TER Report

Implementing two artificial intelligences for the Seven Wonders board game

Prepared by
Ossama Ashraf
Kevin Levy
Nicolas Zimmer

Prepared for
Marie Pelleau

Master Informatique
à Université Côte d'Azur
Premier semestre de l'année 2020-2021

Table of contents

Introduction

Group presentation

We are a group of three Master 1 students at the University Côte d'Azur : Ossama Ashraf, Kevin Levy and Nicolas Zimmer.

Each member of the group worked in an equitable manner on each aspect of the project, being : Implementation, Testing, and Documentation.

Subject presentation

The subject consists of implementing two artificial intelligences for the Seven Wonders game that we implemented during the course : *Projet de Développement*.

Seven Wonders is a board game that features ancient civilizations. At the beginning of the game, each player receives a wonder board. The game is played over three Ages, each using its own decks of cards. Each Age contains 6 rounds. The game uses a card-drafting mechanic in which, once per turn, each player selects a card to play from his or her hand, then passes the remaining cards (face-down) to the next player. In each round the players can build, dump or use the card to build a stage of a wonder. For the full game rules please check 7 Wonders Rulebook - 1jour-1jeu.com.

State of the art

We started implementing our AIs on a game that contains the following rules.
A game has three ages of six rounds each.

Players choose a card in their hand, then pass their hand to either their left or right neighbor, depending on the current age.
For every chosen card a player has three options : he can build the card, dump the card to get coins or use the card to build a stage of his Wonder.
At the end of each age every player fights his neighbors to win fight points.
Players can trade resources with their neighbors for coins.
Card decks are different in each age.
At the end of Age three, points are counted, and the winner is computed.
All the different types of points given by cards and wonder stages were implemented.
Card and wonder stages can give/cost resources.
All cards and wonder stage effects were implemented.
The game can only be played with three players.
The game is implemented in JAVA programming language.

Implemented Features

Guaranteed AI

During the first sessions we had together, we decided as a whole that the Guaranteed AI should have the features listed below :

- If the player gets one specific science point, then whenever he again has the choice between different science points, he should go for the same science point as before.
- If the player has a chance to build the marketplace card, he should do it.
- The AI should focus on military cards especially during Age 3 and towards the ends of each Age.
- The player should be aware of the current military might of its neighbors and weigh out whether it would be important to focus on Military cards or not.
- The AI should analyze its neighbor's strategies and try to block them:
    - Play out cards that would give them an advantage (e.g. use them to build a stage of a wonder or dump them)
- Focus on same colored cards
- Prioritize color of cards: depending on the age?
    - Age 1: Green -> Grey
    - Age 2: Green -> Brown -> Grey
    - Age 3: Green -> Red

These features are hard-coded in a class named GuaranteedStrategy, and our “AI” implements these features simply through *ifs* and *switches*.

Of course during the implementation, we diverged a little bit from these features, mainly through trial-and-error. The final implemented features were the following :

- If the player gets one specific science point, then whenever it again has the choice between different science points, it should go for the same science point as before.
- If the player has a chance to build the marketplace card, it should do it.
- The player builds military cards if it has less military points than its neighbors.
- Analyze neighbor's strategies & try to block them:
    - Build cards that would give them a disadvantage or dump it in order to get coins.
    - Focus on same colored cards
- Prioritize color of cards: depending on age
    - Age 1: Green -> Grey -> Blue
    - Age 2: Green -> Brown -> Grey -> Blue
    - Age 3: Green -> Blue

Even though these features are simple, it makes for quite an effective AI as we will see later on.

Ambitious AI

For our ambitious AI, we chose to implement the Monte-Carlo algorithm. Monte-Carlo works in the following way :

- For each action, the algorithm will simulate a given amount of games after having played this action.
- Then, after having played a given amount of turns, it will return a heuristic for every simulated game.

- Monte-Carlo algorithm will then make an average of every heuristics returned by every game for each action, and returns the action that has returned the best score.

For every simulated turn, in order to choose the action to be played, Monte-Carlo will either be called recursively, or an action will be returned using another algorithm.

The depth of call can be either the maximal depth of the game's tree (the score would then be the returned heuristic), or be a fixed value (then a good heuristic must be used if the algorithm stops before the end of the game).

In our case, we initially decided to use a non-recursive Monte-Carlo algorithm (playing random actions during simulations) with maximum depth. Again, these choices were changed afterwards during development, through trial-and-error.

