PokéBOT, a Pokémon battle bot

Dalla Noce Niko, Lombardi Giuseppe, Ristori Alessandro
Master Degree in Computer science.
{n.dallanoce, g.lombardi11, a.ristori5}@studenti.unipi.it
Artificial intelligence fundamentals, Academic Year: 2022/2023
`https://github.com/nikodallanoce/PokeBOT`

January 3, 2023

1 Introduction

In the AI field, games have always been given a soft spot since they are ideal as first tests for AI algorithms and they provide a variety of environments in which rules are well-defined and can be exploited in order to understand the complexity of such environments and how to apply the same reasoning on harder research fields. For such reason, and for giving nostalgia another ride, we decided to focus our work on *Pokémon* battles.

Pokémon battles are an interesting *multi-agent non-cooperative* environment in which two players have six *Pokémon* each and their goal is to defeat all the opponent's *Pokémon*. Battles are based on atomic turns that are defined as the two moves chosen by the players and resolved simultaneously, such moves can fail given a probability called *accuracy* and their *secondary effects* have an impact on the next turns, for this reason the environment under consideration is also *partially deterministic* and *not episodic*. Moreover, both players do not have perfect knowledge of the environment and they can use this fact in order to gain an advantage over the opponent by switching out the active *Pokémon* or revealing a move or item that was not shown before.

Our aim is not to build the perfect *Pokémon* battle bot (which is near to impossible given the stochasticity of this environment), but to build one that can stand its own against human players.

2 Related works

All of our players were thought to be developed in order to run on an online open-source *Pokémon* battle simulator [6] which was also vital for the performance's evaluation of our bot.

For that reason, our work was mainly inspired by [1], starting from that we looked for more implementations in order to give ourselves an overall idea about what to do. During our research we found the poke-env library [5] which gave us a very useful engine and some *baseline* players that were essential for testing ours; the strongest baseline player of the library was the main metric of comparison of our work.

The choice of players was, instead, based on implementations found on *Github*, where we have found a small, but working, example of rule-based and minimax players [4] and a bot that is considered among the best ones in the online community [3]. The former is built on the poke-env library [5], so we used it for defining the structure of our players, while the latter gave us ideas on how to improve the stats and damage computations.

3 Methodologies

Just like *Pokémon* evolve upon reaching certain conditions, we applied the same reasoning for the development of our bot. First of all we started with a simple player (called *MaxBasePower* or *MBP*) that chooses the non-status move with the highest base power, regardless of the actual damage dealt. Then, we upgraded this first simple player by letting it choose the move with the highest actual damage thanks to a hand-crafted stats and damage computations (we called this player as *BestDamage* or *BD*).

The damage computation is the main engine of our bot, since the more accurate is the damage dealt the higher the performance of the players.

$$damage = \left(\frac{(\frac{2 \times level}{5} + 2) \times power \times A/D}{50} + 2 \right) \times targets \times PB \times weather \times critical \times random \times STAB \times type \times burn \times other \quad (3.1)$$

As we can see from (3.1), the damage computation is made up of many parameters and each one of them contains several rules that can be explained in FOL, an example is shown in (3.2).

$$\forall x \; \textit{Pokémon}(x) \wedge \forall y \; Move(y) \wedge Holds(x, airbaloon) \wedge MoveType(y, Ground) \Rightarrow Immune(x, y) \quad (3.2)$$

After finishing the engine, we had to deal with switches; the main issue with them is that there is not an optimal number and the situations in which we should use this mechanic can not be hard-coded since they would be too many. To alleviate all of this, we decided to build what we have called a *matchup* function that, given the two *Pokémon* , computes the advantage (or disadvantage) the bot has. The computation of the *matchup* value for each not fainted bot's *Pokémon* is based on the damage multipliers coming from their types and moves:

1. First we take the max damage multiplier based on types for both the bot's and opponent's *Pokémon* as shown in Figure 3.1.
2. Then, we compute the difference between the two values obtaining what we called the *type advantage*.
3. We do the same for the moves, in case the opponent's *Pokémon* has no known moves we suppose that it will have at least one move for each one of its types and we assign it a default move.
4. Just like we did for the types we do the same difference to compute the *move advantage*
5. At the end, we sum both values to obtain the actual *matchup* $\in [-8, 8]$ which indicates how favourable is the current situation for the bot's *Pokémon* against the opponent's.

