Comparing The Performance of Minimax and MCTS Agents Playing Pokemon Showdown

Burak Dincer and Wei Hong
New York University,
New York, US
{bd1308,wh951}@nyu.edu
Github Repository: https://github.com/brkdncr94/showdownaiclient

***Abstract*— The following paper explains in detail the two agents that we have implemented to play Pokemon Showdown, an online Pokemon battle simulation.We have implemented a simple Minimax agents and an agent that is based on doing Monte Carlo Tree Search. These agents are tested against other basic agents and the results of their simulated battles are compared.**

I. INTRODUCTION

Pokemon is a role playing game franchise that is very well-known among the general public. Arguably the most interesting facet of the Pokemon games is their duel system in which two players battle each other with teams of up to 6 Pokemon. As a result, an online community of Pokemon battle enthusiasts have been formed, thanks to the availability of Pokemon battle simulators such as Pokemon Showdown.

Pokemon Showdown is a Pokemon battle simulator that let's users play online Pokemon battles with either teams of their choosing or randomly generated teams. The two agents presented in this paper are dealing with doing battle with randomly generated teams. The reason we chose to focus on randomly generated teams is that, assembling a competent and versatile team of Pokemon is in itself a significant artificial intelligence problem.

Our agents are playing the Pokemon battles to win. There are several reasons why playing Pokemon battles is an interesting problem for artificial intelligence. First reason is the stochasticity of the particular problem we have at hand, playing with randomly selected teams. An agent needs to be able to handle the randomness to be able to play well. Another reason is the hidden knowledge aspect of the battle. It should be noted that there are two game modes for battling with randomly generated teams. For the "random" mode, both teams are generated randomly and are different than each other. In this case, the opponent's Pokemon and their Pokemon's moves are unknown until they are used. Therefore, a considerable amount of the information is hidden. In contrast, in the "random mirror" mode, the teams are again generated randomly but the two players get identical teams. With that in mind, the game becomes a perfect information game with the "random mirror" mode. Both of these cases, hidden knowledge and perfect knowledge game playing, are interesting problems on their own. However, if we were to combine them by creating an agent that can play well in both cases, we would be taking a small step towards a more generalizable agent.

The agents are based on the Pokemon Showdown AI Client framework [1], provided by Scott Lee for the Showdown AI Competition 2017 organized by Scott Lee and Julian Togelius [2]. The framework is based on the Pokemon battle simulator by Guangcong Luo [3] that is the backbone of Pokemon Showdown.

II. RELATED WORK

There are several artificial intelligence agents online that play Pokemon Showdown, two of which are David Stone's Technical Machine [4] and Harrison Ho and Varun Ramesh's Percymon [5].

Stone's Technical Machine is using a modified version of the basic Minimax algorithm. It is modified to calculate an expected outcome for each move considered, with the moves resulting in different outcomes with different probabilities. Technical Machine has a significantly larger range than our agents as it also deals with team building by learning from previous matches. As previously stated, team building is out of the scope of our agents since the framework is supplying randomly generated teams.

Ho and Ramesh's Percymon is an agent that is a lot more similar to our Minimax agent. It is also based on Luo's Pokemon Showdown simulator [3]. What's more, Percymon is only dealing with the battling aspect of Pokemon Showdown, just like our two agents, as well. However, we are implementing a simple Monte Carlo Tree Search (MCTS) agent as well as the Minimax agent. Therefore, we are also interested in if, by using MCTS, we can overcome Minimax agents' depth limitation due to the high branching factor in this game and improve, even slightly, on Minimax agents' results.

As acknowledged by the authors of both of the agents mentioned above, the biggest limitation of Minimax while playing this game is the time it takes to evaluate every possible move and their outcomes. When playing Pokemon Showdown, each player has 9 choices, unless it is a special state in which the user's choices are limited as a result of prior actions. As a result, the game has a branching factor of 81 since a turn requires both players to make a choice out

of their 9 options. In comparison, a game of Chess typically has around 30 actions to choose from [6].

By implementing a version of MCTS, we aim to approximate a deeper tree and make decisions based on that tree. Finally, we will see if that approach can actually improve upon the results of a simple Minimax agent that does not go deeper than 2 levels.

