Agent battle track

Russo Giuseppe Gabriele (583744)
Panzani Gianluca (550358)
Mogavero Mario (636852)

Master Degree Computer Science
Artificial intelligence Fundamentals 2024-2025

Contents

1 Introduction

The realm of Pokémon battles, a game cherished for its strategic depth and intricate complexity, offers a compelling environment to advance artificial intelligence (AI) research. This paper outlines the development and evaluation of several AI agents designed to compete in 3v3 Pokémon battles within the **AI Framework of VGC** (Video Game Championships), a specialized platform built to simulate the official battle environment sanctioned by The Pokémon Company. This project aims not only to push the boundaries of Pokémon AI capabilities, but also to provide valuable insights into AI development in strategic, complex game domains.

Our approach was multifaceted, beginning with the creation of a **suite of baseline AI agents**, each embodying different algorithmic philosophies for battle decision making. These agents were rigorously tested against a randomized control agent to establish a performance baseline, and performance was evaluated using metrics based on win rate and residual HP. Subsequently, the agents were pitted against each other in a round-robin format, allowing a comprehensive comparative analysis of their strengths, weaknesses, and strategic nuances. On the basis of these findings, we synthesized a hybrid agent, merging the most effective elements of its predecessors. The hybrid agent was then subjected to a further series of tests against the original agents to assess its performance and adaptability.

The development of these agents is underpinned by a deep understanding of fundamental Pokémon mechanics, including type matchups, stat calculations, move selection, and ability effects. Moreover, the theoretical framework encompasses the design of state evaluation functions for accurate game state assessment, efficient move selection algorithms to navigate the game's vast decision tree, and opponent modeling techniques to anticipate and counter adversarial strategies. The integration of reinforcement learning methodologies is also explored to endow the agents with the capacity for adaptation and experiential learning. This report will provide a detailed exposition of each agent's architecture, the results of their comparative performance, and the synergistic analysis that led to the creation of the hybrid agent, along with its performance evaluation.

The project is available on the GitHub repository on this link.

2 Agents

Before we move on to talk about the various agents, it is necessary to explain how the `BattlePolicy` class is implemented in the VGC ecosystem because all agents are classes inherited from it. The class includes two main methods:

- `__init__` : This method is the constructor of the class and is used to initialize the parameters needed for the action choice policy to work.
- `get_action` : This is the heart of the class. It is responsible for choosing an action based on the current state of the game (`gamestate`). In fact, the method takes as input the `gamestate` and returns an integer representing the index of the chosen action. The possible actions are contained in a list of 6 elements where the first 4 elements represent the 4 moves of the active Pokémon and the last 2 elements represent the switches, that is, the change with one of the Pokémon on the bench. So, the method returns the position of the index in this list.

Due to the amount of variables used in these algorithms, we performed a *Grid Search* to find the values with which it performs better. A `.env` file, with the following syntax, contains the parameters' values of which the algorithm generates all the possible combinations. For every combination, `N_BATTLES` battles (the only mandatory parameter) are performed on which the win rate is computed and, after this, the results are appended on a `.csv` file with the parameters' values too.

```
[file.env]
KEYS=param_1,...,param_n
param_1=val_1,...,val_{m_1}
...
param_n=val_1,...,val_{m_n}
```

In this way, we found the best parameters for the Montecarlo Tree Search and MiniMax algorithms, which are specified respectively into the `MCTS.env` and `MiniMax.env` file. The results with this parameters' combination are discussed in the section 3.

2.1 Random

The first agent we present is the **Random Agent**. This agent selects an action to execute based on a predefined **probability distribution**, without any in-depth analysis of the context or the `gamestate`. In other words, its decisions are governed purely by probabilistic logic, making it a non-strategic approach. The primary goal of implementing a random agent is to serve as a baseline for comparing the performance of more complex and advanced agents. Using a simple and standard approach, it becomes possible to assess the added value of sophisticated decision-making strategies compared to a balanced random choice.

