CS517-Pandemic-Report

Kallinteris Andreas, Misokalos Ioannis, Orfanoudakis Stavros

December 2019

1 Introduction

In this exercise we were asked to implement an agent that will play a modified version of the board game Pandemic. Through this implementation, we were able to become better accustomed to using heuristic functions in order to find the best result, create a dynamic agent that changes strategies depending on the changes of the environment, and finally, use opponent modeling to facilitate this process.

2 Agent's Strategy

Through experimentation we came up with an agent that can adapt depending on the rest of the player's strategies. The way our agent works is through a set of utility functions that search and evaluate action sequences for each round (a total depth of 4 moves). The weight of the utility that is given to the actions themselves has been decided after meticulous testing(manual batch learning). After deciding on the best action sequence, the agent looks at the suggestions the rest of the players give us for two reasons. Firstly, through those suggestions we can create a high level impression of that agent's strategy(policy re-constructions), and use it to make a model of that opponent. Secondly, if an opponent suggests a move that is out of character for him, we assume he has a serious reason to do so, and therefore give his suggestion increased weight. We then compare the utility of the suggestions with the action sequence we found to be best ourselves, and we follow the one that scores better. Finally, after each round, we adjust the weights we attribute to each kind of action depending on the opponent modeling we have made for the other players, so that we better complement their way of playing. For example, if the other players almost never build research stations, we will increase the importance of building them for ourselves, since if no one ever built them the game wouldn't be balanced. By maintaining this balance we find that we can achieve an acceptable win/loss ratio. It is also worth noting that as the game progresses the models we have created for our opponents become more and more reliable.

2.1 Why Heuristic?

A Heuristic Method (with pruning) was chosen to decide the agent's actions due to the fact that there is a small amount of uncertainty (mainly in the way the game's decks work), which is very easy to manage with the use of easily computable statistics. We also considered using Monte Carlo Tree Search but due to the low branching factor and easily manageable uncertainty it was decided that it would be unnecessary.

3 Agent's Implementation

3.1 Detailed explanation of our Utility functions

3.1.1 Evaluating our agents moves through heuristic search function

In this section we explore and attribute a score to almost all of the available action sequences.

Actions	***DiscoverCure***	***DriveCity***	***CharterFlight***	***DirectFlight***
Weight	CureWeight	0	flightScore()	flightScore()

Actions	***ShuttleFlight***	***BuildResearchStation***	***TreatDisease***
Weight	0	BuildScore()	scoreDisease()

- Both DriveCity and ShuttleFlight are considered neither as rewarding nor as damaging actions, so we score these simple actions with a 0.
- CureWeight : This is a very large constant, meaning that if there is an available cure to discover, we prioritize it.
- FlightScore() is a function which calculates the utility of discarding a card. More specifically, we don't want to discard a card unless we are sure that the disease of the same color can be cured by another player. As shown below we improve the chances of winning by not discarding cards early, or at least before we are sure we can cure the disease. Thus, we prune the nodes that try to discard these types of cards. On the other hand, when it is worth flying, we encourage our agent to discard a card with a small weight. DiscardCardWeight is slightly better than just driving to the next city. In our implementation we use the weight 5 for this action.

Number Of Cards	***#cards <<n***	***#cards <= n***	***#cards >n***
Action	Prune	Prune	Discard Card

  * n is number of cards required for a cure, considering the player's role.

  ** We have also implemented a similar scoring function which can also discard cards of a color that is not already cured. This occurs when it is unlikely for the player to collect all the necessary cards for a cure. This function performs significantly better when the game difficulty is higher and the players need to travel a lot in order to treat the cities with 3 cubes that are probably going to have an outbreak in one of the next turns (but increases the compute time by about 4 times so it was discarded).

Number Of Cards	***#cards <<n and not likely to collect the required cards***	***#cards <= n***	***#cards >n***
Action	DiscardCard	Prune	DiscardCard

- scoreDiseases() : This functions evaluates the threat of the diseases (cubes) on the board by analysing the dangers (outbreaks, running out of cubes) imposed by them. Regarding the threat of outbreaks, it calculates the expected time of an outbreak occurring there by taking into account how many epidemics are left on the player deck, whether that city is in the infect deck, and whether it is discarded or it was previously discarded. It also calculates the damage that outbreak will causes (how it will affect our chance of victory and generate further problems down the line). Regarding the danger of running out of cubes for the diseases, it simply calculates the probability of it happening using statistics by taking into account the discard pile and the probability of an outbreak occuring. In order to calculate the final weight we run this function two times: once before the action and once after it. That way, we can use the difference of these two scores to determine how this specific treatment may affect the game state.

score = scoreDieseases().Before - scoreDieseases().After

* This value ranges between very small numbers(20-200) in the first rounds when there is not any immediate danger of outbreaks and can be significantly increased at the later rounds(100-3000).