III. Challenges and Solutions

A. Branching Factor

If we take the option to switch Pokemon into consideration, the effective branching factor of each turn is 81 since each player has 9 options,4 moves and 5 Pokemon to switch to, unless it is a special situation or some of the Pokemon have fainted. Therefore, at the beginning of the battle, the branching factor is more than double that of a game of Chess.
Solution: Since switching Pokemon comes with a huge penalty in terms of complexity, we do not take the option to switch into account unless the simulations show that current active Pokemon is losing. In addition, we assume opponent will never switch their Pokemon. This assumption might be problematic. However, it makes our evaluation and simulations much simpler by substantially lowering the branching factor. As a result, the effective branching factor becomes only 16 for the best case, and 36 for the worst case.

B. Evaluation Method

1) Late feedback: The biggest problem with the evaluation method is that you cannot get any instant feedback after you choose a move. For instance, for moves like sleep that have status effects, the value of the move can only be accurately evaluated after 2 or 3 turns. In addition, the effect of a move is related to opponent's move as well.

2) Current Game State Evaluation: Another reason why the Pokemon battle is complex is that it is extremely difficult to evaluate the current game state. Suppose you have 6 Pokemon with full HP and your opponent only has one with 1 HP. Although it seems that you have a great advantage, it is still possible that you might lose the game if your opponent's Pokemon is a very powerful one with a clear advantage against your Pokemon. It should be noted that this can never be the case in the "Random Mirror" game mode with identical teams of Pokemon.
Solutions: The evaluation of battle Pokemon without simulations is not accurate, even misleading. To get a useful evaluation, the battle needs to be simulated for at least several turns. Since we have previously assumed that neither player will switch their Pokemon, the game becomes a lot simpler to evaluate. Therefore, the HP percent and status changes can be used to evaluate the game state.

C. Problems With Simulations

1) Complex Eules and Moves: First of all, Pokemon showdown battle has lots of specific rules which make the game state and performing simulations a complex and expensive task. For example, some plays, such as trying to put all the Pokemon in the opponent's team to sleep, are illegal. Secondly, all moves are written as separate functions in the game framework [3], which makes 719 different moves with 20 thousand lines of code in all, just to define the different moves. The move functions are uncategorized and keep increasing as new generations of Pokemon games arrive. What's more, new moves tend to be very complex and have weird effects, which make them almost impossible to be categorized to get a generic function.

2) Simulation result is random: As we know, each move has its own accuracy and therefore not every move is guaranteed to hit the opponent successfully. Some moves even have the effect to change the hit and evade rate. What's more, most moves have some side effects which occur at some certain rate. For example, for a move with 80% probability to hit and 30% probability that the side effect will occur, when we do the simulation, there would need to be several branches. These branches would have to cover the cases where the move does not hit, the move hits but the side effect does not occur, the move hits and the side effect occurs. Such stochasticity makes simulating turns a complex task. In addition, due to the stochasticity, there will be no guarantee that the results in the actual game play and our simulation will be identical.

3) Framework Problem: The Pokemon Showdown Client AI framework is written as a NodeJS project. It is not as efficient as it would be were it written in C++ and the amount of information stored in the game states slows down the process of making a copy of the state. It should be noted that making copies of game states and generating new game states is required to do simulations of actions to be taken. As a result, it takes approximately 2 seconds to do a depth 1 simulation, approximately 20 seconds for a depth 2 simulation, and approximately 2 minutes for depth 3, which is the depth we would like to achieve with out Minimax agent.
Solutions: The problems with the performance of the framework's functions is almost impossible to solve without external help. To make the best of the tools provided, currently we are making only 8-10 random simulations and using an average result instead of branching all the possibilities. Framework problem might be solved by rewrite the battle engine. Currently we are just using depth 1 simulation.

IV. METHODS

We are using the game Pokemon Showdown, a Pokemon battle simulation to test our agents. This simulation only approximates the battles in an official Pokemon game as it does not include items such as health potions. It involves two players battling with teams of 6 Pokemon. The users can either assemble their own teams or play with randomly generated teams. We will be focusing on playing with randomly generated teams as assembling a versatile teams is a substantial artificial intelligence problem on its own and therefore out of the scope of our project.

In this project, we have implemented 2 different agents to play Pokemon Showdown to win. The first of these agents is a simple implementation of Monte Carlo Tree Search

Fig. 1. A screenshot from an example Pokemon battle

(MCTS). The agent using MCTS tries to approximate a tree that would be generated by a Minimax agent. By not generating every single possible state, it can afford to go deeper than a simple Minimax agent.

The second agent that we have implemented is a simple Minimax agent. It generates the outcome of every single option and evaluates each state.