The probability distribution used by the random agent is defined as:

$$p_i = \underbrace{\left[\frac{1 - \text{switch\_probability}}{\text{n\_moves}}\right]}_{\text{Probability of moves}} \times \text{n\_moves} + \underbrace{\left[\frac{\text{switch\_probability}}{\text{n\_switch}}\right]}_{\text{Probability of switches}} \times \text{n\_switch}$$

This formula generates a list of probabilities with:

- First 4 elements correspond to the probabilities of the active Pokémon's moves. Each move receives a uniform probability calculated as:
$$\frac{1 - \text{switch\_probability}}{\text{n\_moves}}$$
, where `switch_probability` is the constant governing the agent's inclination to perform a switch.
- Last 2 elements correspond to the probabilities of switching Pokémon. Each switch receives a uniform probability calculated as:
$$\frac{\text{switch\_probability}}{\text{n\_switch}}$$
.

2.2 Logic

The first agent that will be compared with the random agents is the **Logic agent**. This agent is designed to make decisions based on a **logical inference system**. When initialized, the logic agent possesses a `KnowledgeBase`, an instance of the corresponding class. The **Knowledge Base** is the central component that allows the agent to reason in a structured way, leveraging facts, rules, and priorities to determine the best action to take.

The class `KnowledgeBase` is structured like this:

1. **Facts**:
   - Facts are stored in a dictionary within the `KnowledgeBase` and represent information about the current `gamestate`.
   - These facts are dynamically updated based on the current gamestate.
   - Examples of facts include: the type of the active Pokémon, the type of the opponent's Pokémon, the types of available moves, weather conditions, and other relevant details.
2. **Rules**
   - Rules are methods defined in the `KnowledgeBase` class. Examples of rules include:
     - `self.rule_super_effective`: evaluates whether a move is super effective against the opponent's Pokémon.
     - `self.rule_stab`: considers the STAB (Same-Type Attack Bonus) for moves.

  - `self.rule_weather`: evaluates how weather conditions affect the moves.
- Rules do not directly return a value but are invoked by the inference engine, updating an internal list called `actions_priority`.

3. **Actions Priority**:
   - This is a list that contains priority values for each of the 6 possible actions (4 moves and 2 switches).
   - Each rule contributes to updating the values in this list based on the current facts, increasing the priorities to the most advantageous actions.

So, the **inference engine** uses the facts and invokes the rules in the KnowledgeBase to iteratively update the `actions_priority` list. Once the process is complete, the resulting list represents the priorities for the 6 available actions. The KnowledgeBase's `get_actions_priority` method returns this list, and the logic agent uses the `get_action` method to determine the best action. The chosen action is the one with the highest priority value in the `actions_priority` list.

2.3 MiniMax

The second agent developed is MiniMax. It operates by exploring the game tree to a predetermined depth, evaluating potential future game states to make the most advantageous move at each turn. The strength of this agent lies in its carefully crafted heuristic function, which is used to assess the desirability of each node (representing a partial game state) within the search tree.

The heuristic function of the MiniMax agent is a composite metric that encapsulates several crucial aspects of a Pokémon battle, each weighted to reflect its strategic importance. The primary components of this heuristic are as follows.

1. **Winning Condition:** The most significant factor is whether a given game state represents a win or loss for the agent. This binary outcome is the ultimate goal and is given the highest priority.

2. **Residual HP:** The amount of remaining health points (HP) for both the agent's and opponent's Pokémon is a critical factor. Higher residual HP for the agent's Pokémon and lower HP for the opponent's are favored.

3. **Number of Pokémon Alive:** Maintaining a numerical advantage in terms of the number of Pokémon remaining in the battle significantly improves the chances of victory. The heuristic prioritizes states where the agent has more Pokémon available than the opponent.

4. **Status Conditions:** Status conditions such as burns, paralysis, sleep, and poisoning can drastically alter the course of a battle. The heuristic accounts for these conditions, favoring states where the opponent's Pokémon are afflicted with detrimental statuses, and the agent's Pokémon are not.

