Using Stochastic Local Search to Generate Optimal Pokémon Teams

Ethan Hart
Comp 364: Artificial Intelligence
December 15, 2022

Introduction

One of the most popular franchises in entertainment since its debut in 1996 has been Pokemon, the monster-collecting and battling game exclusive to Nintendo consoles. Although it has branched out into TV shows, movies, and even its own trading card game, I've been playing the Pokémon video games ever since I was six years old. Even today, as a junior in college, I am just now wrapping up the newest Pokémon game released on the Nintendo Switch. An integral part of playing Pokemon is catching and building teams to battle against other trainers. The objective of team building and battling is to integrate six "optimal" Pokemon into the user's team and attempt to defeat all of an opposing trainer's Pokemon. However, choosing what Pokemon to have on your team and battle with is a complex problem that has many different solutions.

The goal of my project was to utilize A.I. techniques to select the most optimal Pokemon team to defeat an opponent's Pokemon team. With the release of the newest games, there are now upwards of 1000 Pokemon in the franchise. However, for the scope of this project, I shortened the sample size down to all fully evolved Pokemon from the first games *Pokemon Red Version* and *Pokemon Blue Version.* This decision was made because it is the set of Pokemon that most people are familiar with, and because data entry was time-consuming due to each Pokemon needing extensive entry into the database I created from scratch.

Optimality Definition

Before we continue, we must define what it means to select optimal Pokemon. In my research, I simplified a Pokemon's optimality into two categories:

1.) A Pokemon's typing: Each Pokemon can have either one or two types. A type indicates a category a Pokemon belongs to. Some of these types include "fire", "water", or "grass." When a Pokemon of type *x* attacks a defending Pokemon of type *y* with a move of type *x*, the damage done is partially calculated by how type *x* interacts with type *y*. For example, the "fire" type does neutral damage to a Pokemon that has the "normal" type, does half its damage to a Pokemon that has the "water" type, and does double damage to a Pokemon that has the "grass" type. These damage modifiers are important in Pokemon battling, so it is optimal to select Pokemon who can do double, or "super effective", damage to a defending Pokemon.

2.) A Pokemon's statistics: Each Pokemon has a collection of stats in different categories. All Pokemon have a number associated with their Hit Points (HP), Attack, Defense, Special Attack, Special Defense, and Speed. A good measure of a Pokemon's overall strength is the sum of the numbers associated with each statistical category. Thus, it is considered optimal to select Pokemon with higher base stat totals to add to a team.

Methods

After careful consideration, I decided to use stochastic local search as the A.I. technique to build optimal Pokemon teams. I chose this method for a few reasons. For one, since the search space was small enough (79 total Pokemon), the Pokemon that the search

method would return would be very close to the most optimal Pokemon. Secondly, I thought that stochastic local search would be easy to implement. Finally, I didn't want the traditional local search method to start at the same section of the Pokedex every time it was called, so I went with the stochastic iteration so each search started at a random point.

This project was coded in Java, as I thought that an object-oriented programming approach would be useful, as well as the language being the most comfortable for me to use. The core of my project started with defining a Pokemon class. Each Pokemon, by class definition, has a name, a list of types, a base stat total, and a number for each stat type. These statistics were pulled from a website called Pokemon Database, which stores stat information on every Pokemon.[1]

The next step was adding each Pokemon to a database, which is known as the Pokedex. The Pokedex is its own Java class, and the data structure I used to store each Pokemon was a HashMap that maps a string containing the Pokemon's name to a Pokemon instance that stores all the data listed above. By using the generatePokedex() method in the Pokedex class, an entire Pokedex with all fully-evolved Pokemon from *Pokemon Red Version* and *Pokemon Blue Version* is generated.

Next, I implemented a Java class called TypeEffectiveness(). This class takes in an attacking and defending Pokemon and determines if the attacking Pokemon is super effective against the defending Pokemon. To determine type effectiveness, I constructed a two-dimensional array, with the row representing the type of the attacking Pokemon, and the column matching the type of the defending Pokemon. Each element in the array would have a corresponding multiplier; 2.0 if the attacking type is super effective, 1.0 if the

---

[1] Pokemon Database, ed. "Generation 1 Pokémon Stats." https://pokemondb.net/pokedex/stats/gen1.

attacking type is neutral, 0.5 if the attacking type is not very effective, and 0 if the attacking type has no effect. For each type of the attacking Pokemon, the type is converted for a row to look up, and the corresponding column to look up in the type chart is decided by each type of the defending Pokemon. If the attacking type is super effective against a defending type (and a potential second defensive type doesn't resist the attacking type), we return a Boolean value of true. If the attacking types are not super effective, we return false.