A. *Monte Carlo Tree Search Agent*

The agent named BWAgent implements a simple version of MCTS. The idea is to approximate the outcome of the Minimax algorithm while going deeper than a Minimax agent can by generating only some of the game states. By going deeper into the tree, the agent is meant to make decisions that are focused on the long term gains.

The agent starts by checking if the opponent's Pokemon is super-effective against its Pokemon. If the opponent is super-effective, which means the opponent's attacks will deal significant damage, the agent will switch its current Pokemon with the best replacement in the team. The best Pokemon to replace the current one is defined as the Pokemon with the most Health Points (HP) that the opponent is not super-effective against. Switching when the opponent is super-effective against your Pokemon is trivial as trying to attack will deal minimal damage to the opponent. Therefore, this initial check before we start building the tree will only save our agent computation time.

If the opponent's Pokemon is not super-effective, BWAgent will start planning its next move by building a search tree. Initially the first level of the search tree will be automatically occupied. The root of the tree will be the current game state while its immediate children will be the game states resulting from each of the options BWAgent has. To end a turn and generate the next game state, both players are required to submit their choices. To generate the resulting game states, BWAgent assumes that the opponent will make the choice that will result in the worst case for BWAgent. However, when simulating the opponent's decision, it is assumed that the opponent decides in a greedy manner by looking only 1 game state ahead. Thanks to our assumption about the opponent's choice, our effective branching factor becomes 9 instead of 81.

Nodes of the search tree hold information regarding the parent node, child nodes, the choice that resulted in that node, a win count and a play count. Play count records how many paths there are from a leaf node to the root that include that node. Win count records how many of those paths lead to a favorable outcome. We define how favorable a choice is by the win count to play count ratio of the node that choice leads to.

Once the initial level of the search tree is occupied, the 4 steps of MCTS that are Selection, Expansion, Simulation and Back-Propagation are performed. In the selection step, BWAgent randomly selects a node, a possible child of which has not yet been generated. In other words, if game state X has 9 possible moves, the node containing X can only be selected in the Selection if it has less than 9 children.

After a node is selected, BWAgent randomly chooses an option that has not yet been considered for that node's game state. The resulting game state from that option is generated and added to the list of children of the node that was selected.

Then comes the Simulation step of MCTS. After the new game state that has not been considered before is generated in the Expansion step, the Simulation step performs a random roll-out starting from that game state. In this step, BWAgent simulates the outcome from choosing random options while assuming the opponent greedily does their best, as explained above. The simulation is performed for 5 turns, or until a terminal state is encountered. Currently, a terminal state is defined as a state in which one of the Pokemon on the battle field has reached 0HP and fainted. Please note that simulating for 5 turns ahead was an arbitrary choice. The number of turns could be increased but would result in the agent taking longer time to make a decision.

Once the Simulation step of the algorithm terminates, it is time for the last step, Back-Propagation. In this step, starting from the leaf node where the Simulation step had terminated, the play count of each node on the path to the root of the tree is incremented by 1. If the leaf node is a favorable state, the win counts are also incremented by 1. This will end a single iteration of the tree building.

After repeating the above states for a given number of iterations, currently 20, BWAgent will choose the most favorable option by pushing its immediate child nodes into a priority queue and selecting the node with the highest win count to play count ratio. It should be noted that it is possible for all of the options to lead to a node the ratio of which is 0. That means, as far as BWAgent is able to see, none of the choices it has leads to a favorable state. In that case, BWAgent will make a greedy choice by estimating the HP damage that can be dealt by choosing each option and select the one that will deal the highest damage. This greedy choice mechanism is the same as the one used by

OTLAgent that comes with the Pokemon Showdown AI Client framework [1]. Therefore, we would expect BWAgent to perform at least as well as OTLAgent.

A very important aspect of planning ahead in this game is the amount of information we have about the opponent. To be able to generate game states, the agents need to make some assumptions about the opponent's Pokemon. We have already mentioned that in the "Random Mirror" game mode, we know for a fact that both teams are identical, while that is not the case for "Random" game mode. As a comromise between the two game modes, BWAgent assumes that if the opponent's Pokemon on the battle field is the same species as a Pokemon from BWAgent's team, those two Pokemon are identical. This assumption is trivial for the "Random Mirror" mode. However, even in the "Random Mirror" mode, this means that the agent does not make any assumptions about the opponent's Pokemon that has not been revealed yet. As for the "Random" game mode, this assumption will not necessarily hold true, but it should serve as an adequate estimation of the opponent's Pokemon.

