Pokemon In-Game Party Optimization: Generations 1 - 3

Author: Spencer Stromback, Pharm. D., M.S.

Date: October 7th, 2025

Table of Contents

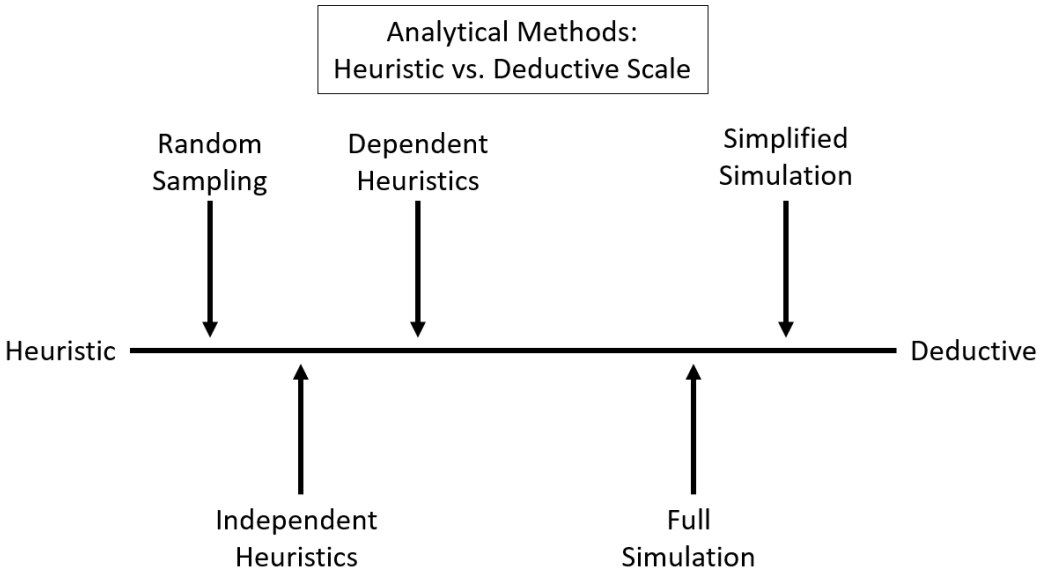
1. Introduction

Debating the fitness of individual pokemon, and by extention pokemon parties, is a longstanding tradition among fans of the franchise. Indeed, online debates on the topic can be found dating back to 1999, less than a year after the release of Pokemon Red/Blue in North America [^1]. Such debates still continue today [^2]. However, much of these discussions are largely subjective in nature. Deliberations often involve a combination of conventional wisdom, player preference, and personal experience. This analysis outlines an objective approach to answering such questions. Specifically, this analysis attempts to utilize complex computational methods to estimate the best possible party of 6 pokemon for a "normal" in-game playthrough of pokemon red, crystal, and emerald versions.

2. Potential Analytic Approaches

There are many potential approaches to consider when designing an analysis like this. Each approach varies, but they all fall somewhere on a scale of heuristic to deductive methodology. Heuristic is defined as "Involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods" [^3]. Conversely, deduction is defined as "The deriving of a conclusion by reasoning" [^4]. In other words, heuristic evaluation methods are generally more simplified, and thus more efficient than deductive methods. On the other hand, deductive methods tend to mirror or measure the exact process of interest as closely as possible. Deductive methods are more likely to find the true optimal result (or are guaranteed to do so, assuming the method is truley deterministic in nature). This is possible because they utilize more information than heuristic approaches when solving the problem in question. Five examples of methods within this spectrum are: independent heuristics, dependent heuristics, random sampling, simplified simulation, and full simulation.

Figure 1: Heuristic vs Deductive Methodology



1. Independent Heuristics

Independent heuristic methods only consider the characteristics of each individual pokemon, and ultimately party, of interest. The greatest advantage of this methodology is its simplicity. Taking encountered pokemon characteristics into account increases the computational complexity of most methods by the number of encountered pokemon in the game (generally by a factor of 600 to 800 times). Examples of this methodology include assessment of base stat totals, effective hitpoints, damage output, and independent type coverage.

When beginning work on this project, independent heuristics was the first general strategy explored. A combination of effective hitpoints, average damage output, and dependent type coverage were consolidated into a single metric. While this method was quick and cheap computationally, it failed to yield compelling results.

Independent heuristics have been previously utilized for similar analyses. In 2021, Tommy Odland posted "The best Pokémon party" to his blog [^5]. In this project, he uses a function of offensive vs defensive type effectiveness against all 151 pokemon as his objective function, which he refers to as "edge". While this is technically a comparison against other pokemon, it does not consider what pokemon are actually encountered throughout the game. Because of this, it is considered an independent method. While Odland's methodology is generally sound, it fails to account for many practical considerations of determining pokemon fitness because it does not make any direct comparisons to any encountered pokemon. It instead favors a more general approach, equally considering each possible pokemon one could encounter. In order to find the optimal party he utilizes an optimization method called "branch and bound". This method is discussed later in the optimization section.

2. Dependent Heuristics

Dependent heuristic methods are similar to independent heuristics, except that they also consider factors of the opposing pokemon. Effective hitpoints, damage output, and type coverage can be compared against aggregate attributes of encountered pokemon throughout the game. For example, a game with an over-abundance of water-type pokemon encounters (such as emerald) could favor electric or grass types.

3. Random Sampling

Player sampling is a statistical approach that relies on collecting data from pokemon players in an effort to determine the best possible pokemon. There are pros and cons to statistical sampling methods. First, it is by far the most subjective method discussed. It relies entirely on player preference, with the assumption that player preference correlates with the ground truth with regard to pokemon fitness. It also relies on player surveys, which can be unreliable and difficult to obtain in any significant quantity. In theory, related statistics such as in-game time could be utilized, but such information is not generally available. Unfortunately, such statistics are not collected at scale within the game itself, as they are with many modern games today.

4. Full Simulation

Full simulation refers to fully simulating the gameplay of the pokemon game of interest. Such simulation is generally achieved through neural networks, often utilizing sophisticated reinforcement learning techniques. This simulated gameplay is then used to measure some measure of efficacy. The most intuitive measure for such an AI model to utilize would be time to beat the game, which can be easily minimized. However, it is unclear if lower game time is necessarily correlated with the best party. Once enough simulated playthroughs are achieved, the best playthrough is saved, and the final pokemon party recorded. There are many limitations to such an approach, the greatest of which being complexity. It is hard enough to train a neural network to beat a videogame at all, let alone try different pokemon party options. This method is also the most computationally expensive by far, likely costing hundreds of not thousands of dollars in cloud computing costs, and requiring weeks to months of processing time to complete.

5. Simplified Simulation

Simplified simulation is a happy medium between heuristic and deductive methods. On one hand, this method is able to represent the underlying features present within the game itself, approximating what actually happens through a real playthrough. And on the other hand, such methods are often simplified to a significant degree, compared to full simulation. A well optimized simplified simulation can take anywhere from minutes to a few hours to complete. There is also a great deal of flexibility offered with regard to choosing which features of the game to simulate, and which to not simulate. But this also means that the programmer must be highly knowledgable about the game or games of interest, as there are many nuiances that affect pokemon battles that would be easy to miss when designing a custom simulation environment.

It is worth noting that, at least in theory, full simulation should be more deductive than simplified simulation. However, it is often not fully deductive due to the significant costs (computational, time, etc.) associated with full simulation

methods. For this reason, full simulation often ends up being less deductive in practice than it would otherwise be from a theoretical standpoint.

For these reasons, simplified simulation was the method ultimately chosen for this analysis.

3. Objective Function

At its core, defining the best pokemon party is a complex combinatorial optimization problem. The objective is to identify the best possible combination of 6 pokemon out of some number of candidates, who perform best against some number of encounters. In theory, if we could measure the fitness of each party pokemon against each encountered pokemon as a single number (with smaller being better), then the objective function would look something like this:

Given...

- n candidates (indexed $i = 1, \ldots, n$),

- m encounters (indexed $e = 1, \ldots, m$),

- a matrix $A \in \mathbb{R}^{n \times m}$ with entries a_{ie} = the fitness value for candidate i against encounter e,

- A set of candidates as $S \subseteq [n]$.

The objective function can be defined as:

$$F(S) := \sum_{e=1}^{m} \min\{\, a_{ie} \mid i \in S \,\}, \;\; \text{minimize} \;\; F(S) \;\; \text{subject to} \;\; S \in \binom{[n]}{6}$$

Or, using shorthand $\wedge$ to denote elementwise minimum across vectors, the function can be simplified to:

$$F(S) := \mathbf{1}^{\top}\Big(\bigwedge_{i \in S} a_i \Big), \;\; \text{minimize } F(S) \;\; \text{subject to} \;\; S \in \binom{[n]}{6}$$

What this complex math is really saying is that we are, for each potential combination of 6 pokemon in our party, taking the best fitness value against each encountered pokemon, then taking the sum of all of those values. That final sum represents the fitness of that party of 6 pokemon. The smaller the number, the better the fitness. See the diagram below for a simplified visual representation of this process.

Figure 2: A simplified visual representation of the objective function

Party Pokémon	Encounter 1	Encounter 2	Encounter 3	...
Venusaur	0.4	0.7	0.5	...
Blastoise	0.7	0.9	0.2	...
Charizard	0.3	0.9	0.3	...
...	...	...	...	...

min values 0.3 + 0.7 + 0.2 = 1.2

Party Fitness

4. Defining Individual Fitness

The above definition of the objective function may seem rather straightforward (depending on your familiarity with mathematics and optimization). However, it overlooks one very important question: How do we define a_{ie}? In other words, how do we define the "fitness" of one pokemon against another?

In the game of pokemon, two pokemon are tested against one another in the context of pokemon battles. Within each battle, each trainer has up to 6 pokemon. These pokemon fight each other 1 on 1, with a pokemon being replaced when it is defeated (reaches 0 hitpoints). It is these 1 on 1 fights, referred to in this analysis as "encounters", within which fitness can be defined.

But how to best define fitness? What makes one pokemon's performance against an encounter better or worse? There are three general approaches to such a definition: Time oriented, general outcome oriented, and measured outcome oriented.

1. Time Oriented Approach

A time oriented approach simply measures how many turns a battle took to complete. This approach is not sufficient on its own. A pokemon losing a battle in one turn, compared to a pokemon winning a battle in one turn, should be considered quite different in terms of fitness. A time oriented approach alone would consider both of these as the same fitness. However, this approach can serve as a useful adjunct measure of fitness, to compare positive outcomes against other positive outcomes, and negative outcomes against other negative outcomes. If a pokemon won a battle in one turn, that is clearly better than winning in two turns.

2. Categorical Outcome Approach

A general outcome approach is a purely results oriented approach. Did the pokemon win the battle? Or did it lose? Or perhaps it tied, or was unavailable? Categorization is a crucial component of effective optimization, and will be discussed further in the optimization section.

3. Measured Outcome Approach

A measured outcomes approach utilizes some continuous measure to compare how well a pokemon performed against a particular encounter compared to others. The only three variables that change from the beginning to the end of a pokemon battle are experience points, move pp, and hitpoints. Experience points have no bearing on fitness, and move pp is simply a surogate for the time oriented approaoch (starting pp - ending pp = number of turns taken). This leaves us with hp as the only viable measured outcome. Thankfully, hp serves as an excellent comparable metric for determining fitness. The starting and ending hp of both the player pokemon and encountered pokemon are always directly correlated to pokemon fitness.

This analysis ultimately utilizes a combination of categorical and measured outcomes to define individual fitness.