Figure 3.1: *Damage multipliers for each type*

3.1 Rule-Based Player

The first enhanced version of our bot is the rule-based player (or *RB*) which uses everything we talked about until now plus some more improvements thanks to an external expert. We defined even more rules to deal with *healing*, *status*, *boost*, *protective* and *retaliatory* moves, while also implementing a strategy for deciding when to switch out and under which situations we should use the gimmick, which is dynamax for the generation our bot will play in.

3.2 MiniMax Player

The second enhanced player we developed uses the minimax algorithm with α-β pruning in order to choose the best action to take (we called such player as *MiniMax* or *MM*). Generally, a player can choose either to make a move or to switch out the active *Pokémon*, for this reason, the mini-max tree has a branching factor that is the sum of the number of available moves of the active *Pokémon* and the number of available switches; the latter are, usually, the number of not fainted *Pokémon* in a team.

The root node of a tree represents the actual current state of a battle that we want to start simulating, in which we assume the bot to take an action first i.e. before the opponent player. The depth of the tree is two time the turns we want to simulate because in a *Pokémon* battle, a turn ends when both the players have chosen an action. In this way, assuming t is the turn we want to simulate, the effects of our actions are computed at each $2t + 1$ depth level, while the opponent ones at every $2t$, with $t > 0$, depth level. Taking into account everything we have seen until now we can say that each one of the nodes simulates the progress of a battle, as an example, see Figure A.1.

As we first said in section 1, a *Pokémon* battle is a partially observable environment in which the opponent's team, *Pokémon* statistics, abilities and moves can not be known a priori and each move has its own probability to hit. As a consequence, the library [5] does not allow to simulate the progress of a battle, otherwise it would have been like cheating. To overcome these limitations, at the beginning of each battle, we build a knowledge base that is composed by the *Pokémon* that belong to our team and all their attributes plus the opponent's active *Pokémon*. Whenever the opponent player chooses a move or switches to a *Pokémon*, we add these information to the knowledge base in order to simulate the progress of the battle more accurately as possible. It is not possible to consider all the variables of a battle, so the most relevant ones we considered are: the weather conditions, the moves, statistics and boosts of both active *Pokémon* and finally, the composition of our team and the opponent's. All these variables may change from turn to turn assuming a crucial role when computing the damage, the recoil or the drain of a move. Then, in order to know whether or not we are at an advantage over the opponent, we developed an heuristic.

3.3 Heuristic

To estimate the utility of a node, we use an heuristic that uses knowledge of the current state to tell us how favourable is a particular battle progress.

Initially, we used a straightforward evaluation function that uses only the knowledge of both active *Pokémon* : we considered the ratio of the current hp (health

points) of the *Pokémon* over its max hp (3.3).

$$f(node) = \bar{hp}(active_{Bot}) - \bar{hp}(active_{Opp}) \quad (3.3)$$

The minimax algorithm with such function has some limitations since it has no global knowledge about the state of the game: as an example, the opponent's active *Pokémon* might be defeated, but all the remaining *Pokémon* of his team are alive; in this case, the rating function gives a very positive score even if our team has only one not fainted *Pokémon* .

Considering these limits, we decided to use a more reliable function that considers the ratio between the hp of the entire team and the number of surviving *Pokémon* . Finally, we added a penalty term on the depth of the node to the evaluation function [1]; this penalty promotes exploration among all possible strategies and discourages excessive turn depth. The resulting evaluation function is a linear weighted sum (3.4) where the weight of each term indicates how relevant the term is to the state evaluation. By defying $alive(team) = num\ of$ Pokémon $alive\ in\ the\ team$, the evaluation function can be written as:

$$\begin{aligned} f(node) = {} & w_1 \cdot \bar{hp}(team_{Bot}) + w_2 \cdot alive\,(team_{Bot}) \\ & - w_3 \cdot \bar{hp}\,(team_{Opp}) - w_4 \cdot alive\,(team_{Opp}) \\ & - w_5 \cdot depth(node) \end{aligned} \quad (3.4)$$

We performed a random search on the weights to find the best configuration of the function and evaluated each configuration by observing the minimax player's performances. The best-resulting weights configuration favours dealing damage to the opposing Pokemon rather than rewarding survival and associates a small depth penalty.