B. Minimax Agent

Since Pokemon battle is a two-player adversarial game, Minimax seems to be a valid approach for an artificial intelligence agent to take. However, it should be noted that this game has a branching factor that is considerably higher than what the Minimax algorithm can handle. Therefore, we needed to take an approach that simplifies the game by making some assumptions.

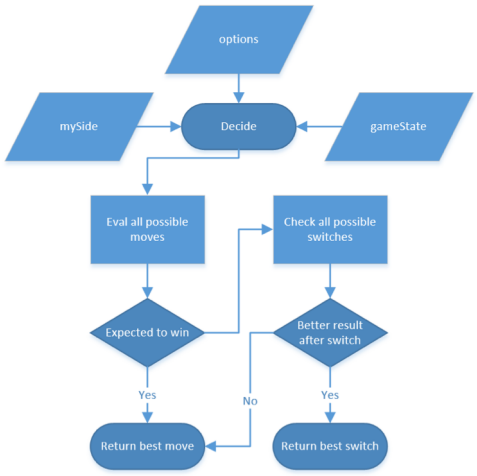

Fig. 2. Decide function

1) Decide Function: Decide function, the data flow of which is shown in Figure 2, is the function called by the interface when it is the agent's turn to make a move. It is in the Decide function that the agents need to find the best option for them out of their options, and make a choice.

First of all, when the Decide function is called, the agent checks all the possible moves, and uses the getWorstState function, shown in Figure 2, to check the worst possible outcome of the selected move after simulating the game several turns. Then, the agent will evaluate the worst possible outcomes that resulted from each of its possible moves, find the best out of these results and make the move that will lead to that particular outcome.

Secondly, if the outcome of the best move has a better game state evaluation point than the current one, it implies that our current active Pokemon will be winning against the opponent's Pokemon. As a result, we will skip evaluating the options regarding switching Pokemon. However, in the case that the evaluation of the upcoming game state is worse than the current game state, we shall check all the possible switches.

Thirdly, when evaluating the outcome from choosing to switch its active Pokemon, the agent will again use the getWorstate function, this time with a switch command. Then, for each switching option, the agent will simulate the game for a few turns and get the worst status after choosing the switch command. The evaluation of the current game state will be compared to the evaluation of the game state that was generated as a result of switching Pokemon and the difference between the two evaluation scores will be stored. The difference in score will be compared to the difference between the evaluation of the current game state and the evaluation of the game state that resulted from choosing the best attacking move. By making these comparisons, the agent will decide if making a switch will put it in an even worse game state than the attacking move, which still did not lead to a favorable game state. If switching Pokemon will result in an even worse situation, it will not be wise to switch to another Pokemon we. Therefore,the agent will choose the best possible attacking move instead of switching.

Finally, the best option that the agent has found will be returned as its decision.

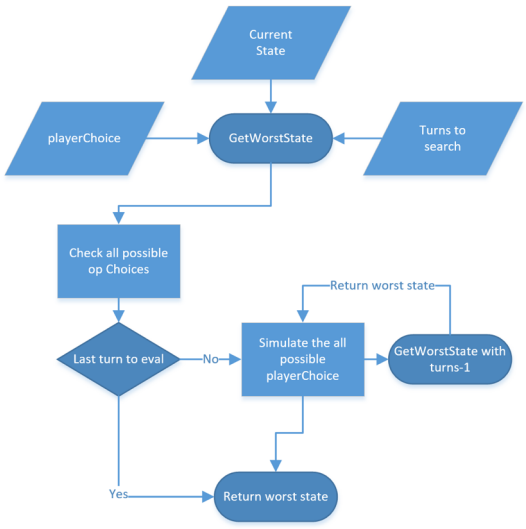

Fig. 3. GetWorstState function

2) getWorstState function: This function aims to get worst outcome that will result from a given choice after simulating the battle in several turns. It uses a recursive method to

choose the worst resulting state. If the parameter that controls the number of turns to iterate over is 0, it terminates as this means the function should not simulate the results any longer. Given our agent's choice, the function will generate the resulting game states for every option the opponent has and when it terminates, returns the worst possible game state.

