the case that some actions are clearly better (similar to the last paragraph), which will cause a very unbalanced MC search tree, since MCS tends to search better nodes. It will sometimes cause the bug of querying a non-fully-expanded node's best child. Our Monte Carlo agent is using the UTC exploring strategy. The Upper Confidence bounds applied to Trees (UCT) is an exploration strategy often used in combination with MCTS that will balance the deep search of successful nodes against the exploration of new game nodes. Such technique tends to make the search tree more balance and avoid the local searching happened, but our bug still happens from time to time. In the cases when our MCTS agent starts with optimal initial conditions, this phenomenon frequently happened, since the expected winning likelihood hovers close to 100% all of the time. Fortunately, it does not lead to harmful effect on the agent performance. Since the MCTS agent dominated the random agent, and a lack of time, we do not apply further techniques for solving such phenomenon.

7 Conclusion and future work

In our project, we made lots of improvements on the original framework. We generalized the game model to be closer to the real world's game, and implemented game-theory based trading decision making methods in the MCTS model. While the main concept in the agent is the same, we made several optimizations and speed-ups to the MCTS agent, which can evaluate all trades and actions in a reasonable time. Last but not least, we investigated the influence of starting positions in *The Settlers of Catan.* We designed several experiments to test our agent's strength excluding the influence of the starting positions.

Our method of Bayesian inference will not work on random generated board layouts. However, a simple heuristic may do the trick, but it will be very interesting to see if reinforcement learning applies. This is actually considered in our project, but with the limit of no available data of boards other than the default one provided by the authors, we ran out of time and did not implement this part. While we used the data to exclude some extent of influences from initial conditions, we still give players one randomly distributed settlements to begin with, which would not be the case if time permits. A complete experiment that starts games with only the two settlements chosen will show more precise talent level of our agent. The game model also left something to be desired. Full trades, without the limit of trading ratios, can be enabled.

8 References

Guhe, M., & Lascarides, A. (2014, August). Game strategies for the Settlers of Catan. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on* (pp. 1-8). IEEE.

Szita, I., Chaslot, G., & Spronck, P. (2009, May). Monte-carlo tree search in settlers of catan. In *Advances in Computer Games* (pp. 21-32). Springer Berlin Heidelberg.

Cuayáhuitl, H., Keizer, S., & Lemon, O. (2015). Strategic dialogue management via deep reinforcement learning. *arXiv preprint arXiv:1511.08099*.

Kaplan-Nelson, S., Leung, S., & Troccoli, N., (2014) AI Agents to play Settlers of Catan.