An implementation of the 7 Wonders board game for AI-based players

Julio Pinto Coelho Ribeiro Jardim
Department of Applied Computing
Universidade Federal de Santa Maria
Santa Maria, Brazil
jcoelho@inf.ufsm.br

Rafael Vales Bettker
Department of Applied Computing
Universidade Federal de Santa Maria
Santa Maria, Brazil
rvales@inf.ufsm.br

Pedro Probst Minini
Department of Applied Computing
Universidade Federal de Santa Maria
Santa Maria, Brazil
ppminini@inf.ufsm.br

Gabriel Gomes Pereira
Department of Applied Computing
Universidade Federal de Santa Maria
Santa Maria, Brazil
ggpereira@inf.ufsm.br

Júlia Gabriela Santi Acosta
Department of Applied Computing
Universidade Federal de Santa Maria
Santa Maria, Brazil
jgacosta@inf.ufsm.br

Joaquim Vinícius Carvalho Assunção
Department of Applied Computing
Universidade Federal de Santa Maria
Santa Maria, Brazil
joaquim@inf.ufsm.br

***Abstract*—Although we can make theoretical models and get patterns through data mining, proving a Nash equilibrium for a complex game is nearly unfeasible without a robust environment for testing. Artificial intelligence (AI) techniques rely on self-learning and benefit from massive numbers of matches. Thus, for any complex game, achieving a Nash equilibrium requires an environment that allows agents to evolve through self-play. This work describes an implementation of the board game 7 Wonders aimed to allow the creation, improvement, and testing of AI agents. To do so, we developed an efficient C++ implementation capable of running hundreds of games per minute, using simple JSON files as input and output. Thus, the input can accept any language or AI method, as long as the output can be written on a text file. Furthermore, all the step-by-step actions are recorded creating a complete log of the game, which can be used to improve the agents towards a Nash equilibrium.**

***Index Terms*—Software Engineering, Artificial Intelligence, Boardgames, 7Wonders**

I. Introduction

There are many challenges concerning the application of Artificial Intelligence (AI) techniques in games. Even with the constant breakthroughs, such as checkers [1], chess [2], go [3], poker [4], and StarCraft II [5], the state-of-the-art does not belong to a single and powerful model, but tailored and carefully implemented solutions. The intersection between these state-of-the-art solutions is the use of neural networks with reinforcement learning, preceded by an initially supervised learning step [5]. Although the results are impressive, the computational efficiency was never the main focus, since all these breakthroughs used high-end GPUs and no constraints for training time. There are not many efforts to make lightweight and efficient intelligent agents, even with the mobile world constantly growing.

In addition to the required computation, board games with stochastic and adaptive characteristics are barely targeted for research in AI (see Section II). Board games like chess are deterministic and a move on the board has specific consequences. Other board and card games are stochastic and a specific action can cause one of many consequences, each one with a predictable probability. Building an intelligent agent capable of challenging humans in such an environment is not an ordinary task. Thus, to properly implement and test an intelligent agent for 7 Wonders (see Section III), we needed a computer program that could emulate the game, taking the game inputs, processing them, and outputting the game status.

The goal of this work is to allow different implementations of AI agents, for 7 Wonders, based on machine learning or symbolic AI. Many theoretical models can be tested on datasets; however, stochastic environments such as 7 Wonders are an ongoing challenge, in which the defined rules can be used to benchmark AI techniques.

Beyond the scope of this work, our ultimate goal is to achieve a Nash equilibrium [4] for the game. A Nash equilibrium is a perfect strategy in the sense that no other strategy can have more gain in the long run. For instance, a Nash equilibrium for a Rock-Paper-Scissors game is just played one of the three with a $1/3$ probability. Anything that deviates from this is vulnerable to exploitation. E.g., if a person gets used to playing more Scissors, the opponent could exploit this behavior by increasingly playing Rock. Which, in turn, could be exploited by another playing Paper. In chess, for instance, some openings allow the player to win the game very fast, but they are extremely risky, allowing good players to counter them, consequently getting the advantage in the game. Although there is no proved Nash opening, there are a set of openings well know and frequently used by professionals.

Although we can make theoretical models and get patterns and strategies through data mining [6], proving a Nash equilibrium for a complex game is unfeasible without a test environment. To do so, we developed an efficient implementation capable of running several games per minute, the bottleneck being the decision time of the bot for deciding the next move.