Using 1000 simulated games for each action, and using maximal depth, each action took an average of several minutes to be computed (with actions being multi-threaded), and gave results worse than the Guaranteed AI. Thus we chose to limit the maximal depth of call of the algorithm to a fixed value (this will also be changed later on, using a dynamic depth instead). Doing this we now need to use a good heuristic.

If the depth of call reaches the end of the game, then the returned heuristic is the minimum of the difference between the player's score and the others' scores. If this result is positive, then it means we won the game.

If the depth of call doesn't reach the end of the game, then the returned heuristic is computed using the current score, and the theoretical score the player could have using the current state of the game. Each aspect of the game that is taken into account when the

real score is computed is treated in our heuristic, and a given ponderation is attributed to each aspect for calibration purposes.

We then realised that it would be smarter to use our Guaranteed AI rather than a random AI in order to play the actions during simulations. This gave us slightly better results.

Obviously, this should have been our way to proceed from the git-go because using an AI returning random actions, we assume that Monte-Carlo will not be a very good player in our simulated games, whereas using our Guaranteed AI, we assume Monte-Carlo will then play smart moves (to a certain extent).

Finally, on the advice of Ms. Pelleau, we implemented a dynamic depth in our algorithm. It will not go very deep during the beginning of the game (because we haven't played yet, so "all bets are off"), and it will tend to go to the maximal depth at the end of the game, because the returned heuristic would then be optimal (because it would say if we have won or not).

In the end, we decided to go with parameters for the number of simulations per action that would give us an average of 30s taken in order to compute an action for the following reasons :

- During a real game, the AI cannot take forever to choose an action. An ideal time taken would be between 30 seconds and 1 min.
- For a sample of 100 games, computing statistics would then take approximately 15 hours ! Going any lower than 100 games wouldn't give good statistics, and 15 hours is as much as we can go for uninterrupted simulations on our personal computers.

The results of our implementation will be presented in part 4, *Results & Statistics*.

Project management

First of all we started by specifying the features that will be in both the Guaranteed and the Ambitious AI.

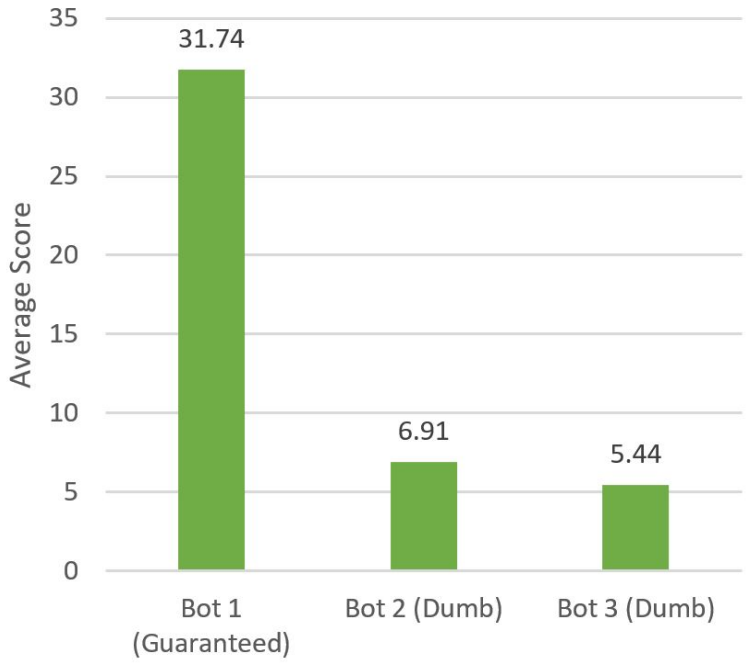
Each feature was developed and tested in separate git branches. As the project progressed, we verified for each added feature if it had a positive impact on the average score; We then decided if the feature would be kept, modified, or entirely removed from the project.