The final design step was implementing the stochastic local search method which is the focal point of the project. This was implemented in the Driver.java class which is the executable file for the project. The code was implemented per instruction of the pseudocode written on page 122 in the course textbook, *Artificial Intelligence: A Modern Approach*.[2] In addition, I added an explored list that would keep track of the Pokemon already picked for the team. The purpose of this was to make sure the algorithm wouldn't pick the same Pokemon more than once.

For the heuristic function *h(x)* that would determine each Pokemon's value, I implemented the following algorithm:

$h(x) = \{2*BaseStatTotal(x) + StatQualityFunction(x)\}$ — If attacking Pokemon *x* is super effective

$h(x) = \{BaseStatTotal(x) + StatQualityFunction(x)\}$ — Otherwise

[2] "Beyond Classical Search." In *Artificial Intelligence: A Modern Approach*, 3rd ed., edited by Stuart Russel and Peter Norvig, 122-25. Pearson Education, 2010. PDF.

Where the helper function *StatQualityFunction(x)* looks through each stat category of the Pokemon *x* and returns 50**n*, where *n* = the number of stat categories that have a base total ≥ 90.

When designing this function, I wanted to prioritize Pokemon that were super effective against an opponent. Secondly, I wanted Pokemon with higher base stat totals to be chosen and rewarded over Pokemon with lower base stat totals. Finally, I chose to implement the *StatQualityFunction(x)* to reward Pokemon that have several exceptional stat categories, as opposed to a Pokemon that has most of its stat total in one category, making that Pokemon one-dimensional.

Results

To display the results of the project, I implemented a series of printing methods in the Driver.java class to print the teams of the opposing trainer and the user's trainer. Some example outputs are as follows:

Figure 1.)

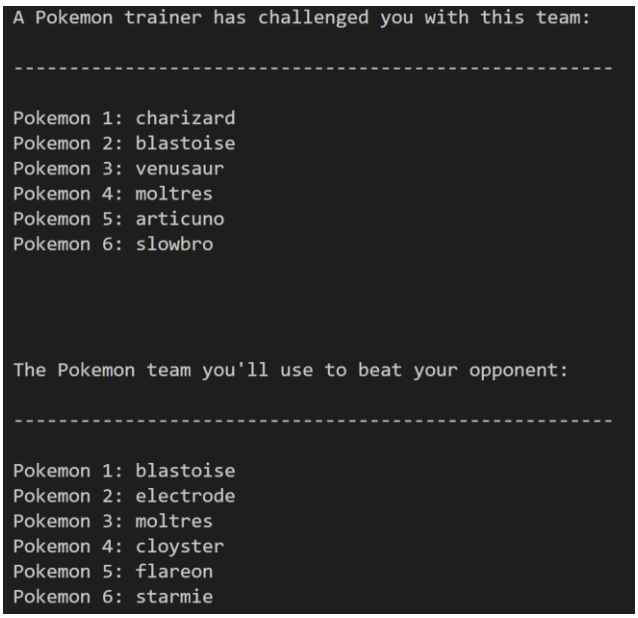


Figure 2.)