5. Simulation

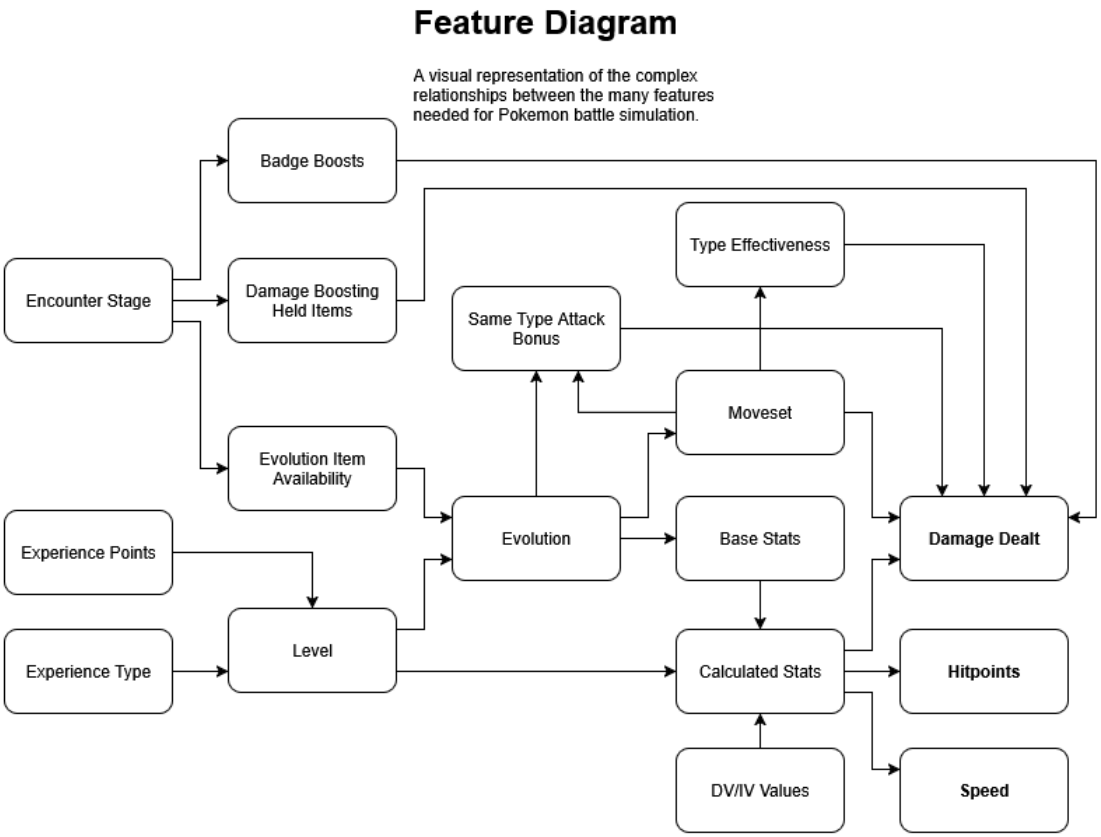
Simulating such a complex process as a full playthrough of a pokemon game requires significant domain knowledge on the subject. In pokemon, there are numerous features at play that impact the outcome of each pokemon battle. Furthremore, the analyst must also choose what features to account for, and to not account for. Accounting for more features brings the end result closer to a deductive, and potentially deterministic, conclusion, but at the cost of computational complexity. Let us next consider the various characteristics of pokemon battles.

Pokemon Battle Characteristics

When attempting to simulate a complex process, it can be helpful to start from the beginning, and try to better understand the fundamentals of the context for which the simulation is being modeled. In other words, what is a pokemon battle?

In its simplest form, each encounter within a pokemon battle consists of two pokemon taking turns dealing damage to one another until one pokemon faints (reaches 0 hitpoints). Thus, the outcome of the battle is ultimately determined by three things: The starting hitpoints, damage dealt, and speed value of each pokemon. If all three of these values can be calculated, then the outcome of the battle can be calculated mathematically without actually simulating the battle itself, turn by turn. While this may sound simple, there are numerous features that are involved in calculating these numbers, each sharing complex interdependencies with one another. Some of these relationships are outlined in figure 3 below.

Figure 3: A visual representation of the complex relationships between the many features needed for pokemon battle simulation.



Due to these complex interdependencies, there are numerous different types of data required to accurately simulate these battles. These include, but are not limited to...

- Basic Pokemon stats: base stats, types, evolution level/conditions, experience types, etc.
- "Staging": a sequential description of the game progression, from location to location
- Pokemon availability: Where each pokemon can be caught, and how
- Pokemon movesets: What moves can be learned, and at what level (or stage if a TM/HM move)
- Encountered pokemon information: Pokemon levels, movesets, stats, etc.

For this analysis, the above data was mined from a variety of sources, including (but not limited to) gamefaqs.com, pokemondb.net, pokeapi.co, bulbapedia.com, and speedruns.com. See the github repository for more details regarding the source information used for this study.

6. Fitness Function

This analysis uses hitpoints as the primary outcome measure to determine fitness. We first define if the battle is a win, tie, loss, or unavailable (the party pokemon is not available for use yet at that point of the game). Then we measure either one minus the percentage of hp lost for the player pokemon (in the event of a win) or the percentage of hp lost for the opposing pokemon (in the event of a loss). This provides a continuous measure to compare various pokemon to one another. However, this alone will not yield a logical fitness value, since wins and losses are being compared on the same continuous scale. Furthermoer, we want our objective function to prefer wins over ties, over losses, over unavailability. This will force the algorithm to prioritize the four outcomes appropriately. To accomplish this, we borrow a strategy from linear programming called "Big M". By increasing (through addition) ties by M, losses by M * M, and unavailability by M * M * M (M = 1000), we adjust the scale of each category such that they cannot intersect. That is to say, if a party barely wins all battles (~0.99 for each of ~800 trainers), and another party looses even one battle (score of at least 1000000), then that one loss will always result in that party being scored worse than the party with all wins, regardless of how bad the wins are. At the same time, wins are still comparable with wins, and losses are still comparable with losses. This means that in the event of an unwinnable battle, the algorithm will still attempt to optimize for the best possible outcome for that encounter. Such delineation of scale between different outcome categories is crucial for effective optimization.

As stated previously, the battles are simulated using the hitpoints, speed, and damage dealt for both the party and encountered pokemon. Using this information, we can mathematically define the fitness of a given party pokemon against an encountered pokemon as follows:

Given...

- candidate i and encounter e with stats $(H_i, S_i, D_{i\to e})$ and $(H_e, S_e, D_{e\to i})$ respectively

- H = HP, S = Speed, D = damage per hit (directional by matchup)

- availability $\alpha_{ie} \in \{0, 1\}$ (1 = available)

- $M := 1000$.

Speed tie adjustments:

$$s := \operatorname{sgn}(S_i - S_e) \in \{-1, 0, +1\}, \qquad D'_{i\to e} := \begin{cases} D_{i\to e}, & s \neq 0 \\ D_{i\to e}/2, & s = 0 \end{cases}, \qquad D'_{e\to i} := \begin{cases} D_{e\to i}, & s \neq 0 \\ D_{e\to i}/2, & s = 0 \end{cases}.$$

Hits to KO (under adjusted damage):

$$k'_e := \left\lceil \frac{H_e}{D'_{i\to e}} \right\rceil, \qquad k'_i := \left\lceil \frac{H_i}{D'_{e\to i}} \right\rceil.$$

Outcome predicates:

$$\begin{aligned} \text{Win}: &\quad (s > 0 \ \wedge \ k'_e \leq k'_i) \ \vee \ (s < 0 \ \wedge \ k'_e < k'_i), \\ \text{Tie}: &\quad (s = 0 \ \wedge \ k'_e = k'_i), \\ \text{Loss}: &\quad \alpha_{ie} = 1 \ \text{ and not } (\text{Win or Tie}). \end{aligned}$$

Final HPs (used only in the indicated outcomes):

$$H_i^{\text{final}} = H_i - \left(k'_e - \mathbf{1}_{\{s>0\}}\right) D'_{e\to i} \quad \text{(used in Win)}, \qquad H_e^{\text{final}} = H_e - \left(k'_i - \mathbf{1}_{\{s<0\}}\right) D'_{i\to e} \quad \text{(used in Loss)}.$$

Then single-battle fitness can be defined as:

$$a_{ie} := \begin{cases} M^3 & (\text{unavailable, } \alpha_{ie} = 0) \\ M & (\text{Tie}) \\ 1 - \dfrac{H_i^{\text{final}}}{H_i} & (\text{Win}) \\ \dfrac{H_e^{\text{final}}}{H_e} + M^2 & (\text{Loss}) \end{cases}$$

Feel free to view the repository for more details as to how these steps are programatically implemented and executed.

7. Optimization

Now that both the objective function and individual fitness function have been defined, we can simply calculate the fitness of each combination of 6 party pokemon via the objective function and identify the best combination, right? In theory, yes. However, this is easier said than done due to the binomial nature of combinatorial optimization.

Calculating each individual fitness value of each separate pokemon against each encountered pokemon is actually quite asy. For example, in gen 1, there are 81 evolutionary chains (rows) compared against 644 encountered pokemon (columns). This makes a table with 52,164 values to calculate. While this may sound like a lot, to a computer such a task is trivial. However, calculating the fitness values for each party combination of 6 evolutionary chains is another story.

Consider the binomial coefficient (also commonly referred to as n choose k):

$$\binom{n}{k} \;=\; \frac{n!}{k!\,(n-k)!}, \qquad 0 \le k \le n.$$

This formula defines the number of different combinations of k choices out of a total pool of n options.

Within the context of this study, the value of k is always fixed at 6 (since our party can never exceed 6, and any additional party members will always yield an equivalent or better result). However, n can vary from generation to generation. Pokemon Red version has 73 available evolutionary chains, Crystal has 124, and Emerald has 125. Due to the explosive nature of the binomial coefficient, even small inputs like these can result in unexpectedly large numbers of potential combinations. For example, red version has over 218 million potential combinations to choose from (218,618,940). Crystal and emerald both have over 4.4 billion and 4.6 billion (4,465,475,476 and 4,690,625,500)

respectively. Notice how adding even one additional option from crystal to emerald added over 220 million additional combinations! Depending on the level of program optimization and computing resources available, making this many calculations could take anywhere from days to weeks to complete.

So how can the efficiency of this process be improved? While small to moderate gains can be made through computational optimization methods (including vectorized computing, parallel processing, batch computing, multithreading, etc.) the greatest gains are typically seen through reduction of the search space. If we know (or reasonably suspect) that the optimal solution is within some sub-section of all possible combinations, then the required number of combinations to search can be effectively reduced by skipping unnecessary combinations. Such reduction is generally accomplished through the use of optimization algorithms. These algorithms can be deterministic or non-deterministic in nature, and can reduce the search space anywhere from 10x to under 10000x the original size depending on the circumstances. While many optimization methods exist, this report will discuss the following: Pareto dominance filtering, genetic algorithms, branch and bound, and mixed integer linear programming (MILP).

> *Note: It is worth noting that certain restrictions were at play during the optimization of these fitness values. Such restrictions included the restriction of one starter pokemon per party, one Eevee evolution per party, and exclusion of specific pokemon (Shedinja). These will be discussed in more detail later in the report, in the "Limitations of Branch and Bound" and "Perfect information assumption: Moveset" sections.*

Pareto Dominance Filtering

Pareto Dominance Filtering is a general approach for search space reduction that can be combined with nearly any optimization method. The principals behind it are simple: If one option is better in every way than another option, then by definition the strictly worse option cannot be in the optimal solution. The strictly better option would always supercede it. This allows for simple filtering of pokemon out of the total pool if their individual fitness values are strictly worse for all columns than that of another pokemon.

A word of warning: While pareto dominance filtering may seem simple, things can quickly go wrong if accompanying constraints and/or restrictions are not properly accounted for. This is discussed in detail later in the "Limitations of Branch and Bound" section.