5. **Weather Conditions:** Weather effects such as Rain, Sun, Sandstorm, and Hail can significantly influence the effectiveness of certain moves or Pokémon types. The heuristic considers the current weather condition and its potential impact on both the agent's and opponent's teams.

6. **Type Effectiveness:** A core mechanic in Pokémon is the concept of type matchups. The heuristic incorporates the advantage of super-effective moves, giving higher scores to nodes where the agent can exploit type weaknesses in the opponent's team.

For each turn, the MiniMax agent evaluates all possible actions, which include choosing between the four possible attacks for the active Pokémon or switching to one of the two Pokémon on the bench. The agent uses the MiniMax algorithm, combined with the aforementioned heuristic, to evaluate the potential outcomes of each action. Critically, switching a Pokémon is not without its drawbacks. To accurately reflect the strategic cost of switching, a penalty term is incorporated into the heuristic evaluation for any action that involves switching. This penalty takes into account the potential loss of time and the opportunity cost of not using an attack that turn.

Another crucial consideration in the implementation of the algorithm is the inherent turn structure of Pokémon battles. Each turn comprises one action by our agent followed by one action by the opponent. To address this, a **partial state simulation** function was incorporated into the MiniMax algorithm. This function simulates the outcome of the agent's chosen move and then returns the relative partial gamestate. The MiniMax algorithm then continues its evaluation from this partial state, effectively extending the search horizon by one ply (half a turn).

Furthermore, to manage the computational complexity, we introduced a relaxation of the problem. In a fully accurate simulation, factors such as move priority and the possibility of moves missing would need to be considered. This would further explode the branching factor, rendering the search intractable. Therefore, our MiniMax implementation assumes that **both players always hit their moves and that our agent always acts first** within a turn. While this simplification deviates from the true game mechanics, it allows for a more efficient search while still retaining a good approximation of strategic decision-making.

The MiniMax algorithm can then determine the optimal sequence of moves to guarantee victory, if one exists, or to make the most advantageous decisions in complex endgame situations, given this simplified turn structure. To verify the correctness of the tree structure, we have implemented a visualization of the tree using the Python package `pyvis` (as described in the 2.4 section).

2.4 Montecarlo

The third agent is based on a "no Pure" version of the **Montecarlo Tree Search (MCTS) algorithm** because of the introduction of some heuristics.

The MCTS is divided in 4 phases which we have implemented as follows:

- **Selection phase**: the algorithm follows a path in the tree until a leaf is found, like the pure version. Each child's selection step is based on the *Upper Confidence bound applied to Tree (UCT)* with the *UCB1 formula*.
- **Expansion phase**: due to the huge value of *branching factor*, which amounts to 36 because of the 6 possible moves of each player, we used the `KnowledgeBase` to reduce the number of children expanded. The algorithm chooses the best moves for itself and the best ones for the opponent (to consider the worst cases), and then generates all the possible combinations of moves. For each combination, a child for the leaf selected on the previous phase is generated.
- **Simulation phase**: this phase consists in generating a path until a termination node (or an end-game state) is found, from each expanded child.
- **Backpropagation phase**: for each termination node, the result of the battle is backpropagated on the previous nodes until the root is found. On each backpropagation step, the current node's utility value is updated based on that result.