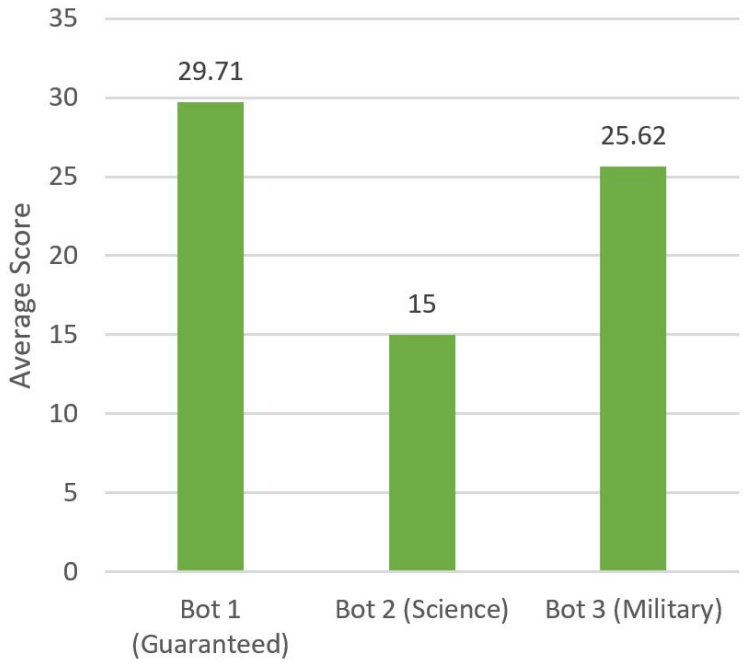
Our work was organised mostly around the meetings we had, where we would first talk about what we had done since our last meeting, assign issues to each other, and then work on our respective issues while giving feedback to each other (much alike a *SCRUM* framework).

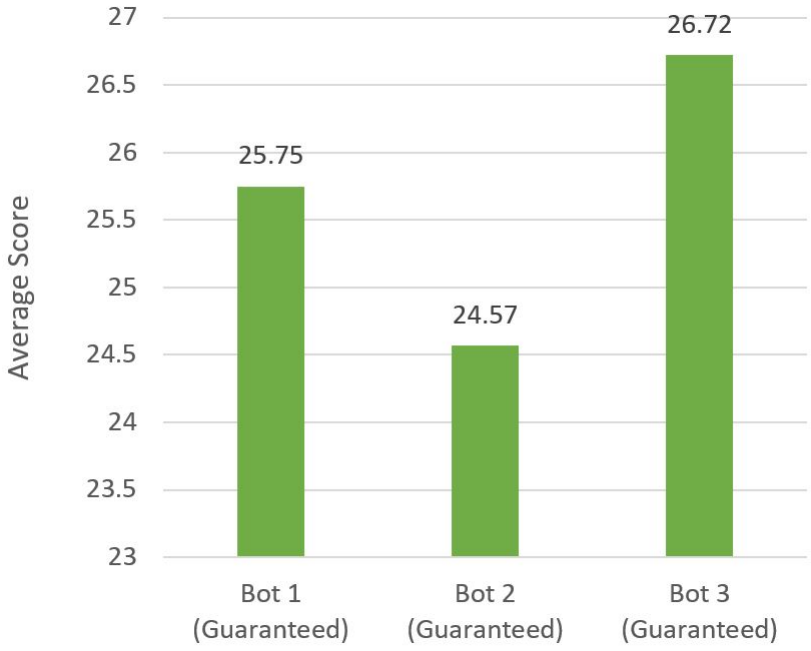
Results & Statistics

All the data shown was computed over a sample of 100 games.

Guaranteed AI







As we can see from these results, our Guaranteed AI performs well against both “Dumb” bots (bots that play their actions randomly) and against smarter bots such as Science and Military. These two last strategies focus on Science cards, and on Military Cards, respectively.

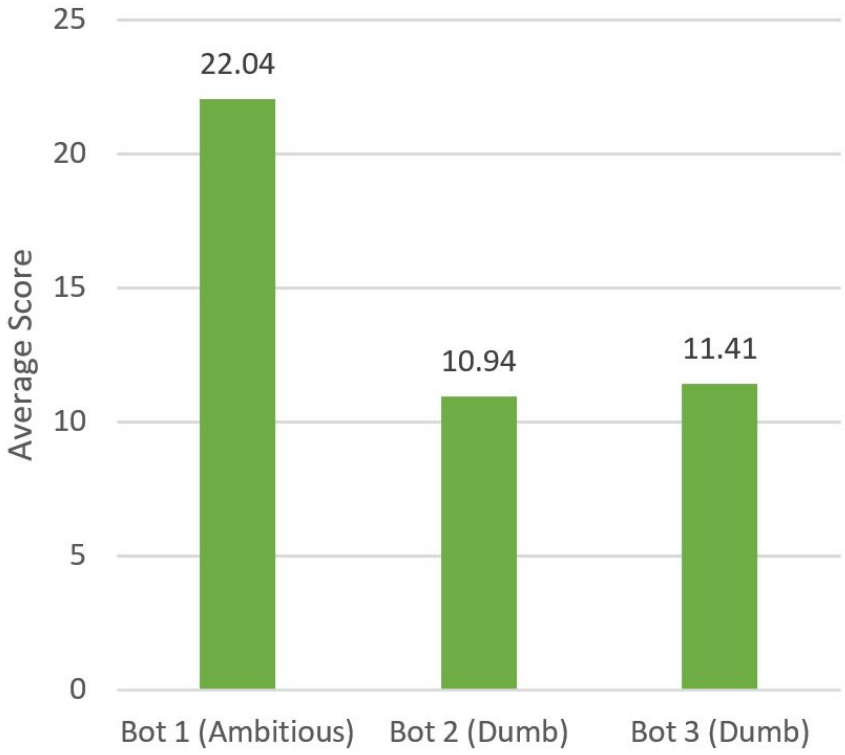
These results are logic since our Guaranteed AI knows how to behave in the different game situations and it gets a lot of military points to win fights while the Dumb bots just chose random actions.