Genetic Algorithms

Genetic algorithms are an optimization algorithm based on the theory of evolution by charles darwin. This method is deceptively simple: Compare a small subset of combinations, and keep the top performing ones. Then duplicate them, while making random revisions to the new duplicates. These revisions can be small or large on average, depending on the defined mutation rate. This method localizes the search algorithm to the surrounding search space around top performing combinations, effectively reducing the search space. While this method is often effective, it does have limitations. First, it can be susceptible to local minima. If two good combinations exist far apart from one another

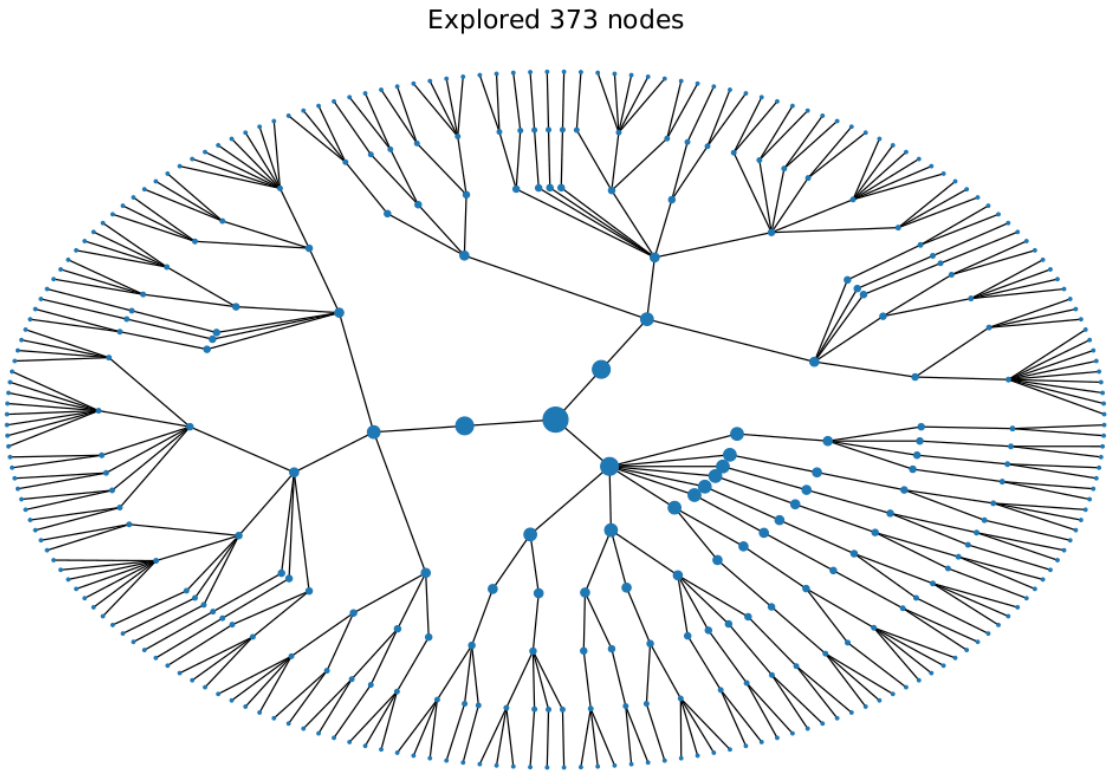
(meaning they are quite different from one another) then it can be easy for the algorithm to get "stuck" searching around one combination, and will never check the other. Another limitation is that genetic algorithms are not an exhaustive search. That is to say, they can never guarantee the optimal solution will be returned. Of course, the liklihood that the optimal solution is found can be increased through proper hyperparameter tuning. But by definition, the only way to know that you have the best solution is to check all possible solutions, or to mathematically eliminate non-optimal solutions through verifiable methods. Genetic algorithms only search a cross-section of possible solutions.

Branch and Bound

Branch and bound is another popular optimization method. In fact, this is the method utilized by Odland in his previous pokemon optimization study referenced earlier in the analytic approaches section [^5]. Granted, his simplified method did not require any optimization method at all (his search space of 593,775 combinations could be searched in 18 seconds). But he decided to utilize branch and bound optimization anyways, reducing the search time from 18 seconds down to 0.4 seconds. This represented a 45x reduction in search space.

In this context, the branch and bound algorithm works by sequentially searching each party position, from 1 to 6. It begins with position 1, then branches out to consider each possible addition for position 2, considering the additional benefit. It then continues searching and evaluating each of these positions for each branch until it has searched all branches, representing all possible combinations of 6 pokemon. If this sounds like an exhaustive search, that's because it is. Or rather, it would be, if not for the key feature of branch and bound: bounding (hence the name "branch and bound"). Think of bounding like a sort of pruning of bad branches. If we can mathematically determine the best possible solution by continuing to search that branch, then it may be possible to determine when a branch can no longer result in an optimal solution, even given the best-case scenerio.

Figure 4: A visual representation of branch and bound (Courtesy of Tommy Odland)



Applying Branch and Bound to this Analysis

In the context of this particular analysis, bounding is not possible in a strict sense. This is because any pokemon could, theoretically, have a perfect fitness (0) against all other encounters, effectively reducing the overall party fitness to zero at any time. Instead, the bounding for this analysis is performed differently. It relies on two assumptions:

1. The ideal solution consists of all wins

2. Some encounters have few pokemon that can win, either due to difficulty or lack of early-game pokemon availability

With these two assumptions in mind, the following bounding approach becomes feasible:

1. Identify any encounters (if any) in which there is no winning party option. Then for each of these encounters, identify the largest common order-of-magnitude denominator. For example, if the best result is a tie (a score of 1000), then the denominator would be 1000. If the best score was 1000000.234, then the denominator would be 1000000. We then subtract the common denominator from all pokemon against that encounter. This allows for ties or losses to be compared on the same 0 to 1 scale as wins if no wins are available.

2. Identify the column with the fewest "winning" scores (between 0 and 1). Assuming the optimal solution achieves all wins, then one of these identified pokemon must be present in the optimal solution. These pokemon will be candidates for position one.

3. For each winning pokemon identified in step 2, identify the column for each of those pokemon that has the fewest wins. Each of these pokemon will be candidates for position 2, when paired with the parent position 1.

4. Repeat steps 2 and 3 until either 6 pokemon are reached, or the party achieves an all-winning score sooner. In the event of an early all-winning scenerio, the remaining options can be any pokemon.

5. After identifying all potential all-win solutions from steps 1 through 4, remove all other combinations which do not achieve an all-winning score.

This algorithm resulted in the following search-space reductions:

Generation	Original Combinations	Reduced Combinations	Reduction Factor
Gen 1 (Red)	170,240,452	484	371,715.8×
Gen 2 (Crystal)	4,249,404,082	15,698,807	270.7×
Gen 3 (Emerald)	4,465,475,476	357,043	12,506.8×

The computational gains from this approach are substantial...at least in theory. One of the best features of this approach is that it guarantees the optimal solution will be found each time (assuming proper bounding is applied). Compare this to genetic algorithms, which are not deterministic. However, this approach was ultimately not utilized after identifying several key limitations.

Limitations of Branch and Bound

While this method initially seemed to work well, significant limitations were identified which involved the exclusionary criteria of the analysis. When determing the optimal pokemon party, there are select sets of pokemon of which only one can be chosen out of that set. The obvious example is starter pokemon sets: A trainer may only choose charmander, bulbasaur, or squirtle. So any optimization solution that includes more than one is considered invalid. This raised concerns with regard to the proposed pruning and bounding methods. First, pareto-dominance can be easily violated. If pokemon are dominated (and ultimately excluded) as being strictly worse than squirtle, but charmander is needed for the optimal solution, then these exclusions no longer make sense. The assumptions regarding search space reduction through identification of all-win scenerios are also similarly violated. For example, squirtle could be the only winning solution for one column, while charmander is the only winning solution for another. In such a case, an all-win solution would technically exist, but in reality would not since both pokemon cannot be in the same party. This limitation could be worked around through revision of the bounding methods and rules. Although this proposed method was not rigorously explored, it is believed that a working solution would involve creating distinct pokemon groups for each combination of unique exclusionary sets, and exploring each through a separate branch-and-bound. (For example, in gen 1 there would be 9 distinct populations to explore: 3 for each starter pokemon, multiplied by 3 for each eevee evolution. So one for bulbasaur + jolteon, another for bulbasaur + flareon, etc.) Due to the estimated increase in computational complexity for this method, a mixed integer linear programming method was utilized instead.

Mixed Integer Linear Programming (MILP)

Mixed-integer linear programming can be best described as a “tell the solver what you want, not how to get it” approach. You describe your problem with decision variables (some forced to be whole numbers to model yes/no or pick-k choices), linear rules those variables must satisfy (constraints), and a single score to minimize or maximize (objective). Once the problem is written this way, a MILP solver does the heavy lifting: it first relaxes the “must be whole numbers” rule to get a fast, optimistic bound on the best possible score, then uses smart search (branching) plus powerful math tricks (cuts and heuristics) to rule out huge swaths of impossible or inferior options without enumerating them all. If a feasible integer solution is found, the solver keeps tightening bounds until it can certify that nothing better exists, giving you a provably optimal, repeatable answer. Conceptually, it’s like combining the precision of an exact method with the efficiency of pruning, but you rarely hand-craft the pruning yourself; you focus on modeling your choices and trade-offs with linear relationships. The main trade-offs are that 1. you must express logic and costs linearly, which sometimes requires clever reformulations, and 2. very large models can still be hard. But for many realistic sizes, MILP delivers optimal solutions far faster than brute force and with stronger guarantees than heuristics.

What is a "Solver"?

A solver is the engine that takes your problem description (variables, constraints, and an objective) and figures out the best allowed answer without you telling it which steps to try. Think of it like a super-charged GPS for math: you enter the map (rules) and destination (goal), and it explores smartly under the hood, testing possibilities, relaxing hard requirements to get optimistic bounds, pruning routes that can’t possibly beat the best one found so far, and using proven algorithms (like simplex, interior-point, branch-and-bound, cuts, and heuristics) to move quickly toward optimal. It returns a concrete solution (the chosen values for your variables) and often a certificate of quality, either “proven optimal,” or “within X% of optimal,” or “infeasible” if no solution can satisfy the rules. You control when it stops (time limits, gap targets), but you don’t micromanage the math; the whole point is that modeling and solving are separate, so you focus on describing the problem and the solver focuses on finding the ideal solution efficiently.

MILP Methods

Given…

- Rows (candidates): $i \in \{1, \ldots, n\}$

- Columns (encounters): $j \in \{1, \ldots, m\}$

- Penalty scale: $M := 1000$

- Cost / fitness matrix: c_{ij} comes from the single-battle fitness a_{ij} (defined below), which lies in

$$a_{ij} \in \begin{cases} [0, 1] & (\text{win}) \\ \{M\} & (\text{tie}) \\ [M^2,\, M^2 + 1] & (\text{loss; offset by } M^2) \\ \{M^3\} & (\text{unavailable}) \end{cases}$$

Decision Variables:

- $x_i \in \{0, 1\}$: choose row i.

- $z_{ij} \in [0, 1]$: column j is "covered" by row i (the row that sets the column minimum).

> *Note: We can relax z_{ij} to continuous; at optimum they are integral because costs are positive and we enforce exactly one $z_{\cdot j}$ per column.*

Objective Function:

$$\min \sum_{j=1}^{m} \sum_{i=1}^{n} c_{ij}\, z_{ij}.$$

This works because, for each column j, $z_{\cdot j}$ places all mass on the chosen row with smallest c_{ij}.