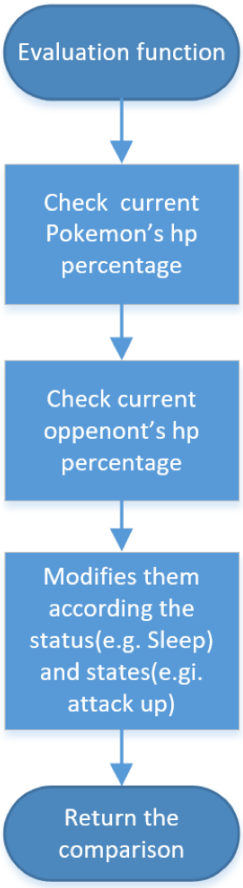

Fig. 4. Evaluation function

3) Evaluation Function: Game state evaluation is one of the most important parts of Minimax algorithm, or any kind of game playing artificial intelligence algorithm that depends on searching through game states. Our agents are using the state evaluation functions to estimate which player is winning or losing in any given game state. A better and more sophisticated evaluation function would provide a more accurate view of the game state. Therefore, it can help the agents, even with shallower simulations, get better results than a bad and misleading evaluation function would. The evaluation function we used is a simple one. It depends on the current HP percent of the agent's and the opponent's Pokemon on the battle field. If a Pokemon has some debuff status, such as sleep or confusion, the percent will be cut to 2/3 of the original value, and modified according to the positive status changes such as attack boosts.

V. Results

To test our agents, we have made them battle against the basic agents that were provided along with the framework itself. The battles were simulated locally, in the "Random Mirror" game mode. Although the agents have not been tested in the "Random" game mode that does not give identical teams to both sides yet, they should be able to perform almost equally well in either game mode.

A. Monte Carlo Tree Search Agent

TABLE I
Competition vs Greedy Algorithm

Agent	Win of games	Win rate
Minimax Tree Search	26	52%
Sample Greedy Agent	24	48 %

The results of the simulated battles between BWAgent and the basic greedy agent provided with the framework can be seen in Table I. As mentioned above, the MCTS agent makes the same decision the greedy agent would do if none of the options it has lead to any states where it is winning. The agent performing almost exactly the same as the greedy agent leads us to believe that it had to make the greedy choice most of the time that it was playing. The reason might be the simple state evaluation function it was using and this requires further investigation.

B. Minimax Tree Search Agent

TABLE II
Competition vs Greedy Algorithm

Agent	Win of games	Win rate
Minimax Tree Search	69	69%
Sample Greedy Agent	31	31 %

TABLE III
Competition vs Minimax Algorithm

Agent	Win of games	Win rate
Minimax Tree Search	9	90%
Sample Minimax Agent	1	10 %

We could say that the results of our Minimax Tree search agent's simulated battles are quite promising. The results can be seen in Table II and III. The reason why the battle with sample minimax algorithm was only run 10 times is that the fact that the sample Minimax agent performed quite slowly. Therefore, it did not allow us to do too many tests with it.

Currently our Minimax agent's search depth is set as 1 when calling the getWorstState function to simulate the resulting game states. If we could make the necessary improvements in our agents and in the framework to boost the performance and allow us to perform a deeper search, at

least depth 2, in an acceptable amount of time, we believe the results of the Minimax agent would only improve. What's more, the analysis of a battle log seems to approve our assumptions. The agent does well on move evaluation but poorly on switching Pokemon(e.g. switch immediately after takes a hit). This is due to the fact that to accurately evaluate the results of switching Pokemon, we need the agent to go at least one more level deeper in the tree search.

VI. Conclusion

In this project, we have implemented two agents to play Pokemon Showdown battle simulation based on Minimax tree search and Monte Carlo Tree Search. The results of the Minimax agent are quite promising and the Minimax tree search agent plays considerably better than the greedy agent and almost beats normal Minimax agent every time, although we were able to conduct only a small number of tests with the sample Minimax agent. However, it is quite clear that our BWAgent, based on MCTS, performed quite poorly. It was expected to work at least as good as the greedy agent it was tested against, but did not really outperform it. Therefore, the results from the MCTS agent require further inspection to improve performance. Due to the time and framework limits, our work is far from being perfect, or completely satisfactory. If we can simulate deeper and come up with a better evaluation function, we think our agents can do considerably better than they do now.

References

[1] S. Lee, "Pokemon showdown ai client."
[2] S. Lee and J. Togelius, "Showdown ai competition."
[3] G. Luo, "Pokemon showdown."
[4] D. Stone, "Technical machine."
[5] H. Ho and V. Ramesh, "Percymon: A pokemon showdown artificial intelligence."
[6] G. N. Yannakakis and J. Togelius, "Artificial intelligence and games," in *Artificial Intelligence and Games*, 2017, p. 53.