- BuildScore(): This function tries to determine the utility of building a Research station by considering the card that would be discarded as well as the number of already built stations and the number of neighbouring cities it has. We also provide a building multiplier specifically for the Operations Expert.

Cases	***card to discard is our main color and not Operations Expert***	***City Neighbours <= 1***	***Distance to nearest Research Station <= 2***
Action	Prune	Prune	Prune

Criteria	***Operations Expert multiplier***	***Number of existing Research Stations <= 3***	***Distance to other Research Stations >5***	***Number of existing Research Stations >4***
Weight	4x (score)	score += 50	score += (60 to 100)	score += (-40 or- 60)

.

- We also score the final position of the agent:
  - scoreQuarantineSpecialist(): This is a Quarantine Special specific function that scores the final standing position of the player piece considering the potential outbreaks and infections that may occur. Specifically it gives a high utility for each neighbor that can be infected in the next turns. This score could range from 0, when there there is no immediate threat of infection, to a few hundred in very dangerous situations.
  - scoreNearCure() : This functions is mainly enabled in the final turns of the game which may return very big scores in the case that the player is about to have enough cards for a cure. This way we are encouraging players who are about to cure in the next round to take into account the amount of actions they will have to use in order to travel to a station in order to cure. This way, they will try to stay closer to a research station, depending on how many diseases they are about to cure.
- **!Note**: All of the previous utility functions score the available moves without taking into consideration the other players' tendencies. Therefore to take into account the other players, we multiply the BuildScore() utility with the value of $BuildTendencyMine$, which is a float that help us decide whether an action is worth being done by us or if the other players are more likely to do it themselves. We also multiply the result of scoreDeseases() with $TreatmentTendencyMine$ and the result of FlightScore() with $FlightTendencyMine$. Each of these three unique variables is produced by our opponent modeling module and will be further explained at the appropriate section later.

3.1.2 Evaluating suggestions from other players

Another important section of our implementation is the part where we evaluate the suggestions from the other players. To do this, we use the same functions as the ones previously described. The only thing that changes here is the fact that other players have more information than us and are also using different evaluation functions than us, so they are more likely to suggest to us different actions than the ones we thought were the best. At this point we have to make clear that we think we have implemented an almost optimal heuristic function which returns the best possible action sequence with the help of all the available information a single agent may have. Therefore, for the suggestions not to be overshadowed by our own heuristic function, we multiply each of the BuildScore(), scoreDiseases() and FlightScore() scores with the corresponding tendency of each agent. For example, when agent 1 suggests that we build a research station we use the following formula to calculate the utility of that specific action:

$$totalscore+ = BuildScore() * (1 - BuildTendency[1])$$

This could be explained as, the greater the tendency of a player to suggest that we build the less weight we put on his suggestions. On the other hand when a player doesn't suggest that we build research stations often, but suddenly want us to build one, we assume that he is signalling us, and we therefore put more weight on his suggestion so that we are more likely to build this station. This implementation was a result of many hours of testing and studying, so we think that's one of the best ways to handle the uncertainty of the other players hands. We calculate all of the players suggested moves this way.

3.2 Making suggestions for other players

One of the most important features of the pandemic board game is being able to suggest action sequences to your teammates. In our modified version of the game we have to be able to suggest a move to another player while having information only about his hand and role along with all of our agent's information (hand, role and opponent modelling for the other agents). In order to achieve this, we made a few modifications on our code:

- We changed our FlightScore() evaluation function to account for the case where we are making a suggestion. In that case we consider our hand as well. This way we encourage a player to discard a card of a specific color when we are about to cure the disease of said color. Ideally, we would like the other player to not ignore this type of suggestion as we are using it to signal information.
- We also modified our BuildScore() function in a way that we now will suggest to a nearby player that they consider building a research station, if we have a cure ready, giving this action great priority which is represented in its score (around 1000). This way, if the distance between us is less than 3 cities, meaning that we can reach his city and have one more action left, we would like him to build a research station for us (if viable) and help us win the game as quickly as possible. The previous consideration doesn't apply when we already have a research station close to us (at most 3 cities away).