Our program reads text files as input, with some being formatted as JavaScript Object Notation (JSON). The proper

formatting is shown on Tables I and II. The game outputs information that is human readable, also formatted as JSON, so the player or bot developer can keep up with the current status of the game. The implemented game generates detailed data about a hand-by-hand match, this time as a CSV file, which can later be used to fine-tune the AI algorithm.

In Section IV we show the architecture and how it can be used to train and test AI agents, along with a demonstration of how to use an AI in the game (Section V).

II. Related Work

Rubin *et al.* [7] used an upper confidence tree (UCT) search to develop a bot for Settlers of Catan, optimizing it to improve its victory rate, without relaxing the rules of the game. The UCT-based implementation was able to bargain and exchange resources with other players.

Robilliard *et al.,* [8] wrote about the Monte Carlo tree search (MCTS) algorithm for creating an AI for the 7 Wonders game. They implemented using a MCTS with susceptible levels, in which the nodes correspond to the possibilities of plays. A second AI was implemented deterministic, using fixed rules. In the end, they compared the two AIs, showing that the first had better results.

In this work, we developed an implementation of 7 Wonders. The software uses a simple file reading as input; allowing the use of bots, created with different AI techniques, and from any programming language capable of writing text files. We also used a rule-based system to develop and test the bot, placing three bot instances to play the game against each other, similarly to [7] and [8]. However, unlike [7] that compared the win rate with another work, our work has the goal to be an environment for testing AI techniques. Thus, the first agent developed has the only purpose of illustrating the use of agents in the game.

III. 7 Wonders overview

7 Wonders is one of the most awarded boardgames, played from 2 to 7 players, where each player receives a board representing one of the seven wonders of the ancient world. The game is split into three ages, which have cards that are divided into seven types: civil, scientific, commercial and military structures, raw materials, manufactured goods, and guilds. Fig. 1 displays the board of one player from our game implementation.

Civil, scientific, and guild cards generate victory points (VPs), commercial cards provide coins or advantages in purchases in the commerce, and military cards grant shields for conflicts. Furthermore, cards related to raw materials and manufactured goods generate the necessary resources for the construction of the structures initially mentioned.

At the end of each age, battles occur between adjacent players, winning the one who possesses more shields. For each conflict won at the end of the first, second, and third ages; one, three, and five VPs are gained, respectively. A lost battle is penalized with the loss of one VP.

Fig. 1. A screenshot, from our implementation, that shows a player's board, buttons for choosing the action, cards in hand, cards played, resources, and the victory points.

At the end of the third age, the VPs of each player are counted and whoever has more points wins the match. Detailed rules can be found in the game manual[1].

IV. Implementation architecture

A. Game simulation

To test the bots, there was a need for an application that can yield results of the simulation. By receiving commands in the JSON format – which can be easily utilized by bots –, our implementation can accurately simulate a full match. More specific actions (e.g. choosing which resource to produce for cards that can produce one or the other) were automated to reduce the quantity of information passed through the JSON file, maintaining it as simple as possible.

B. Class Descriptions

Explained below are all the classes of the game, which can be visualized in Fig. 2.

1) `Game`: The main class of the program, responsible for managing the flow of the game by connecting every other class and their methods. Specifically, the `Game` class manages:

- Game start and end;
- Dealing of cards and passage of turns and ages;
- Input commands (e.g. `build_structure`, `discard`, etc.);
- Recording the game status.

2) `Player`: Mainly responsible for managing all the actions that may be performed by the player and also keeping track of their resources. Examples include:

- Battles and resource management;
- Building Wonder stages and structures;
- Applying Wonder effects.

3) `Card`: Contains all the information regarding 7 Wonders cards, such as the required resources to play them, chaining information, specific age where the card is available, name, and so forth.