4 Assessment

In order to evaluate the strength of our different player strategies, we first ran them in a local server, a feature implemented in [5], that simulates a player vs player battle. It did not work out of the box due to some incompatibilities between the Node JS version and the Python one, so, in order to solve this problem and to improve reproducibility, we wrote a Docker [2] file that is available in the root of the GitHub repository. The poke-env library [5] implements a "baseline" player that has an Elo rank of 1100-1200 points on the remote server [1]. So, our aim was to develop at least two players, a rule-based (subsection 3.1) and a minimax (subsection 3.2), that could defeat it. Then, we made them battle against each other a thousand of times to assess their strength. At this point, it faced out that the mini-max player made too many *Pokémon* switches, so we developed a new strategy in which we exclude them from the mini-max tree computation. In Figure A.2 we can see the differences with respect to the approach described in subsection 3.2. In this way, the player chooses whether or not switch a *Pokémon* at the beginning of each turn, only based on the matchup score, just like the rule-based player does.

So, going back to performances between each player, we can declare, by looking at Table 1, the rule-based and minimax players to be the best ones as expected.

Player	MBP	BD	RB	MM
MBP	/	0.104	0.084	0.114
BD	0.896	/	0.398	0.452
RB	0.916	0.602	/	0.571
MM	0.886	0.548	0.429	/

Table 1: *Percentage of win for every player against each other*

We have also noticed some more things: first of all using the matchup function for choosing the best switch increased the performance of about 5% (with a proper switch strategy of course) against the baseline player from [5], while using the gimmick strategy had the major effect with a 8-9% increase in performance.

As a final test, we run our two best players for 100 matches on the remote server [6] against human players on generation 8 random battles. We can look at their ELO progression in Figure 4.1. It turns out that

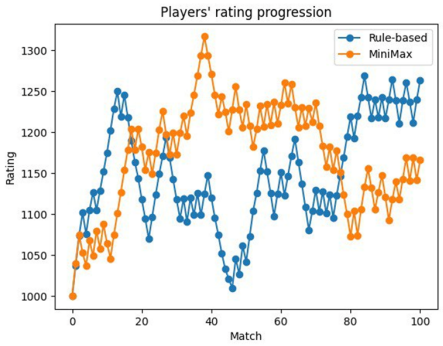

Figure 4.1: *ELO progression of RB and MM players against human players*

the two players float around the 1200 ELO points, making them somewhat at the level of an average skilled human player.

5 Conclusion

Building a *Pokémon* battle bot proved to be an exciting and motivating challenge and this project allowed us to apply some artificial intelligence methods, especially those regarding game theory and adversarial agents. We overcame many difficulties while "evolving" our bot and its gradual refinement, that ended up in two different but near equally strong players, has resulted in an artificial "trainer" with the strength and behaviour of an average skilled human player, which was our initial aim.

References

[1] Scott Lee and Julian Togelius. Showdown ai competition. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 191–198, 2017.

[2] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[3] pmariglia. A pokemon showdown battle bot written in python. `https://github.com/pmariglia/showdown`, 2018.

[4] RemptonGames. Pokemon ai battlebot. `https://github.com/RemptonGames/Pokemon-Showdown-Agent`, 2020.

[5] Haris Sahovic. The pokemon showdown python environment. `https://github.com/hsahovic/poke-env`, 2019.

[6] Pokémon Showdown. Pokémon showdown! battle simulator. `https://play.pokemonshowdown.com/`. Accessed: 2022-12-15.

A Appendix

A.1 Team contributions

- **Niko:** setup of local server, minimax player, α-β pruning and heuristics.
- **Giuseppe:** gimmick rules, α-β pruning, heuristics and scripts for running the players.
- **Alessandro:** stats and damage computation, baseline players, matchup rules and rule-based player.