After the described phases, the algorithm has to choose one of the children of the root as the move to be returned by the `get_action` method. This choice is based on the *utility value* of the children and on a *heuristic function* which has the goal to choose in a smart way the best next node in case of nodes with similar utility values. This function has the following conditions (which are inserted in the `for` which iterates on the children):

- *Switch condition*: we have empirically observed that a higher number of switches causes a higher number of lost matches, although the switch moves had a higher utility value. So we have introduced a condition which, in case of similar utility and current node with switch move, chooses the non-switch one.

$$|utility(best\_node) - utility(current\_node)| < \underline{treshold_1} \quad \text{AND} \quad is\_switch(current\_node.move) \tag{1}$$

- *Total playouts condition*: the aim of this condition is to choose the node with the higher number of total playouts instead of the utility value itself.

$$|utility(best\_node) - utility(current\_node)| < \underline{treshold_2} \quad \text{AND}$$
$$total\_playouts(current\_node) > total\_playouts(best\_node) \cdot \underline{value\_higher\_than\_1} \tag{2}$$

We observed that the agent, with this heuristic function, has a win rate gain of even $\sim 5\%$ against the Random agent.

The last topic to be addressed is the validation of the behaviour of the algorithm. The complex dynamic behaviour of the data structure brought us to search a visualization method with which we could see if the algorithm did what it was supposed to do. This was done using the python library `pyvis`, which premises to create an HTML file with a customizable graph visualization of the tree.

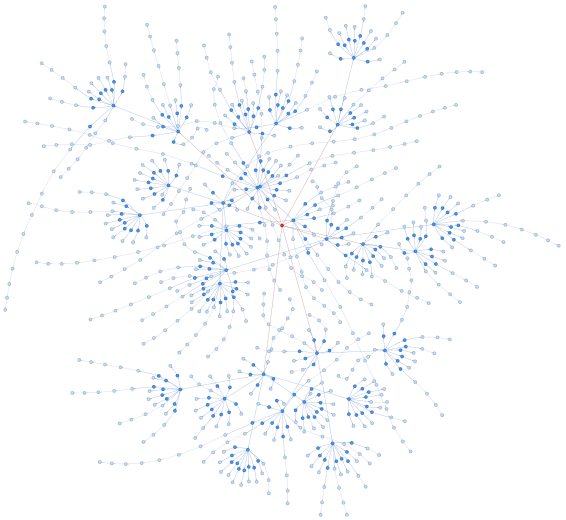

Figure 1: Higher view of the tree. The nodes of the tree are blue excpet for: the red one that is the root and the light blue ones that are generated by the simulations.

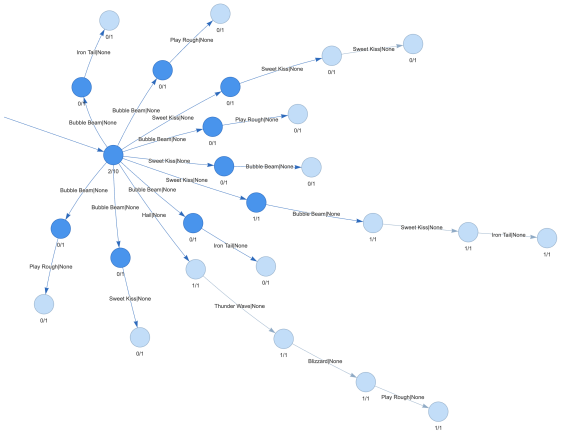

Figure 2: Zoomed view of the tree. On each edge there are the moves combination which premises the state transition. On each node there is the utility value.

2.5 Combined

Recognizing the strengths and limitations of different algorithmic approaches, we developed a hybrid agent that synergistically combines the Monte Carlo Tree Search (MCTS) and MiniMax algorithms. This agent aims to leverage the exploration capabilities of MCTS in the early stages of the game, where the branching factor is high, and the precise evaluation power of MiniMax in the endgame, where a focused search is crucial.

2.5.1 Early Game: Monte Carlo Tree Search with Logical Move Pruning

For the initial turns of a battle, the hybrid agent employs the MCTS algorithm. MCTS excels at exploring large search spaces by strategically balancing exploration and exploitation. However, the Pure MCTS approach can be computationally expensive in a game like Pokémon, which has a substantial branching factor due to the numerous move and switch options available each turn.

As mentioned in the section on the Monte Carlo agent, we combined the Knowledge Base and the MCTS algorithm to reduce the branching factor and thus the computational cost.