Constraints

Pick exactly six rows:

$$\sum_{i=1}^{n} x_i \;=\; 6.$$

Each column is covered by exactly one chosen row:

$$\sum_{i=1}^{n} z_{ij} \;=\; 1 \qquad \forall\, j = 1, \ldots, m.$$

Only chosen rows may cover columns:

$$z_{ij} \;\le\; x_i \qquad \forall\, i = 1, \ldots, n,\ \forall\, j = 1, \ldots, m.$$

Forbidden rows:

$$x_i \;=\; 0 \qquad \forall\, i \in (\text{excluded set}).$$

At-most-one per exclusivity group G_k (e.g., $\{1, 2, 3\}$, $\{33, 34, 35\}$):

$$\sum_{i \in G_k} x_i \;\leq\; 1 \qquad \forall\, k.$$

Output. The solver returns $x^\star$ (the optimal 6 rows) and $z^\star$ (which of those 6 provides the column minimum for each column).

8. Results

Optimal Parties

The results of each generation are as follows (listed in order of aquisition):

Generation 1 (Pokemon Red Version)

Team Fitness: 12.584199

Generation 2 (Pokemon Crystal Version)

Team Fitness: 12.550387

Generation 3 (Pokemon Emerald Version)

Team Fitness: 19.590999

9. Discussion

Pokemon Archetypes

When assessing the results of this analysis, there were three common archetypes of pokemon that the algorithm appeared to favor: Early evolutions, snipers, and tanks.

- Early Evolutions: Comparing the results from all three generations, it is clear pokemon that evolve early are highly sought-after. This is fairly straightforward: Having a stage 3 pokemon with high stats early in the game poses a significant advantage. Anyone who has played Pokemon Red version knows how powerful having a Nidoking is before leaving Mt. Moon! Unconditional trade evolutions also fall into this category: Alakazam, Gengar, and Golem are all frequent choices throughout these three generations. Item-based evolutions, such as held-item trade evolutions and stone-based evolutions, vary more from generation to generation based on the availability of said items. Some insect types, such as Ledian and Beautifly, also have quick evolutions. These pokemon carry the early game, which is critical due to the lack of early pokemon and move variety.

Examples: Nidoking, Alakazam, Gengar, Golem

- Snipers: These pokemon generally have either high attack and/or special attack, paired with a high speed stat. Their goal is to take out the opposing pokemon as quickly as possible, ideally in one hit. These pokemon are far more likely to achieve a perfect fitness value (with zero hitpoints lost) than other pokemon. This also highlights the importance of the speed stat. If your pokemon is slower, it will never achieve a perfect outcome, since it will always take at least some amount of damage.

Examples: Alakazam, Dugtrio, Jolteon, Gengar, Fearow, Swellow

- Tanks: These pokemon tend to rely on high defense and/or special defense, paired with a high hitpoint value. Another form of taking comes from immunity to a particular type of move. For example, if a pokemon only has normal-type moves, then it cannot deal damage to a ghost type pokemon, and the ghost type will achieve a perfect score. Ground, ghost, and dark-type pokemon tend to be prioritized for this reason.

Examples: Golem, Gengar

It is also worth noting that almost all pokemon chosen were available relatively early into each game, generally before the third or fourth gym. The notable exception to this rule is Rayquaza in Gen 3. Rayquaza being selected highlights

the excessive difficulty of the endgame of pokemon emerald, requiring such a late-game addition to the party to offset challenging battles.

Assessment

Gen 1: Pokemon Red Version

The chosen party for generation 1 is very intuitive. The starter choice of squirtle makes sense. As a water type, it provides a distinct advantage against the first gym leader Brock, as well as the many other rock and ground types encountered early on in the game. It also has a higher defense, which is of greater utility against these encounters. Both squirtle and nidoran hold down the fort so to speak in the early game, until Alakazam can be acquired. Alakazam quickly becomes the go-to for the party, being easily one of the strongest pokemon ever in the pokemon franchise. Dugtrio, Jolteon, and Gengar are all acquired later to help fill in the gaps: Dugtrio against psychic and fire types, Gengar against normal types, and Jolteon against water types. Dugtrio is an interesting choice, considering the party already has Nidoking at the time of aquiring Dugtrio. As it turns out, Nidoking can't learn the dig tm. In fact, Nidoking can't learn any ground type moves until the earthquake tm is acquired towards the end of the game. And since dig is a 100 base power move in gen 1, it is incredibly beneficial to be able to use early. These are both contributing factors for Dugtrio's inclusion on the team. It is also worth noting that a dugtrio can be caught in Diglet cave at a high level relative to that point of the game. However, this is not taken into account during the analysis.

Gen 2: Pokemon Crystal Version

The chosen party for pokemon crystal is less intuitive then than the previous. First, is the choice of starter. Most casual players of the game consider either Cyndaquil or Totodile as the best starter in gen 2. Totodile has strong physical attack which serves it well early on, and Cyndaquil is by far the best fire type available for quite some time. Moreoever, fire-typing is invaluable against the many bug-types of the second gym, as well as the many grass types of sprout tower. But as it turns out, there are a considerable number of effective early options in crystal version. Pidgey, Ghastly, and Geodude are all exceptional early game options. Pidgey offers strong STAB normal and flying type moves with good speed, geodude offers a high defense with significant physical resistances and strong physical attacks, and ghastly is simply immune to normal damage, making it an auto win against many early game pokemon. Chikorita also learns razor leaf relatively early, which is effective against select encounters. A similar pattern to gen 1 is seen, where this group of early advantage pokemon are there simply to buy enough time until Alakazam can be caught. At which point Alakazam begins to carry most of the game by itself. Alakazam especially benefits from tms in gen 2, as it can learn fire punch, ice punch, and thunderpunch. Another puzzling choice for gen 2 is the decision to get both Pidgey and Spearow. It is worth noting that due to the presence of the return tm (a 106 base power normal type attack), the normal-type badge boost after the third gym, and normal-type boosting items, normal types are at a premium in gen 2. It is not at all surprising that Pidgey is caught and utilized early, and mostly abandoned once Spearow is acquired.

Gen 3: Pokemon Emerald Version

Pokemon Emerald is by far the most difficult of the three generations, as evidenced by its higher optimal party fitness, and the much lower differential between sequential max and original experience encountered (See Figure 19). This means that encountered pokemon are, on average, closer to the player's level than in other generations, making for more difficult encounters. The starter choice of Torchic may be unexpected, but it makes sense after considering it further. There is a serious shortage of effective fire type pokemon, and eventually Blaziken is able to utilize the high-power overheat tm to great effect. Alakazam can be acquired relatively early, again sweeping most of the remainder of the game once acquired. Tailow, Lotad, and Torchic again seem to be recruited simply to make it until Abra can be evolved into Alakazam, with Golem coming from Geodude shortly after. These five pokemon all synergize well with one another for the majority of the game. Towards the end of the game the sixth and final party option is acquired: Rayquaza. Rayquaza's primary goal is to beat up on the elite four and steven towards the end of the game. And he does quite a good job, being able to learn thunderbolt, earthquake, ice beam, and overheat.

All Generations: Best Pokemon Award

After seeing the chosen parties for all 3 generations, it should come as no surprise that Alakazam wins the award for all-around best pokemon from gens 1 to 3. His early availability, fast level 16 trade evolution, high speed and special attack, and powerful psychic typing and moves grand him a distinct advantage in each of these three diverse pokemon games. Generation 1 makes him even more powerful due to the combination of the special stats into a single stat. Generation 2 grants him a significant advantage due to the presence of fire, ice, and thunderpunch tms relatively early on. But in gen 3 he largely stands on his own merits, with the only real additional advantage being the presence of the shock wave tm. Well done, Alakazam!

Honorable mentions go to Golem and Gengar, who both appeared in two out of three optimal parties. This further supports the notion that unconditional trade evolutions are extremely powerful.

10. Limitations

There are too many limitations to this analysis to discuss them all here. Instead, some of the more important and interesting limitations are outlined below:

(Compromises/excluded features)

1. Non-damaging moves and effects

There are many moves which do not deal damage in pokemon. These include stat modifying moves, healing moves, and status inflicting moves, among others. In theory, an ideal simulation would account for all of these possilities during each battle. However, doing so would drastically increase the complexity of the simulation due to the butterfly effect. The butterfly effect refers to the different branching possibilities that can result from any single decision or outcome. When using a sleep-inflicting move, will the move hit, or not? Will the pokemon be put to sleep? If so, for how many turns? Would it be more beneficial to use swords-dance before attacking? Or just start attacking? Each of these possible scenerios would require a branching analysis of all possiblities for each battle. Compare this with the current approach, where we assume that only the best damaging move is used in each battle. This can be pre-calculated, and plugged into a formula along with the hp and speed values of each pokemon to immediately determine the outcome of each battle without needing to simulate anything. This ability to directly calculate the result of each battle is essential to the feasibility of this analysis, and is ultimately why we only consider damaging moves.

As a result, this places significant bias away from pokemon with powerful status-inflicting moves. For example, in gen 1 parasect is known for its signiture sleep-inducing move: spore. Spore has 100% accuracy, and puts the opposing pokemon to sleep. And in gen 1, sleep lasts anywhere from 1 to 7 turns! In many circumstances this can feel like an auto-win. But because this move is not considered, parasect has close to zero chance to be present in the optimal party. This is also true for many pokemon with powerful stat modifying moves, such as calm mind, swords dance, dragon dance, curse, or belly drum. Such moves can make a pokemon into a powerful sweeper, but are ultimately not considered in this analysis.

2. Consumable items

For similar reasons to non-damaging moves and effects, consumable items are also not considered in this analysis. They would simply be too difficult to track, and would greatly increase the computational complexity of the analysis due to the butterfly effect.

3. Moves and TMs

In an earlier iteration of this analysis, pokemon were split not only into their evolutionary lines, but also between the different types of moves they were allowed to learn. These different variations of evolutionary lines were referred to as "variants". This separated them into groups by what types of moves they could learn, up to four types per variant. The intention of this was to cut back on pokemon swapping back and fourth between five, six, or more moves between fight to fight. It was also meant to place less bias towards pokemon who can learn many different types of moves. But this came at the cost of significantly increased computational complexity. A pokemon with seven different types of moves, for example, would need to be split into 37 different variants. Doing this for each pokemon increased the number of potential choices, and thus number of possible combinations, drastically. In this earlier iteration, there were simply too many combinations to feed into a linear optimizer. Instead a genetic algorithm was utilized. In the end, the tradeoff between complexity and simplicity in this case ultimately ended up favoring simplicity. So the ability for more than four move types being available to each pokemon at a given time is acknowledged as a limitation of this analysis.

4. Perfect information assumption: Pokemon

In this partial simulation environment, the player has the agency to choose which of their six pokemon they will use against any other pokemon. We act under the assumption that the player will accordingly choose the best choice out of their six pokemon to use against each encounter. But the question remains: Is this a reasonable assumption? Would the player realistically know the best pokemon to use at all times? In most cases the answer would be somewhere between yes, and possibly. In fact, beyond the first pokemon, we actually get notified of what is coming next. We then have the choice to switch to a different pokemon. In this case, the player in fact does have full information. But this option of switching is not available for the first pokemon used. Say you are on the S.S. Anne, and you go up to fight a sailor trainer. What types of pokemon do you think he might start with? Almost certainly water-types. Therefore you would choose either your electric or grass-type pokemon to use against them. In this case, and in many cases, a reasonable assumption can be made as to what pokemon to start with. But not for all. Despite this, the simulation and subsequent optimization only works if the trainer posesses this agency to choose the best party pokemon against each encounter. So the assumption is made of perfect knowledge of the first (and every subsequent) pokemon encountered.

5. Perfect information assumption: Moveset

Taking the previous point a step further, the player has the agency to choose what pokemon to use against each encountered pokemon. But that choice will be largely dependent on what moves the opposing pokemon has. That is information the player simply does not have. How can we really know exactly how much damage the opposing pokemon will deal to us? If they are faster? In reality, we can never be able to perfectly answer these questions. So like with the perfect pokemon information assumption, we must also assume perfect information regarding the stats and moves of each encountered pokemon in order to accurately choose the most effective party pokemon to face them.