A.2 Github metrics

Member	Commits	Major activity period
Niko	55	middle of november
Giuseppe	37	middle of november
Alessandro	57	start of november

Table 2: *Github contributions for each member*

A.3 Relationship with the course

- Study and definition of the problem:
  - Definition of the environment and its characteristics.
  - Definition of the type of game.
  - Acting under uncertainty.
- Definition of the agents:
  - Logical agent (Rule-based)
    - Definition of rules in FOL.
  - MiniMax algorithm
    - α-β pruning.
    - Definition of the Heuristic function.
- Exploit non-cooperative Game Theory:
  - Weak dominant strategy for type matchup.

A.4 Mini-max tree

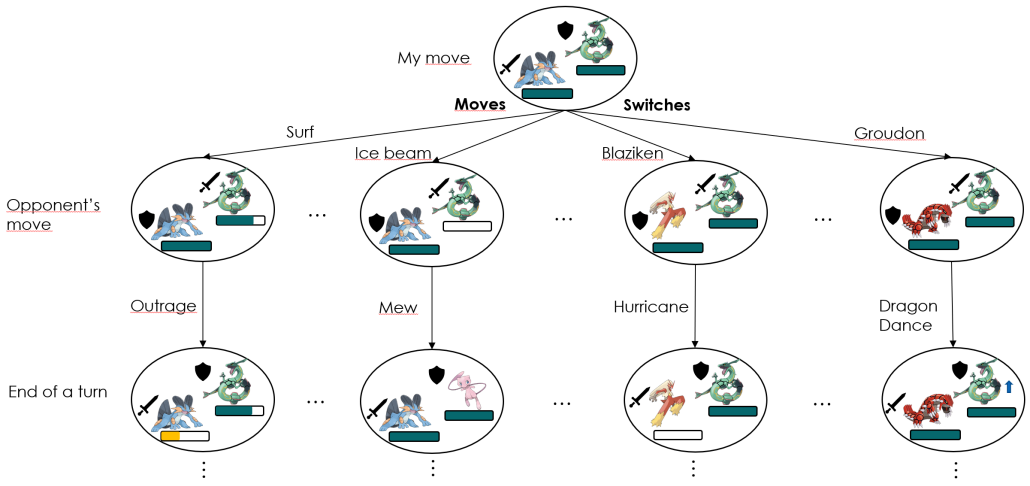

Figure A.1: *Minimax tree with switches.*

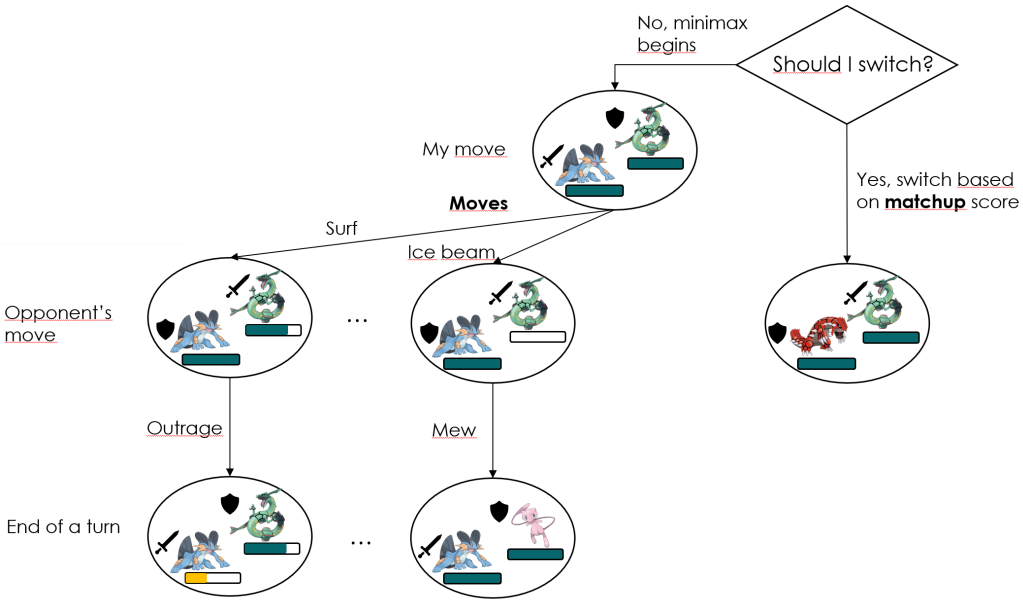

Figure A.2: *Minimax tree without switches.*