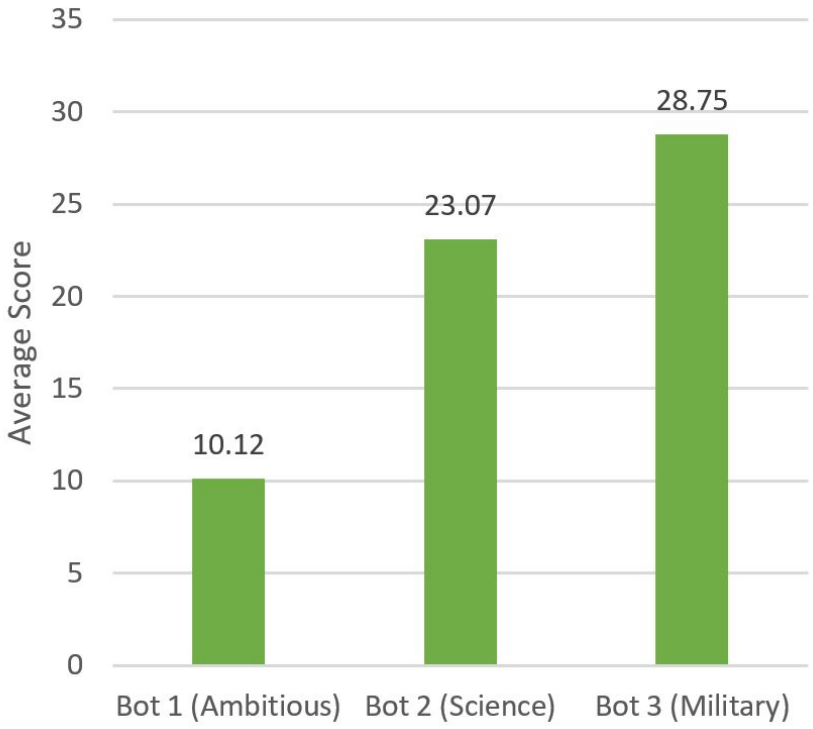
The final results are close to the Military bot, as is to be expected since our Guaranteed AI gets most of its points out of military fights. The used algorithm we ended up with is pretty straightforward, and it was detailed in part *3.1. Guaranteed AI*.

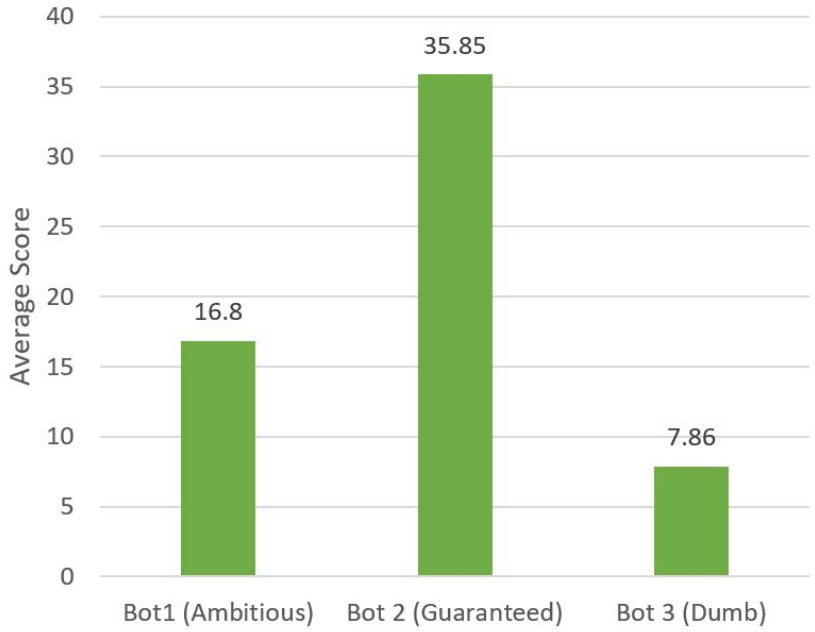
Overall our Guaranteed AI returns the best results when it is tested against all the other bots and AIs we have implemented.