This assumption is necessary to the simulation and optimization components of this analysis. But it raises concerns due to potential edge-cases. Suppose you have a Chansey in your party. You have extremely high hitpoints and special defense, but your defense is paper then. Even a single physical attack will likely knock you out in one hit. You are facing off against a Kingler. Should you use Chansey? Does the Kingler have any physical attacks? Or only special attacks? There is no way of knowing for certain without perfect information. Because of this, having perfect information available tends to skew the results of the analysis towards hyper-situational pokemon such as Chansey. It could be argued that Alakazam would fall into a similar boat. Despite being incredibly fast and strong offensively, it is quite weak defensively. A single strong physical attack would likely knock it out. How do we know if our attack will be enough to finish off the opponent? Will they get a chance to attack us back? These are all valid questions to ask. But again, in order for this analysis to work this assumption still must be made, while accepting these limitations.

However, there is an even more extreme example of this edge-case, beyond even Chansey. That edge-case is Shedinja. Shedinja is a pokemon in generation 3 that has only 1 hitpoint. However, it has an incredibly powerful ability called wonder guard. This ability makes it so shedinja can only be damaged by types it is weak to: Flying, rock, ghost, fire, and dark type moves. Everything else deals 0 damage. This truley brings the question of perfect information into question. Such a pokemon will warp the entire optimization process around it. But when used in real life, many trainers will see it faint from an unexpected move from an opposing pokemon. And despite all of that, it simply would not be fun to have a pokemon that is esentially a game of rock paper scizzors with zero chance of failure. That is just not fun. For that reason, Shedinja was excluded entirely from the pokemon pool in this analysis. (It is worth noting, however, that even if shedinja is included, it is still not included in the final optimal party).

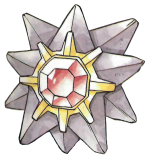
6. Early game difficulty

Some might argue that the weighting of this analysis is disproportionately skewed towards the early game. This is evidenced by the most difficult battle of each generation being from one of the first routes of the game. This early difficulty is in large part due to the extreme sparsity of early game pokemon availability. Some method of weighting could be applied to the algorithm to favor mid to late-game fitness, at the cost of increased complexity. This analysis instead favored weighing all battles equally.

11. Appendix

Most Difficult Encounters

Generation 1 (Pokemon Red Version)



5. Misty - Cerulean Gym - Starmie (Level 21) - Fitness 0.59 by Alakazam

4. Rocket - Mt. Moon - Raticate (Level 16) - Fitness 0.60 by Nidoking

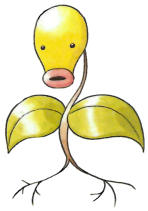
3. Rival (Green) - Indigo Plateau (Final Battle) - Alakazam (Level 59) - Fitness 0.67 by Alakazam

2. Bug Catcher - Viridian Forest - Caterpie (Level 6) - Fitness 0.82 by Squirtle OR Nidoran (Male)

1. Rival (Green) - Route 22 - Pidgey (Level 9) - Fitness 0.83 by Nidoran (Male)

Generation 2 (Pokemon Crystal Version)

5. Clair - Blackthorn City Gym - Kingdra (Level 40) - Fitness 0.648 by Pidgeot



4. Sage Troy - Sprout Tower - Bellsprout (Level 7) - Fitness 0.652 by Pidgey

3. Blue - Viridian City Gym - Alakazam (Level 54) - Fitness 0.67 by Alakazam

2. Falkner - Violet City Gym - Pidgeotto (Level 9) - Fitness 0.74 by Geodude

1. Youngster Joey - Route 30 - Rattata (Level 4) - Fitness 0.92 by Pidgey

Generation 3 (Pokemon Emerald Version)

5. Wallace - Pokemon League - Milotic (Level 58) - Fitness 0.72 by Ludicolo

4. Youngster Billy - Route 102 - Zigzagoon (Level 5) - Fitness 0.74 by Taillow

3. Youngster Allen - Route 102 - Zigzagoon (Level 4) - Fitness 0.79 by Torchic

2. Cooltrainer Katelynn - Victory Road - Slaking (Level 43) - Fitness 0.86 by Rayquaza

1. Youngster Allen - Route 102 - Taillow (Level 3) - Fitness 0.95 by Torchic

Generation Differences

Despite their similarities, significant differences exist between the first three generations of pokemon. Some of fhese differences are summarized in the following table:

Generation 1	Generation 2	Generation 3
Critical hit rate tied to speed	Held items introduced	Abilities introduced
Combined special stat	Badge type boosts	No badge boosts
No natural counter to psychic types	Dark and Steel types introduced	Difficulty

Critical Hit Rate

In generation 1, there is a significantly higher preference placed on pokemon speed than in other generations. Four of the 6 chosen pokemon have 100 or more speed (with Nidoking still having a respectable speed of 85). This is likely due in part to how critical hit rate is calculated. In most pokemon games, the baseline critical hit rate is constant at 6.25%, and either 25% or 12.5% for high critical hit ratio moves for gen 2 and 3, respectively. However, for gen 1 the critical hit rate was calculated via the following formulas:

For regular moves:

$$\left\lfloor \frac{\text{BaseSpeed}}{2} \right\rfloor$$

For high critical hit rate moves:

$$\min\left(8 \left\lfloor \frac{\text{BaseSpeed}}{2} \right\rfloor,\ 255\right)$$

This means that a Jolteon would have a 25% critical hit rate for all moves! And a Persian using slash would have a 99.6% critical hit chance! The increase in damage output for high-speed pokemon is significant in generation 1.

Combined Special Stat

In generation 1, another notable difference is that special attack and special defense is combined into a single "special" stat. This granted many pokemon a significant advantage. For example, Alakazam had 135 special in gen 1, but starting in gen 2 his special defense would be lowered to 85, a net reduction of 50 base stats. Likewise, chansey had 105 special in gen 1, but their special attack would later be reduced to 35 in gen 2, a reduction of 70 base stats!

Gen 1 Psychic Types

It is difficult to describe just how powerful psychic types were in generation 1 compared to other pokemon types. There were numerous reasons for this. First, psychic types tended to have the highest special stats among pokemon in the game. And since special attack and defense were combined into a single stat, this made these pokemon strong both offensively and defensively. Second, there were no resistances or super-effective types against psychic type in gen 1, except for bug type (which was considered extremely weak, both from a pokemon and moveset perspective). It is rumored that ghost type was supposed to be super effective against psychic type, but wasn't due to a bug. And even if it was, the only damaging ghost-type move in gen 1 anyways was lick, which only had 30 base power. For these reasons, psychic types were nearly unstoppable in generation 1.

Held Items

Held items were first introduced in generation 2. The most notable of them allowed pokemon to deal increased damage for a single type of move. These items allowed for increased offensive output in gens 2 and 3, helping to offset the decreased critical hit rate from gen 1.

Badge Boosts

Starting in gen 1 and continuing to gen 2, obtaining certain gym badges boosted the stats of the trainer's pokemon. This practice was ended in gen 3, however. What is notable is that in gen 2 there were also badge type boosts that were present, in addition to the existing stat boosts. This added yet another multiplicative type-based damage increase, further increasing the offensive damage output of pokemon in gen 2. It is possible that the lack of badge boosts in gen 3 contributed to the greater difficulty observed via the higher optimized team fitness score.

Abilities

Abilities were first introduced in generation 3. While the majority of them are inconsequential to this analysis, some are quite powerful. For example, intimidate always lowers the opposing pokemon's attack stage by 1. This may not seem like much, but in an optimization problem every advantage counts. Other abilities were impactful as well, but are too numerous to discuss here. Two notable abilities that required special consideration were truant and wonder guard. Wonder guard will be discussed in the limitations section. As for truant, this was accounted for through special logic in the battle simulation program.

Difficulty

While this is not an explicitly defined difference between these generations, it is a difference observed in the data. Figures 9, 14 and 19 represent the difference between the sequential max experience (the maximum experience at a given point of the game) and the actual experience of the encountered pokemon. In practical terms, this difference represents the experience difference (and thus level difference) between the player's pokemon and the encountered

pokemon. At the top of the chart, the experience is the same. As it goes down, this represents a greater experience and level difference, and thus easier battles. In other words, the larger the area is between these two lines, the easier the game is. Comparing these three figures, there is a noticable difference between gen 3 and gens 1 and 2. In gen 1, there is a significant drop after saffron gym. This drop is maintained on and off throughout the remainder of the game, ranging between 60,000 to 80,000 experience difference. Likewise in gen 2, after the first pokemon league battle, the remainder of the postgame is trivial, averaging around 120,000 exp difference. In contrast, in gen 3 the experience difference tends to hover around 25,000 experience difference, rarely ever exceeding 50,000. This difference in difficulty is confirmed by the roughly double fitness value for gen 3 relative to gens 1 and 2.

Alternate Analysis - No Trade Evolutions

Due to the obvious power of the trade evolutions (as evidenced by the frequent inclusion of Alakazam, Gengar, and Golem in the optimal parties), some might wonder how these optimizations would fare if we were to exclude trade evolutions? The following are the results for each generation without trade evolutions.

The results of each generation are as follows (listed in order of aquisition):

Generation 1 (Pokemon Red Version)

Team Fitness: 14.037523 (12.584199 with trade evolutions)

Differences: Gengar -> Gyarados, Alakazam -> Kadabra

Generation 2 (Pokemon Crystal Version)

Team Fitness: 15.798934 (12.550387 with trade evolutions)

Differences: Golem -> Graveler, Gengar -> Raichu, Alakazam -> Starmie

Generation 3 (Pokemon Emerald Version)

Team Fitness: 27.320856 (19.590999 with trade evolutions)

Differences: Alakazam -> Kadabra, Golem -> Regirock

Alternate Analysis Discussion

It is interesting to note that Kadabra was still present in gens 1 and 3, despite not being allowed to evolve into Alakazam. It was only in Gen 2 where a better alternative was found in Starmie. The switches from Gengar to Gyarados and Raichu in gens 1 and 2 respectively make sense. Haunter is simply not powerful enough on his own. He is also unable to learn many of the tms that Gengar can in gen 2. It was especially surprising to see Regirock being chosen over Graveler. Both pokemon are comparable, but it would seem that the additional power of Regirock outweighs the earlier availability of Graveler in this case. Lastly, it is interesting to note that while the fitness values for gens 1 and 2 did not change considerably (1.5 to 3 points roughly), for gen 3 it did change considerably (nearly 8 points). This is also a testiment to the difficulty of gen 3 relative to gens 1 and 2.