2.5.2 Endgame: MiniMax Precision

As the battle progresses and the number of remaining Pokémon dwindles, the game state becomes increasingly tactical. When a situation arises where both the agent and the opponent have only one Pokémon remaining

each, the hybrid agent transitions from the MCTS approach to the MiniMax algorithm.

This transition is motivated by the fact that the reduced game state significantly lowers the branching factor, making it feasible for MiniMax to search the entire remaining game tree to a substantial depth. In this endgame scenario, the precise evaluation capabilities of MiniMax, coupled with its meticulously crafted heuristic function (as described in the previous section), become highly advantageous. The MiniMax algorithm can then determine the optimal sequence of moves to guarantee victory, if one exists, or to make the most advantageous decisions in complex endgame situations.

By strategically combining MCTS with logical move pruning for the opening and mid-game, and switching to MiniMax for the endgame, the hybrid agent aims to achieve both effective exploration and precise calculation, adapting its approach to the specific demands of each stage of the battle. The performance of this hybrid approach, compared to the individual MCTS and MiniMax agents, is further analyzed in the following sections.

3 Statistics

The idea was to create an automated way to evaluate our agents. So we have implemented a Python script (`pkm_battle.py`) which performs `N_BATTLES` battles for each combination of value assignment to the parameters indicated in the `.env` file. Then it saves the metrics, computed for the `agent0` (which you can find in section 3.1), in a `.csv` file with the values assigned to the parameters. In this way, we could find the best combination of parameters' assignment with which the agents perform better (except for Random and Logic agents which don't use any parameters). An in-depth explanation of the commands that can be used and how to execute battles is present in the `README` file of the GitHub repository.

The heatmaps represent the performance of the agents in different matchups generated with 100 and 300 battles battling the agents against each other using the best hyperparameters found. The values in each cell correspond to the results that the row agent (agent on the left) achieves against the column agent (agent at the bottom). These percentages are calculated based on multiple battles between the agents, providing a comparison of their relative strengths. Higher values indicate a better performance of the row agent against the column agent.

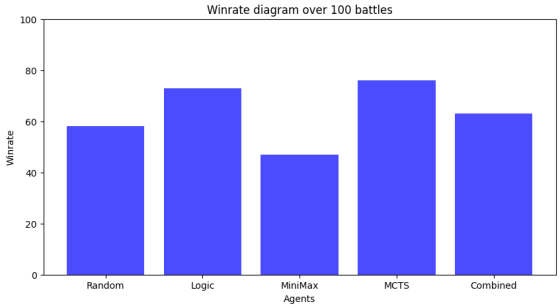

Figure 3: winrate against Random agent over 100 battles.

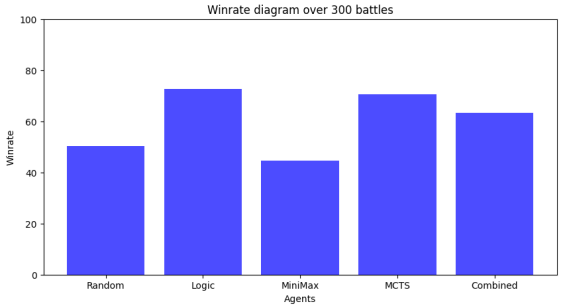

Figure 4: winrate against Random agent over 300 battles.

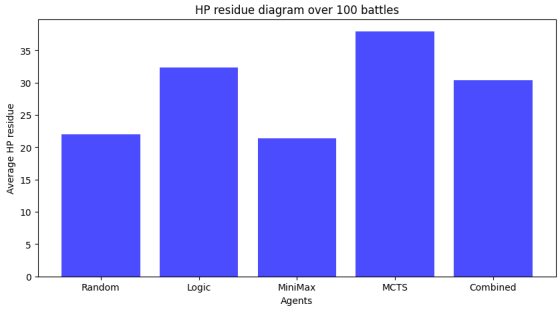

Figure 5: hp residue against Random agent over 100 battles.