[1]https://rprod.com/uploads/file/7WONDERS_RULES_US_COLOR.pdf

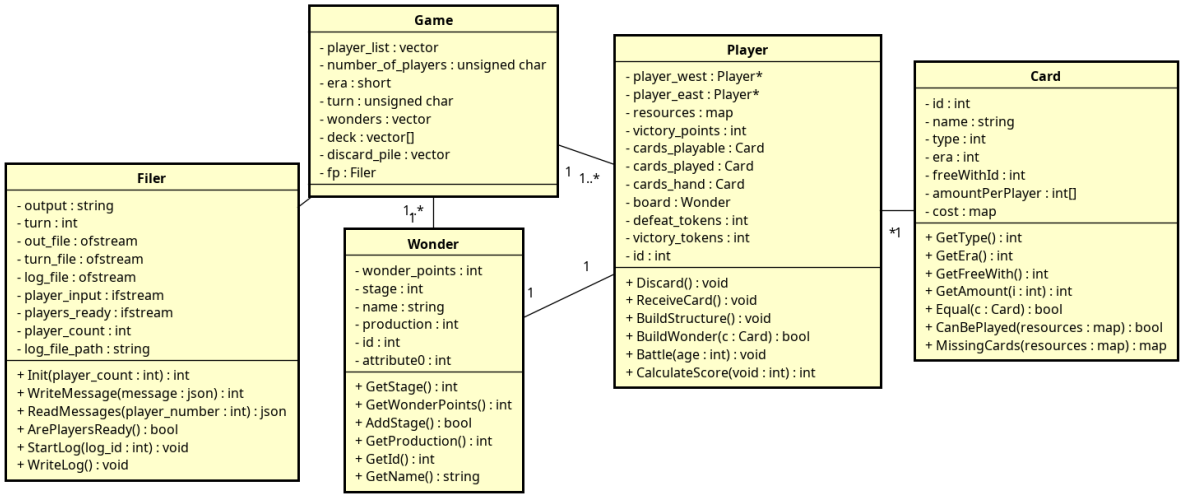

Fig. 2. Class diagram showing the project main methods

4) `Wonder`: Describes all the boards available in the game. Specifically, there is a main `Wonder` class from which all the particular subclasses (e.g. `Olympia_a`, `Babylon_b`, etc.) derive by inheritance. Each subclass contains basic information, such as the current built stage, initial production, resources required for the next stage, and so forth.

5) `Filer`: Manages the input and output of the program. It is responsible for:

- Reading the `ready.txt` file to verify if each player acted in the current turn;
- Reading each `player_i.json` file, which contains the current play of the `i` player;
- Writing the game log.

In summary, the methods contained in `Filer` are the channel of communication between the bots and the game. `Filer` is directly managed by the `Main` class.

C. Input-Output

JSON format files are used for input and output. Files named `player_i.json` are meant for players to write the commands. They also must append a single “ready" to a file named `ready.txt`, so the game will check if everyone is ready. The game handles deleting the contents of `ready.txt`, so the players must only overwrite `player_i.json`. The input consists of a command file for each player, containing two arguments:

- A command, which may be to build a structure, a stage of wonder, discard or special actions;
- A card name.

As a result of each round, an output file called `game_status.json` is generated containing the current state of the game with updated information for each player, such as the cards in hand, cards that can be played, cards already played, resources, partial victory points, stages of wonder built, etc. An error file can also be generated if the player has written an invalid command or one which cannot be performed by any game rules, such as lack of resources. At the end of a match, the output is a file with the final results of the game.

Table I and Table II show the structure of the JSON file, as well as possible game commands.

TABLE I
JSON INDEXES

Index	Meaning
`command`	Top-level JSON object.
`subcommand`	Command to be executed.
`argument`	Card name to be used by `subcommand`.
`extra`	Card name (when playing a discarded card).

TABLE II
VALID SUBCOMMANDS

Subcommand	Action
`build_structure`	Tries to play a card if there are enough resources.
`build_hand_free`	Tries to play a card for free (Olympia A-2).
`build_wonder`	Tries to build the next stage of the Wonder.
`discard`	Discards the selected card for coins.

Except for Olympia A second stage’s effect (`build_hand_free`), every other Wonder special is called automatically. Note that `extra` in Table I will only be used by Halikarnassus players, as it is possible to play two cards in the same turn right after certain stages are constructed.

V. BOT DEVELOPMENT

As the main purpose of this work is to facilitate the creation of AI-based players; we create an input-output interface as simplified as possible. The only knowledge required is the operations (for game actions) and its files (only two); one file to read, and one file to write. Thus, any programming language capable of reading and writing text files can be used.