12. Additional Figures

Figures for Generation 1 (Pokemon Red Version)

Figure 5: Best Party Matchups Gen 1

Legend: Blue to red = Win (Blue = Better, Red = Worse), Yellow = Tie/Loss, Black = Unavailable

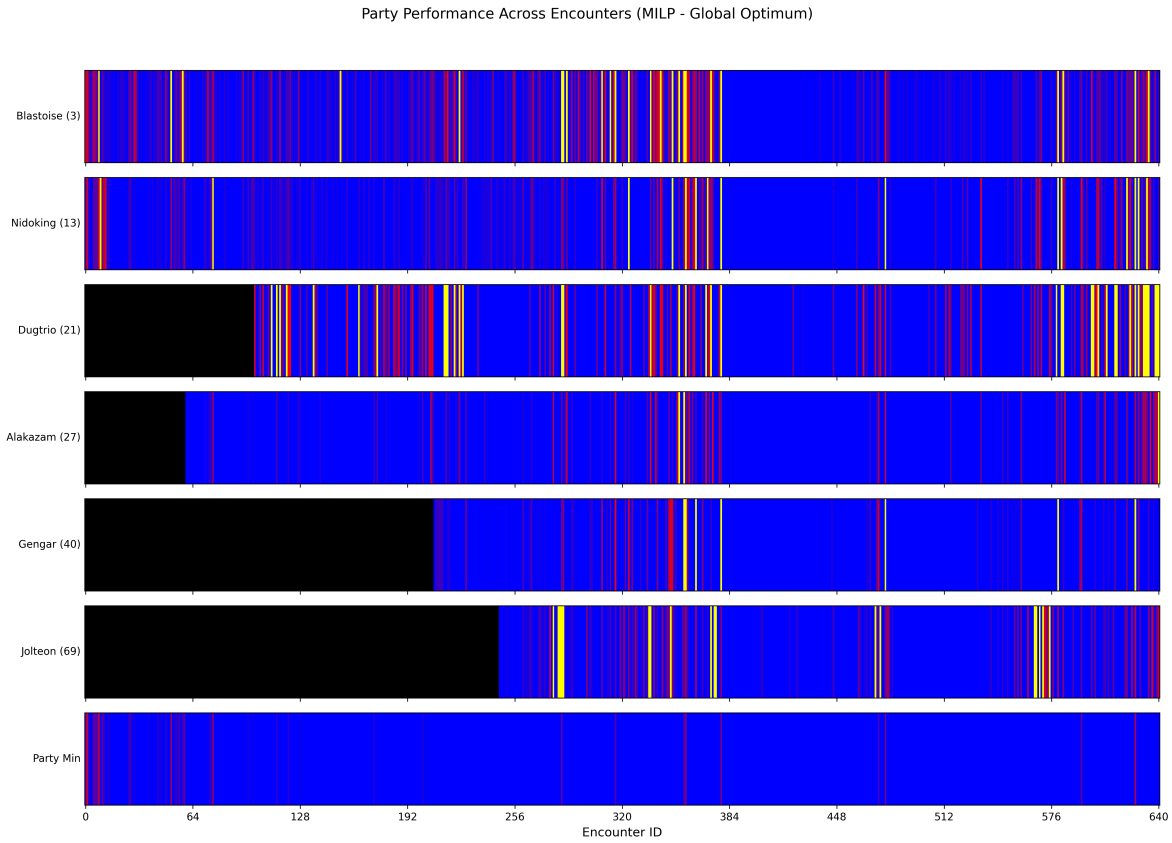


Figure 6: Best Party Matchups Gen 1 (Unique Best)

Legend: Blue to red = Win (Blue = Better, Red = Worse), Yellow = Tie/Loss, Black = Unavailable

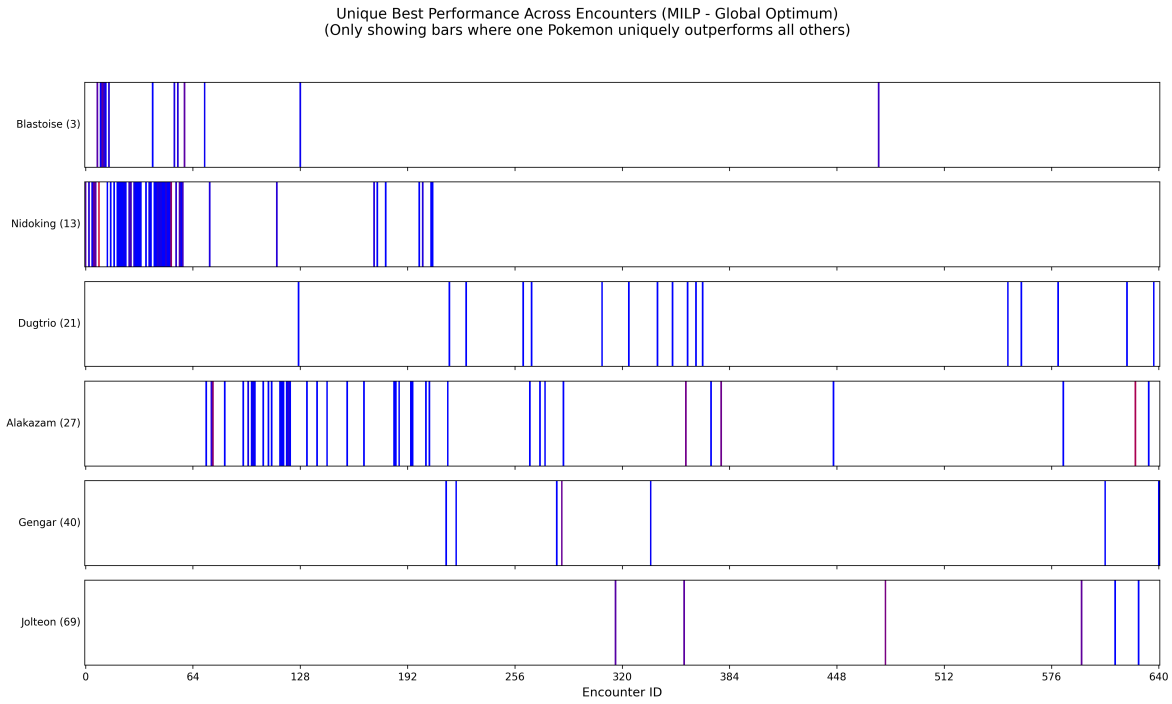


Figure 7: Experience Progression over Encounters Gen 1

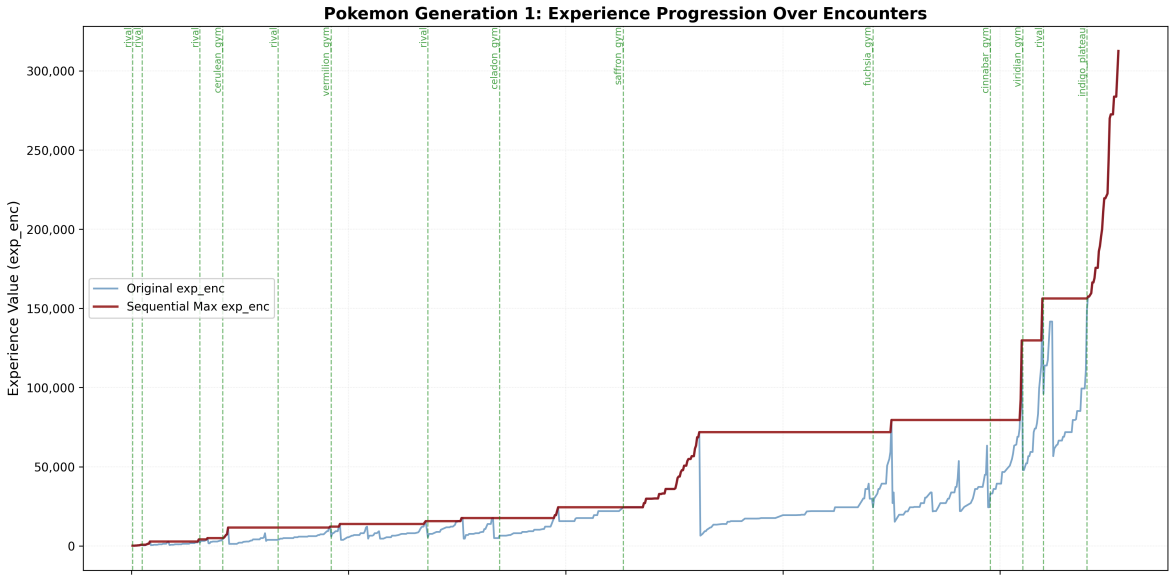


Figure 8: Experience Progression over Encounters Gen 1 (Log Scale)

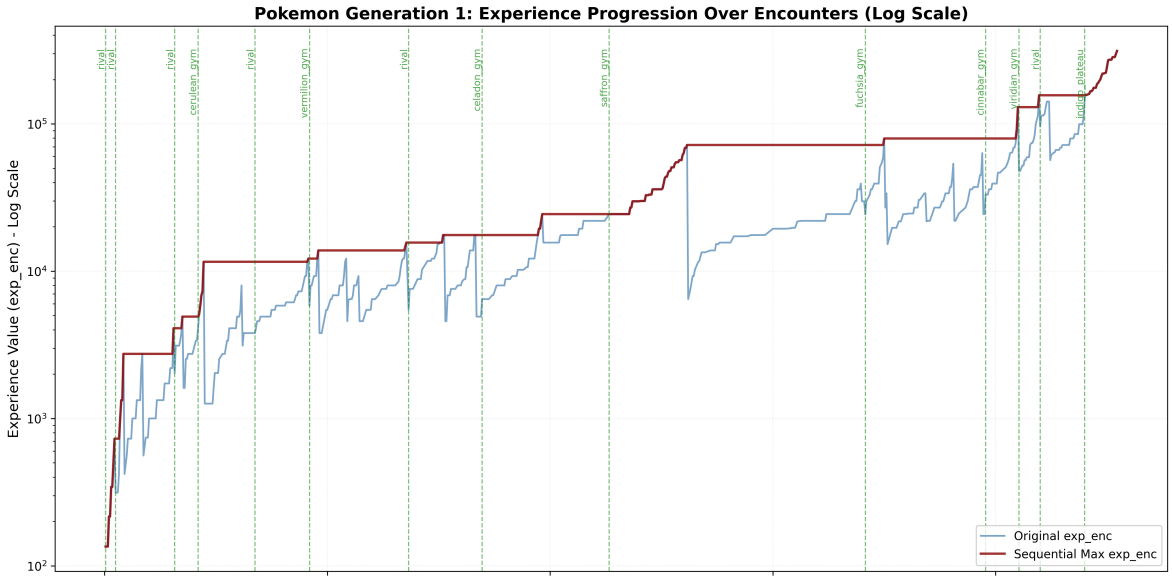
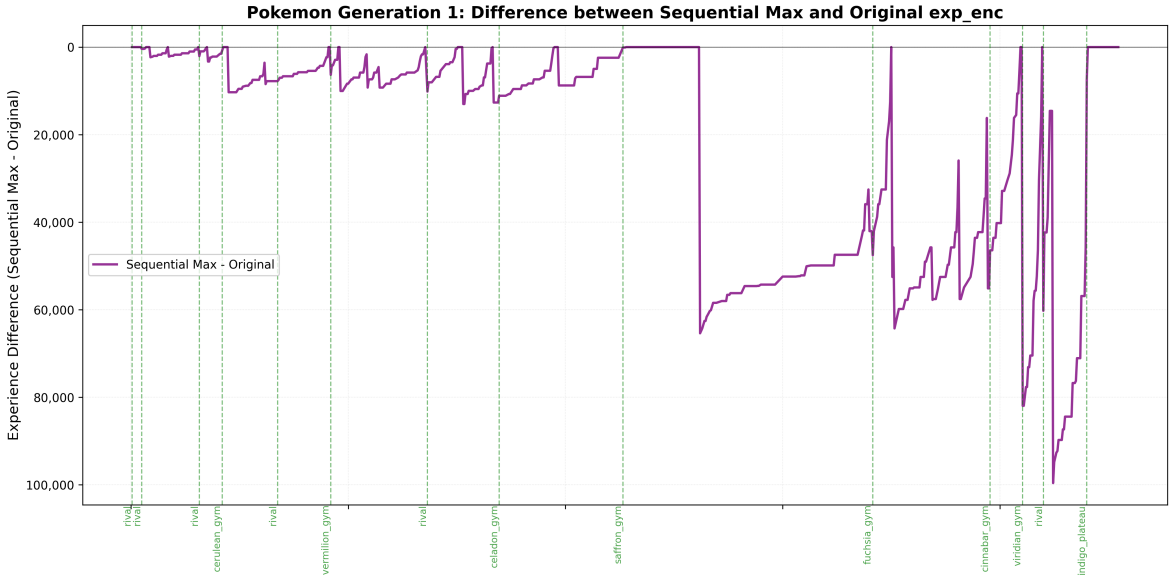


Figure 9: Difference Between Actual vs. Max Experience over Encounters Gen 1



Figures for Generation 2 (Pokemon Crystal Version)

Figure 10: Best Party Matchups Gen 2

Legend: Blue to red = Win (Blue = Better, Red = Worse), Yellow = Tie/Loss, Black = Unavailable