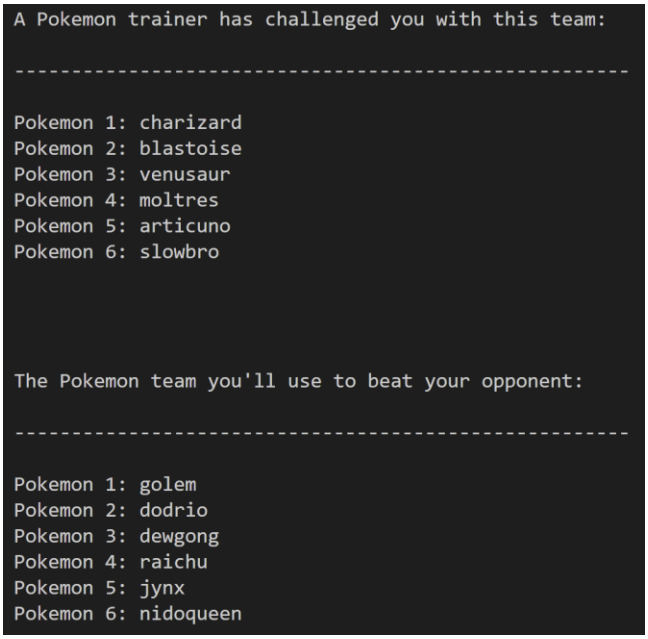
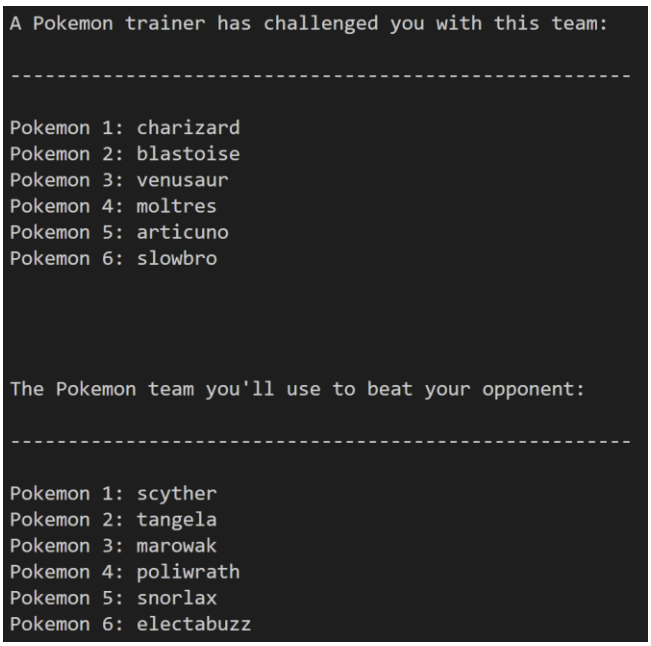


Figure 3.)



In summary, the results are overwhelmingly positive. Generally, for each Pokemon on the opposing trainer's team indexed [1…6], the corresponding index of a Pokemon on the user's team has a super effective typing against the opposing trainer's Pokemon. Then, there are a few cases where instead of a super effective Pokemon being selected, a Pokemon with high base stats that are of high quality are chosen, such as a Snorlax being selected to counter an Articuno in Figure 3. Only a few times did the search method select a Pokemon that would not be considered wholly optimal, such as Jynx (which has ice and psychic typing) being selected to counter Articuno (which has ice and flying typing) in Figure 2. This occurs on rare occasions due to a potential team member's neighbor having a lesser value than it, allowing for the local search to identify the potential team member as a local maximum and add it to the team.

Conclusion

The approach to selecting optimal Pokemon teams using stochastic local search was a success. The technique was able to consistently generate user teams that can counter an opposing Pokemon at the corresponding index for both teams. Even in the rarest times when a questionable counter Pokemon was chosen, other team members could usually counter an opposing Pokemon that may take advantage of the questionable team member.

Future Work

There are several directions that future work could take to improve this project. This project only included Pokemon from the first generation of Pokemon games, and further work could work on adding all of the existing Pokemon. Considering the scope of this project and the time that I had to design it, I simplified Pokemon value down to a Pokemon's typing and stats. What could be added in the future is the consideration of the moves that each Pokemon can learn. Stochastic local search suited this project well, as approximations of optimal Pokemon were closer to the most optimal Pokemon than not since a small Pokedex of 79 Pokemon was used. As more Pokemon are added to the Pokedex of this project, the local search technique that is used could be shifted to simulated annealing, which will give more optimal solutions on larger datasets.

Bibliography

"Beyond Classical Search." In *Artificial Intelligence: A Modern Approach*, 3rd ed., edited by Stuart Russel and Peter Norvig, 122-25. Pearson Education, 2010. PDF.

Pokemon Database, ed. "Generation 1 Pokémon Stats." https://pokemondb.net/pokedex/stats/gen1.