A simple bot can be created just by reading the available cards, selecting a random buildable card, and writing

the command in the `player_i.json` file. Supposing the `game_output.json` file has the following text:

```
"players":{ 0:{
        ...
        "cards_hand":[
            "Marketplace", ...
        ]
}}
```

The bot could choose to try building *Marketplace*, by writing to the corresponding `player_i.json file`, such as:

```
"command":{
    "subcommand": "build_structure",
    "argument": "Marketplace",
    "extra":""
}
```

And then appending a single *ready* to the `ready.txt` file, marking the play as finished.

VI. Rule-Based AI

To test the software performance, as well as to create data for training future bots, we created a basic agent-based on previously mined rules [6] and a guide available in the Board Game Arena website[2]. The bot changes weights mainly based on the current Wonder, board, and table configuration. Table III shows an example with simple conditions for assigning weights to each card.

TABLE III
Example rules

Card name	Condition	Weight if True	Weight if False
Stone Pit	You have less than 2 raw materials	3	1
East Trading Post	You do not have raw material and your east neighbor has	4	1
Craftmen's Guild	Neighbors have more than 3 manufactured goods	4	1
...	...	...	...
Palace		5	

The action of building a card receives a weight relative to the available rules. If there is more than one card option with the highest weight, a random choice is made between them. On each round, the agent observes the changes in the `game_status.json` file, which has game-state information. When a new round begins, the model checks which cards are available to be played and gets the weight of each one from the predefined rules.

VII. Results

To evaluate the implementation, we performed two different tests[3]. The first test was aimed to calculate the implementation time, simulating an agent without spending time deciding the action. To do so, the bot applies the first possible action: build the wonder stage, build a structure or discard (if there is no playable card). This experiment retrieved an average time of 70 milliseconds per match (850 matches per minute). For the second test, we used the AI agent described in Section VI. This agent retrieved an average time of 180 milliseconds per game (330 games per minute).

Also, the experiments generated log files containing the hand-by-hand matches – cards played, resources, cards in hand, card and action chosen by each player for each turn – that can be used as an initial supervised step to feed AI algorithms that require a large amount of data to be trained efficiently.

VIII. Final Remarks

We developed an efficient implementation[4] of 7 Wonders capable of running hundreds of games per minute reading ordinary JSON files as input (Section IV-A). Although the file-based input and output affect the performance, it makes it easier for developers to create their own agents. Furthermore, the implemented game generates detailed data about a hand-by-hand match, which can be used to train and improve AI models.

We showed the basic details of the architecture, so other developers may improve the game and create their own AI (Section IV-B). Furthermore, we created a basic rule-based AI which generates data to be fed into a machine learning model. As a work in progress, our final goal is to deliver an AI capable of challenge top-level players. This AI will be closer to a Nash equilibrium, based on hundreds of game statistics and winning patterns [6], and fed with thousands of 7 Wonders matches, extracted from this work.

References

[1] J. Schaeffer, *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer New York, 2013.

[2] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[3] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 01 2016.

[4] N. Brown and T. Sandholm, "Superhuman ai for multiplayer poker," *Science*, vol. 365, no. 6456, pp. 885–890, 2019.

[5] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. P. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, "Starcraft II: A new challenge for reinforcement learning," *CoRR*, vol. abs/1708.04782, 2017.

[6] J. Assunção, G. Pereira, J. Acosta, R. Vales, and L. Rossato, "Data mining 7 wonders, the board game," in *Proceedings of SBGames 2019*, pp. 583–586, 2019.

[7] G. Rubin, B. Paz, and F. Meneguzzi, "Optimizing uct for settlers of catan," in *Proceedings of SBGames 2017*, pp. 221–227, IEEE, 2017.

[8] D. Robilliard, C. Fonlupt, and F. Teytaud, "Monte-carlo tree search for the game of "7 wonders"," in *Workshop on Computer Games*, pp. 64–77, Springer, 2014.

[2] https://en.boardgamearena.com/forum/viewtopic.php?f=192&t=14557

[3] The experiments were performed with 1000 runs on an Intel Core i5-7200U 2.50 GHz processor with 8 GB DDR4 RAM.

[4] The source code is publicly available at https://github.com/dmag-ufsm/7Wonders