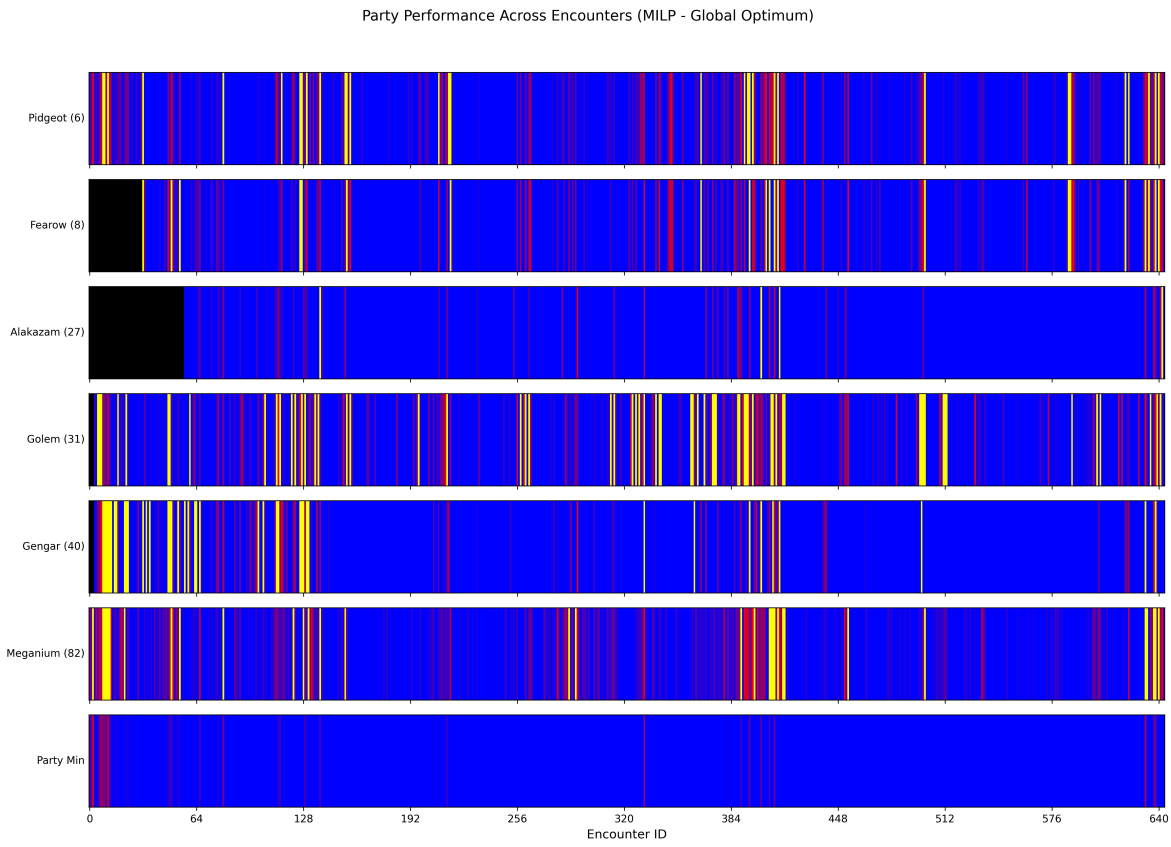


Figure 11: Best Party Matchups Gen 2 (Unique Best)

Legend: Blue to red = Win (Blue = Better, Red = Worse), Yellow = Tie/Loss, Black = Unavailable

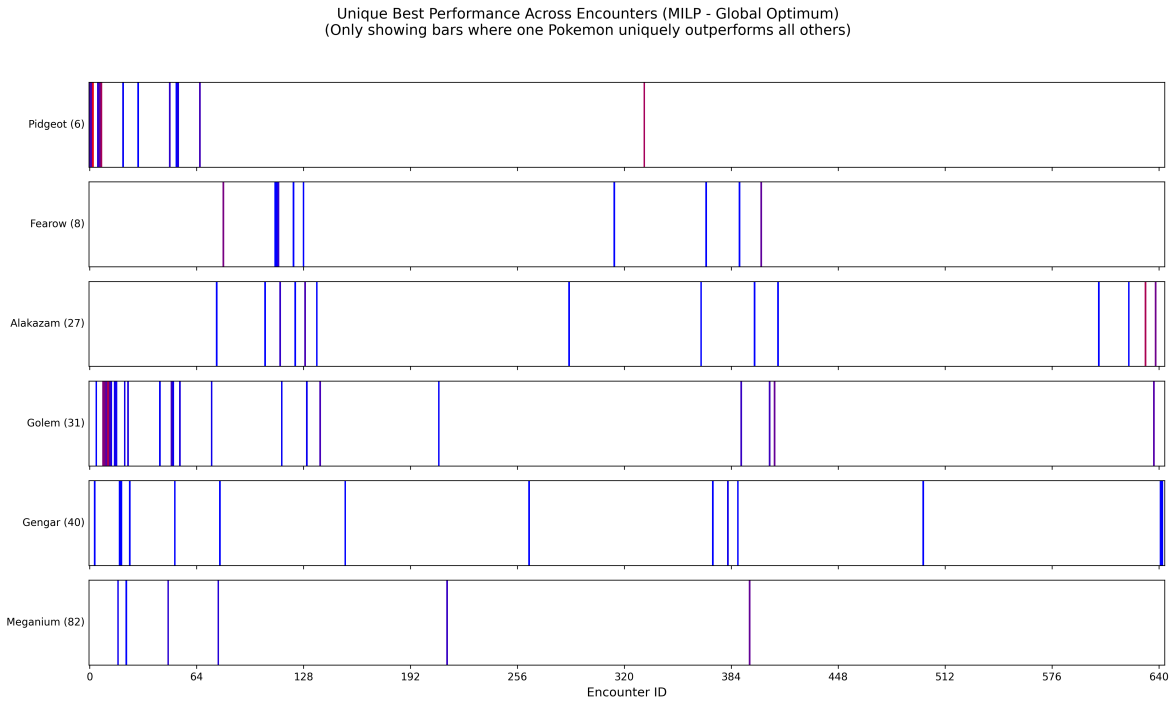


Figure 12: Experience Progression over Encounters Gen 2

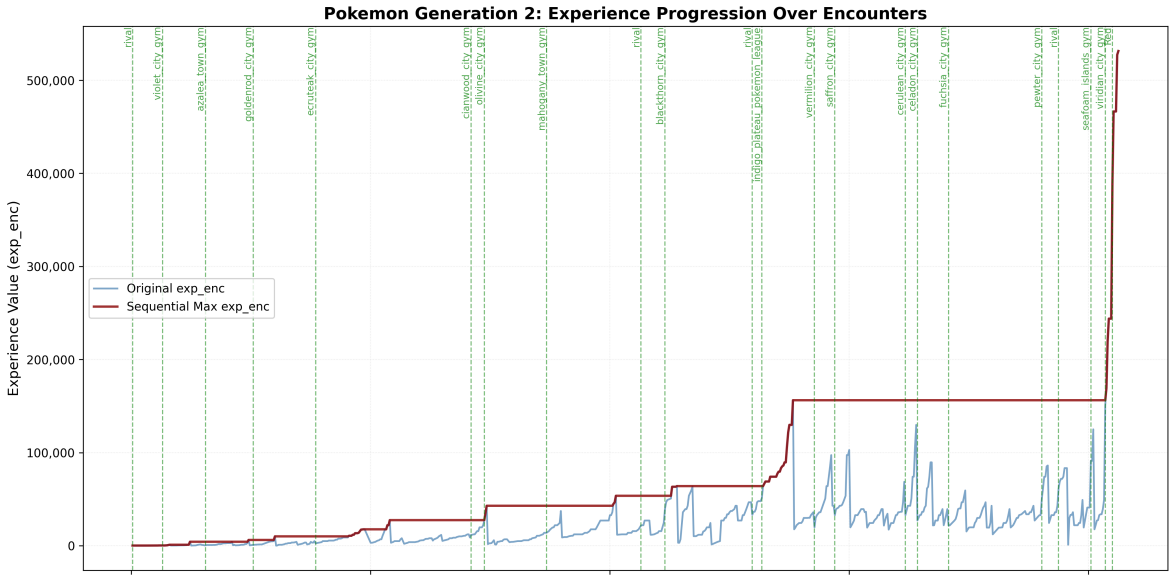


Figure 13: Experience Progression over Encounters Gen 2 (Log Scale)

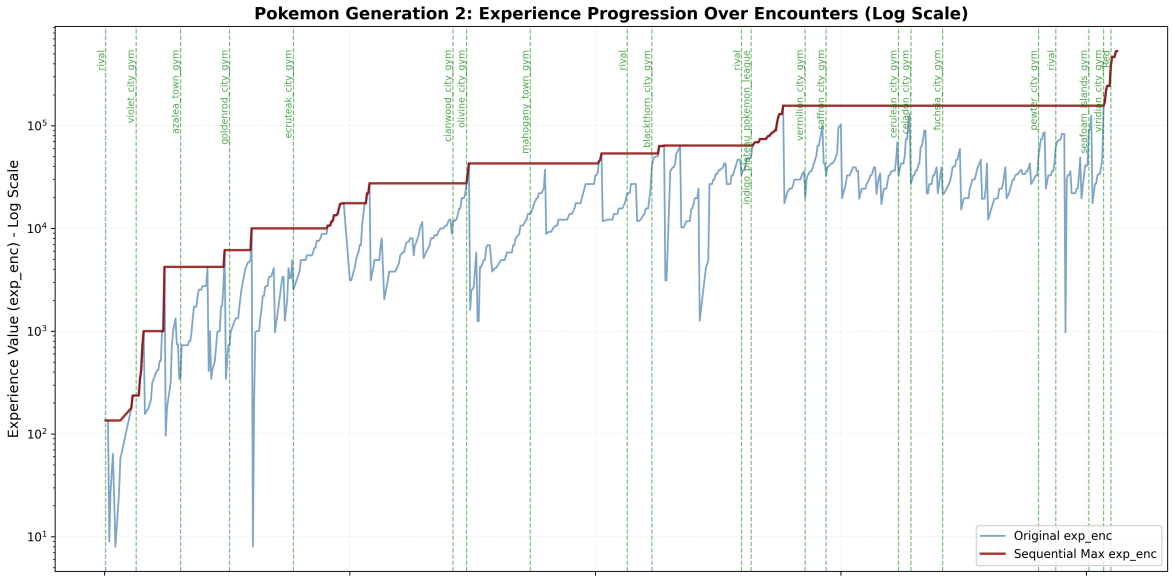
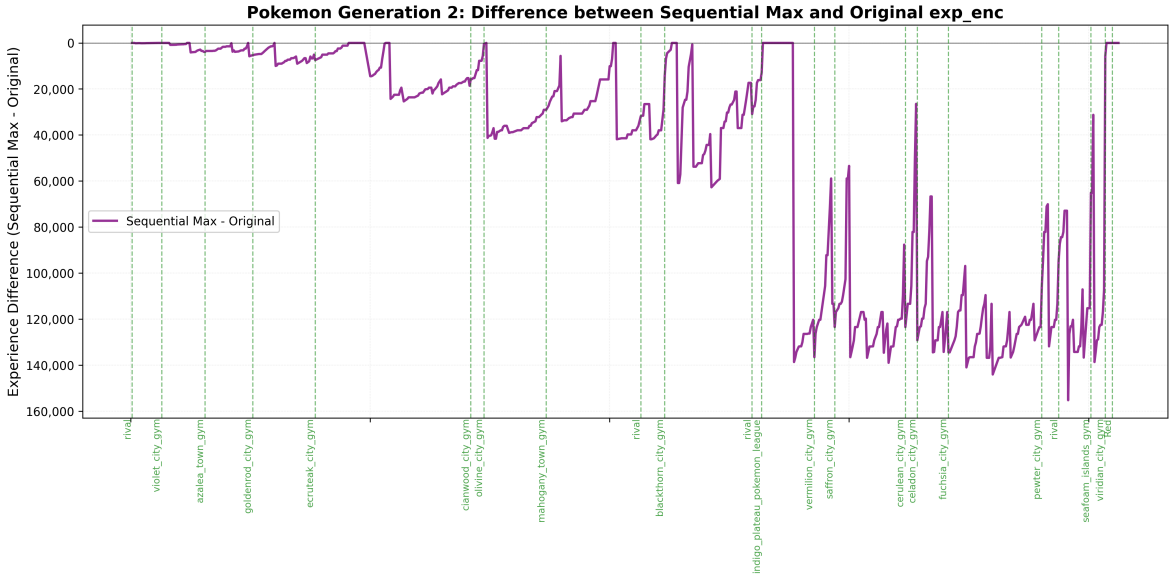