When the Guaranteed AI is playing against itself, the results are obviously very similar. Since the game is subject to randomness (while dealing cards for example), small deviations can be observed, which could be compensated for with a larger statistical sample.

Ambitious AI







The results returned by our Ambitious AI perform acceptably against two “Dumb” strategies, but severely lack strength against other types of AIs, such as Science, Military, and of course Guaranteed.

This lack of performance can be explained simply by the depth used and the number of simulations per available action run : In order to have tolerable computational time, we decided that an action shouldn’t take longer than an average of 30s. Running on our personal computers, this amount of time is too short to give good results : The limited number of simulated games would result in a bad statistical sample, which would lower the probability of the answer given by the algorithm being correct, and the limited depth forces us to return a non-optimal heuristic (if the depth doesn’t reach the end of the game).

These two factors combined, themselves being limited by the sheer time complexity of the algorithm, explain why our results for the Ambitious AI are lower than for our Guaranteed AI.

Even more so, in order to get these statistics, each bar chart featuring the Ambitious AI took around 12 to 15 hours to get the necessary data, on a reduced sample.

It is very probable that using more powerful computers, in the same amount of time, the results returned by our algorithm would be better; but to what extent we do not know, since we cannot run statistics with large enough samples, higher depth, and more simulated games because of the time it would take.

However, it is our general opinion that the Monte-Carlo algorithm is not ideally suited to the Seven Wonders game. Compared to our Guaranteed AI, Monte-Carlo returns worse results, and takes a very long time to return them.

With better hardware, we might be able to return better results, but even if we did, it would still take around 30s/action, which adds up to a total of 9min/game. This brings us to the conclusion that in this situation, the implementation of the Monte-Carlo algorithm is not time-efficient, and isn’t adapted to the game Seven Wonders.

Conclusion

To conclude, our Guaranteed AI gives out good results. It has an average score higher than any of our other AIs (including our Ambitious one, we will touch on that later), and was simple to implement, which allowed us to refine some features that we had discussed earlier during our first meetings.

However, our Ambitious AI, using the Monte-Carlo algorithm, doesn’t have a very good overall performance. Other than simply the average score, a good metric we could use is the average score over the computation time.

The Ambitious AI we implemented takes very long, and gives acceptable performances only against DumbStrategy (an AI playing random actions). It also takes an average of 30 seconds to choose what action it should play, whereas our Guaranteed AI gives good results quite fast.

Therefore, one might think implementing Monte-Carlo was a waste of time. It is not our feeling : This implementation gives very good results if we have enough computational power in order to use it properly.

The performance of our Ambitious AI is limited by the time it takes in order to choose actions, which is in turn dictated by our machines' performances. Given the same amount of time, and using better performing hardware, our Ambitious AI should give better results because it would go to a higher depth, and it would be able to simulate more games.

Thus,we came up to the conclusion that the Monte-Carlo algorithm is not ideally adapted to the Seven Wonders game due to its time consuming aspect and to its need to be run on powerful machines.

However,implementing the Monte-Carlo algorithm was very interesting for us, and it gave us the opportunity to learn an elegant AI algorithm that we could use in the future for other projects.

Personal Reflexions & Perspectives

It is our global opinion that if we wanted to have better performance for our Ambitious AI, we should have added better rules and tactics to our Guaranteed AI in order to have even better results, instead of implementing a whole new algorithm like we did with the Monte-Carlo algorithm.

Given everything said in the conclusion, we are quite satisfied with the results we have. Maybe we should have tried to implement another algorithm, more suited to our needs, and as smart and elegant as the Monte-Carlo algorithm in order to have higher performance.

If we had more time, we should also have worked more on the returned heuristic, which maybe would have given us better results than the ones we have.

Another problem with an algorithm that takes so long is the fact that computing statistics takes a very long time, and in order to complete everything in time, we had to cut down our statistical sample from 2000 games to only 100; Having statistics with such a reduced sample can make our results less reliable, because of initial conditions (such as the content of the decks, the order of the received hands, etc…).