3.3 Detailed explanation of our Opponent Modelling

The process of deciding what our strategy would be, as far as opponent modeling is concerned, was probably the most time consuming. At first we tried implementing complex learning algorithms, but because of the nature of the game we had too few data nodes, and therefore there was not enough time to develop a legitimate model of the opponent. We therefore proceeded with an algorithm that would work equally well, or at least almost equally well, during all parts of the game. Unfortunately, the restricted information we had about our opponents made it difficult to implement a more in-depth algorithm. Since our only clue about our opponents was their suggestions to us each time we were about to play, thus we decided to use those very suggestions as the building blocks of our opponent modelling. We then reached a dilemma concerning two distinct possibilities that would be hard to take into account at the same time. On one hand, some agents might use their suggestions to signal us about moves they would like us to do to facilitate them. On the other hand, some agents might simply suggest that we do what they would do in our turn. To assume the former would be an optimistic approach to the problem, and if an agent fell into the latter category it would lead to miscommunication. Therefore, we went with the second, safer option and tried to take the first category into account later, during our decision process.

The main idea behind our opponent modeling algorithm was the maintenance of a balance through policy reconstruction. After running our agent many times with different weights on each kind of action, we have decided on the weights that increase our chances of winning the most. Those weights are the ones we would ideally like for all players to use when calculating their best moves, but this is an unrealistic expectation. Therefore, since we cannot change the way they calculate their best moves, we will change our own strategy to compliment their way of playing(similar to [1] but with far simpler learning method due to the lack of data points). We have made float scales from 0 to 1 for all of the actions which represent the tendency an opponent has to make that action in general. Everyone, including us, starts from the value 0.5 which is the aforementioned balance. During each round, we examine the sum of each type of action that an opponent suggests we do against the sum of the respective type of action we would do if we were to follow our heuristic function. We keep those values in separate counters for us and the opponent. The greater the difference between those counters is, the bigger the adjustment we will make for the opponent's tendency. If an opponent makes more moves of a certain kind (for example treatDisease) than we do in average, his tendency for said action will be adjusted to be greater than 0.5

$$\textbf{treatmentTendency += (1-treatmentTendency)/5}$$

Inversely, if he makes less moves of that kind, his tendency will be adjusted to be lower than 0.5

$$\textbf{treatmentTendency -= (1-treatmentTendency)/5}$$

. At the end of the round, we want to adjust our own weights in a way that maintains the balance at 0.5, therefore the sum of all the players' tendencies for each move divided by the number of the players must be 0.5. So for example if the 3 other players have a treatmentTendency of 0.2 and therefore don't make enough treatments, we want to increase ours by an amount that would satisfy the following formula:

$$\begin{gathered}\textbf{(Agents[0].treatmentT. + Agents[1].treatmentT.} \\ \textbf{+ Agents[2].treatmentT. + treatmentT.)/4 = 0.5}\end{gathered}$$

We achieve this by setting the value of our own treatmentTendencyMine according to the following formula:

$$\begin{gathered}\textbf{treatmentT.Mine = 2 - (Agents[0].treatmentT} \\ \textbf{+ Agents[1].treatmentT +Agents[2].treatmentT)}\end{gathered}$$

And normalise:

$$\textbf{treatmentTendencyMine = (treatmentTendencyMine +1) /3}$$

3.4 Choosing our Actions

We explained in the opponent modelling subsection how our opponent's suggestions affect the way we evaluate our actions, but we haven't yet explained how the process of choosing the best action sequence takes place. All in all, we have 4 action sequences to chose from, 1 from the heuristic function, which is also the one we have deemed to be the best sequence ourselves, and 3 from the other player's suggestions. As explained previously, the utility of the sequence that is provided by the heuristic function is calculated by using standard weights and dynamic factors that change depending on our tendencies. The utility of the sequences that are provided from the other player's suggestion is calculated slightly differently. It is inserted into a function that weighs it by examining each individual action, but this time instead of increasing its value the higher the player's tendency for this kind of action is, we instead give more value to actions that seem out of character for that player based on our opponent modelling, since that is a good way to signal us that they need us to help them do something they wouldn't do by themselves. After each of those action sequences is evaluated we compare them, and the one that scores the highest utility is chosen as our action sequence for this round.

3.5 Learning/ Training Weights

3.5.1 Objective

The objective of the learning process was not only to balance the weights of each utility function but also to reduce the computation time corresponding to each utility function, since the run time had to be kept under a reasonable constraint.

3.5.2 Challenges

The game has a significant run to run variance making reinforcement learning techniques hard to implement. To run batch based learning with every feature of our agent enabled and with every utility function on it's most accurate version/highest complexity, with a batch size at least 1000 (which was decided because it provides a good estimate of the actual win rate) it would take an estimated 12 days for only 1 epoch which was impractical.

3.5.3 Solution

The manual learning / training started with the agent at a state (utility weights, utility function complexity) which was considered from a basic analysis to be good enough. Then, we altered the weights that were used by the utility functions and tested each of those weights keeping the one that returned the best results each time. For each variation, 1000 iterations where tested and the one with the highest win rate was chosen. hat way a local maximum was reached.