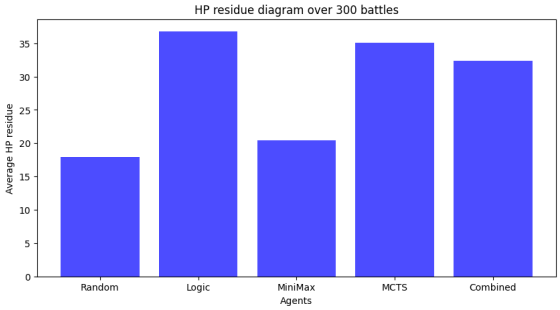

Figure 6: hp residue against Random agent over 300 battles.

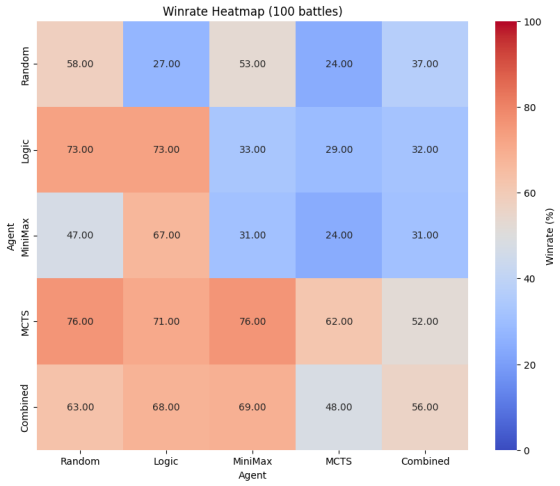

Figure 7: winrate over 100 battles.

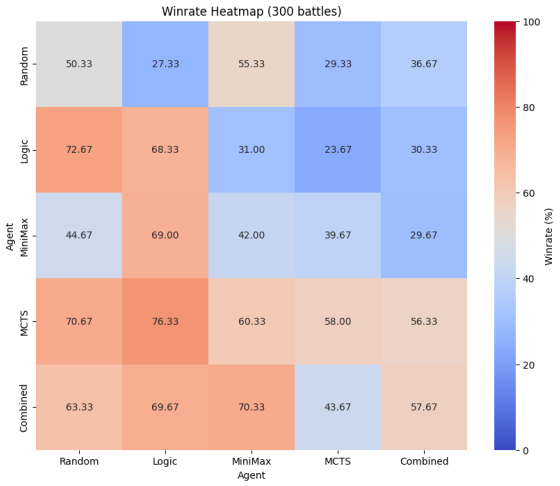

Figure 8: winrate over 300 battles.

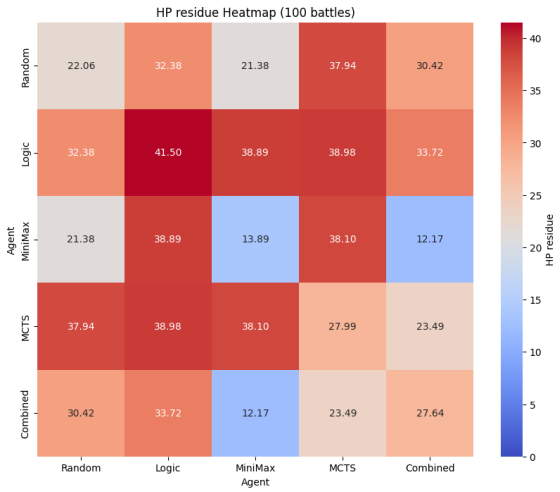

Figure 9: hp residue over 100 battles.