4 Results

In this section we will try to demonstrate some of our main experimental results. All of the following test runs were done by 4 identical agents with different attributes each time depending on the test case. Every result that is presented here was the outcome of 1000 iterations [1] of the pandemic game. Also, for each scenario we are testing how the corresponding agent performs on different difficulty levels. One of the main ways to change the difficulty level is by controlling the number of epidemic cards in player deck. So, 4 cards is the normal difficulty, 5 the hardest, 3 is easy, while 2 and below is very easy.

At this point we would like present the metrics we chose to measure the agent's performance. By comparing different metrics we can understand the strategy of the players.

- Win Ratio shows how often our agent wins the game.
- Lose Ratio attributed to insufficient cubes.
- Lose Ratio attributed to Outbreaks, indicates how often the limit of maximum outbreaks is breached.

[1] After many test runs about iteration number, we noticed that 1000 iterations had the least deviation to time ratio of the results, around 1%. This means we have minimized the random factor between our experiments, thus every result is reproducible.

- Lose Ratio attributed to insufficient cards, indicates how often the agent loses because he didn't manage to cure all the diseases in time. On the other hand this metric also shows us how many games were unwinnable due to sequence of the drawn cards of each player.
- Average diseases cured is a metric that shows us, how efficient an agent is at discovering cures.
- Average cubes left in game board pile is a metric that shows us how efficient an agent is at treating diseases. The higher the number of cubes are in the pile, the further the players are from losing the game due to insufficient cubes.

Results:

- At first we would like to set a reference point. In order to do that, we will demonstrate how the initial agent which was given to us as an example performs in comparison to our own agent, which we are going to test later. This agent does not implement suggestions, opponent modelling or any other dynamic utility functions.

Number of Epidemics	***Win Ratio %***	***Lose Ratio % to Insufficient cubes***	***Lose Ratio % to Outbreaks***	**Lose Ratio % to Insufficient cards**	***Average Diseases Cured***	**Average Cubes Left (pile)**
3	0%	6.3%	16.37%	77.33%	0.001	46.97
4	0%	21.97%	42.83%	35.6%	0.001	41.35

- In this table, we disabled our dynamic function scoreDisease() so that we can observe how much it affects the overall performance of the agent. In order to quantify that, we will replace this function with another that is static and much faster. In this scenario, opponent modeling and suggestions are enabled along with all other advanced functionality of our agent.

Number of Epidemics	***Win Ratio %***	***Lose Ratio % to Insufficient cubes***	***Lose Ratio % to Outbreaks***	**Lose Ratio % to Insufficient cards**	***Average Diseases Cured***	**Average Cubes Left (pile)**
2	29.4%	8.2%	26.6%	35.8%	2.6	39.3
3	16.3%	16.1%	55.7%	11.9%	1.7	42.7
4	6.5%	23.6%	68.6%	1.0%	1.3	46.6
5	2.5%	26.6%	70.8%	0.0%	0.6	49.0

- In the following table we can finally see how our completed agent performs while we change the number of epidemics in the player deck.

Number of Epidemics	***Win Ratio %***	***Lose Ratio % to Insufficient cubes***	***Lose Ratio % to Outbreaks***	**Lose Ratio % to Insufficient cards**	***Average Diseases Cured***	**Average Cubes Left (pile)**
2	24.2%	0.6%	1.7%	73.5%	2.8	46.1
3	18.8%	6.3%	20.4%	54.5%	2.4	38.2
4	9.6%	17.6%	52.2%	20.6%	1.7	38.9
5	5.3%	27.4%	62.9%	4.4%	1.1	39.5

Overall, we can see that the best win ratio we have achieved in normal games is 9.6%. This outcome is not as bad as it may seem, due to the nature of the pandemic game. On the other hand, it is obvious we have improved a lot the lose to outbreak ratio along with lose to cubes ratio ,which were the main focus of our experiments. Also, it is obvious that when the game has less epidemics the lose ratio to insufficient cards is getting higher.

5 Running Instructions

All of our code is implemented in class PlayerLAB51743225.java which is a class that extends Player.java. So in order to run our agent simply compile the whole source folder. In case someone wants to run our agent along with others, simply copy our Player*.java to your Player package and change the corresponding name when instantiating the object in SimulatePandemic.java.

References

[1] Sandip Sen Neeraj Arora. “Learning to take risks”. In: *AAAI-97 Workshop on Multiagent Learning* 0.0 (1997), pp. 59–64. DOI: `http://sandip.ens.utulsa.edu/research/web/papers/TakeRisks1997Sen.pdf`.