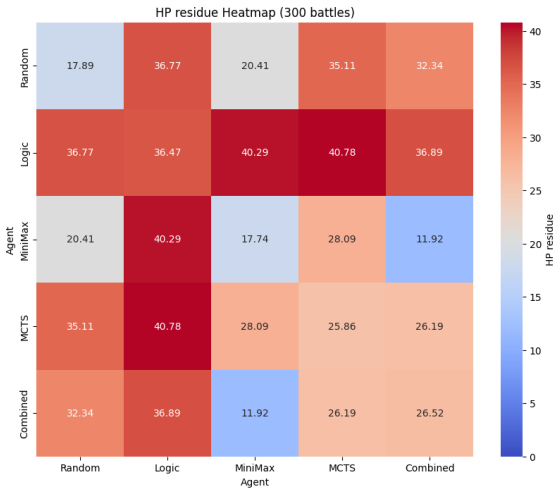

Figure 10: hp residue over 300 battles.

4 Conclusions

Analyzing the collected metrics and the generated plots, we have drawn these conclusions:

- **MiniMax performs a bit worse**: because it assumes that the opponent always acts optimally, choosing the best possible move to maximize the payoff. However, when faced with an agent that acts completely randomly, such as a "random agent", this assumption no longer holds. The random agent does not choose moves strategically; instead, it simply "chooses randomly", without following any optimal logic. As a result, the MiniMax algorithm fails to provide an effective strategy against an unpredictable opponent. The strategic behavior of MiniMax, which tries to "predict" the opponent's best move, does not adapt to the randomness of the agent, leading the algorithm to make suboptimal decisions because it relies on incorrect assumptions about the opponent's behavior.

  An evidence of this can also be seen when comparing MiniMax's performance against a logic-based agent. In battles against a logic agent, the MiniMax agent wins 70% of the time over 100 and 300 battles, as shown in the heatmaps in 7, 8. This is because, unlike the random agent, the logic agent always chooses the move with the highest priority according to the rules in its knowledge base. In contrast to the randomness of the random agent, the logic agent's behavior is structured, which allows MiniMax to anticipate its moves more effectively, resulting in higher success rates during the battle.

- **Unbalanced self battle**: when an agent battles against itself, using the same strategy and battle policy, the outcome is not necessarily close to a 50% win rate. This is because there is an element of randomness in the generation of the players' teams, which introduces uncertainty into the battle. Even though both agents follow the same strategy, the random selection of Pokémon for each team can lead to one agent having a significant advantage over the other. For example, one team might consist of Pokémon with strong type advantages over the opponent's team, resulting in a higher likelihood of winning. Conversely, the other team may have weaker type matchups or a less optimal composition, leading to more frequent losses. This randomness in team generation creates variability in the outcomes, meaning that, despite the agents following identical strategies, one may win disproportionately more often due to being "favored" with a stronger team composition, which is particularly advantageous when exploiting type effectiveness. Therefore, the results are not evenly split, and the win rate can diverge significantly from 50%, even when both agents are using the same battle approach. Obviously, this behavior is mitigated with the growth of the number of battles.

- **Combined doesn't dominate**: the combined agent performs slightly worse than the standard agents due to the *MiniMax effect* described in the previous point. By attempting to balance strategies across multiple agents, the combined agent may fail to capitalize on the specific strengths of individual strategies, leading to suboptimal decisions in critical situations.

  Otherwise achieves excellent results against other agents. This suggests that its strategy is particularly effective when facing structured or predictable opponents but struggles to adapt to the unpredictability of random strategies.

In conclusion, the analysis demonstrates that the effectiveness of a strategy is highly context-dependent and varies based on the nature of the opponent. While algorithms like MiniMax perform exceptionally well against predictable agents, their performance drops significantly when facing random opponents, revealing the limitations of assuming optimal rationality. Similarly, the role of randomness in team composition and battle dynamics highlights how external factors can heavily influence outcomes, regardless of the strategy employed.

The behavior of the combined agent reflects the trade-off between versatility and specialization, suggesting that a better balance of strategies could lead to more consistent results. This study paves the way for further exploration to refine algorithms and enhance their adaptability to unpredictable contexts, ultimately fostering greater robustness and strategic flexibility.