Figure 14: Difference Between Actual vs. Max Experience over Encounters Gen 2



Figures for Generation 3 (Pokemon Emerald Version)

Figure 15: Best Party Matchups Gen 3

Legend: Blue to red = Win (Blue = Better, Red = Worse), Yellow = Tie/Loss, Black = Unavailable

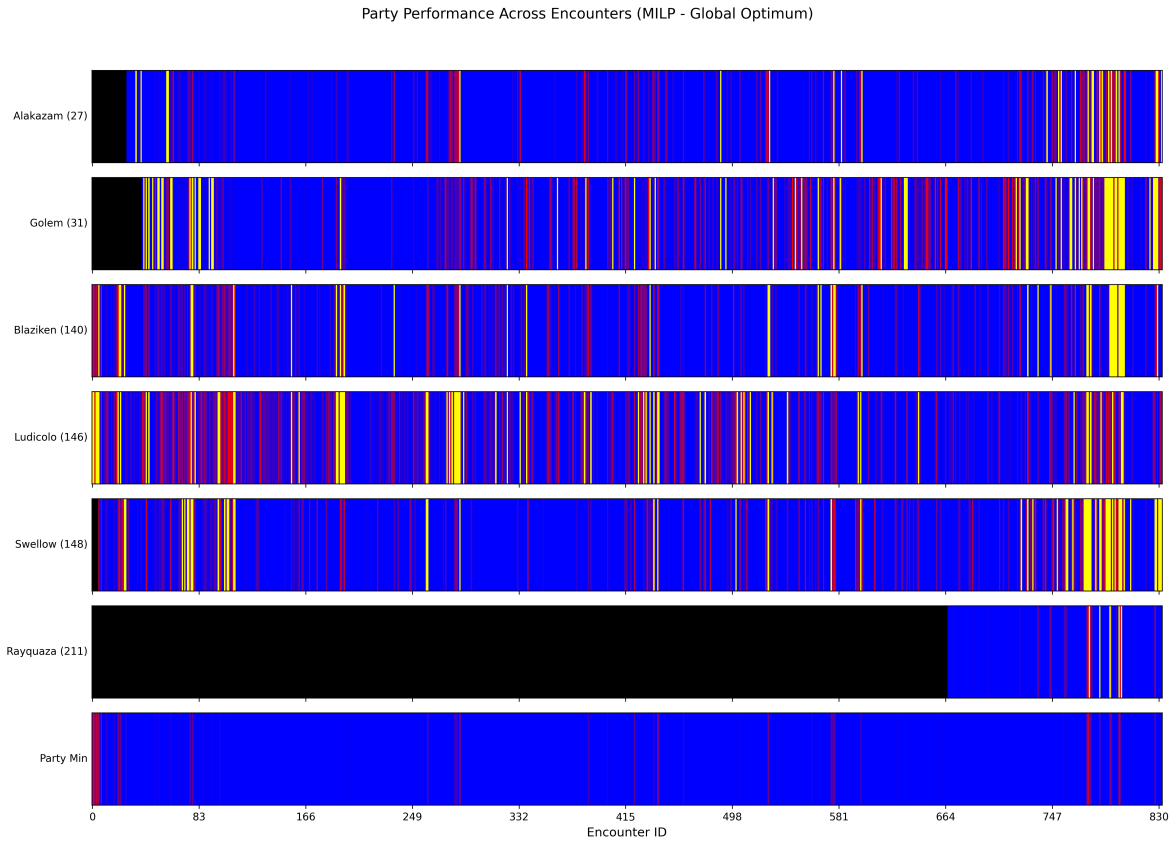


Figure 16: Best Party Matchups Gen 3 (Unique Best)

Legend: Blue to red = Win (Blue = Better, Red = Worse), Yellow = Tie/Loss, Black = Unavailable

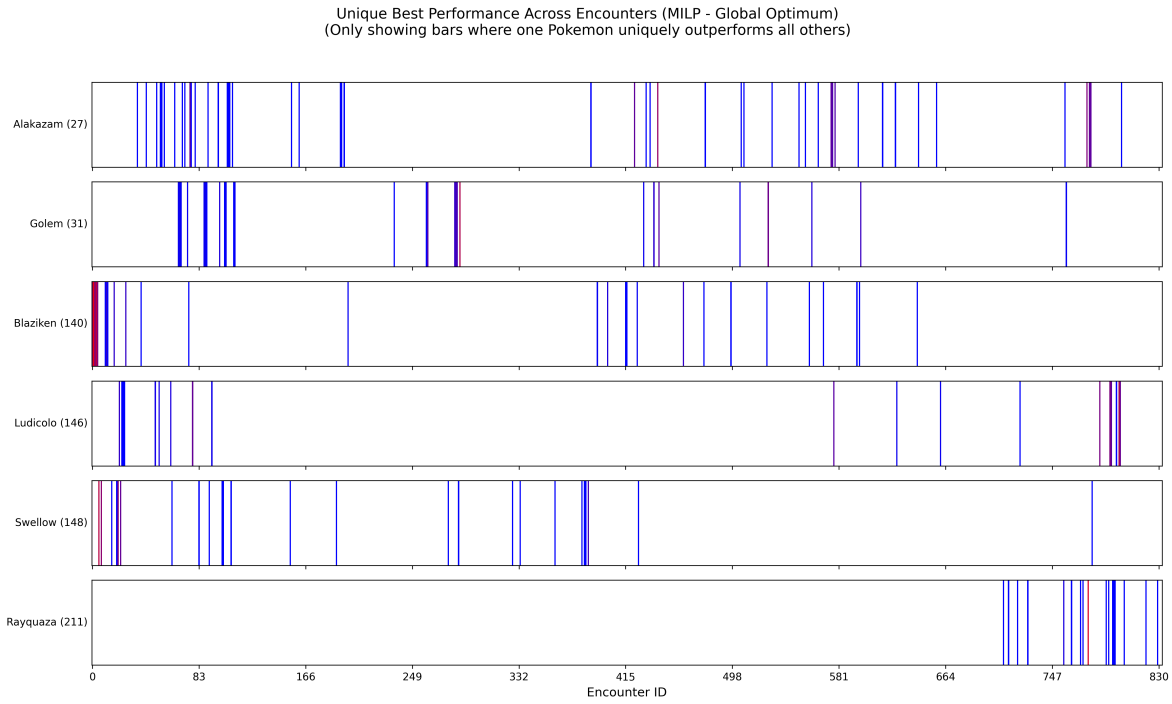


Figure 17: Experience Progression over Encounters Gen 3

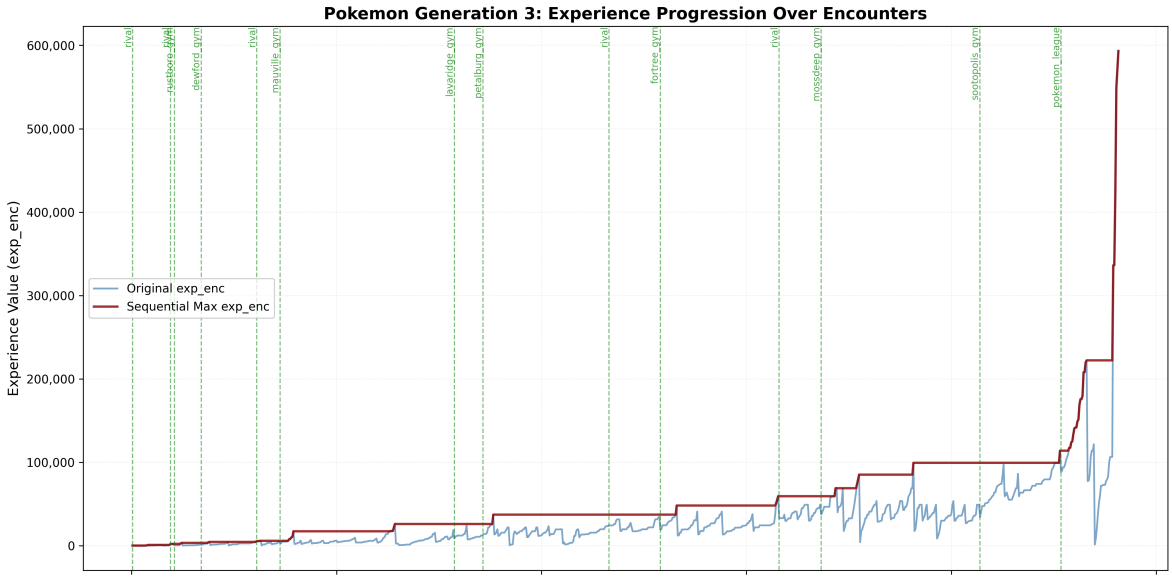


Figure 18: Experience Progression over Encounters Gen 3 (Log Scale)

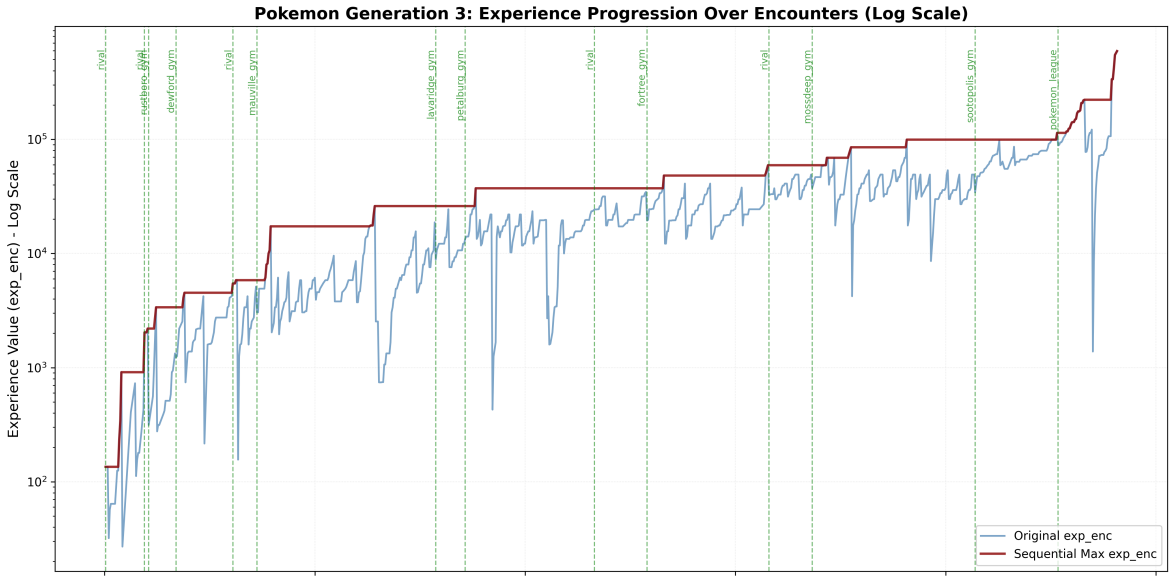
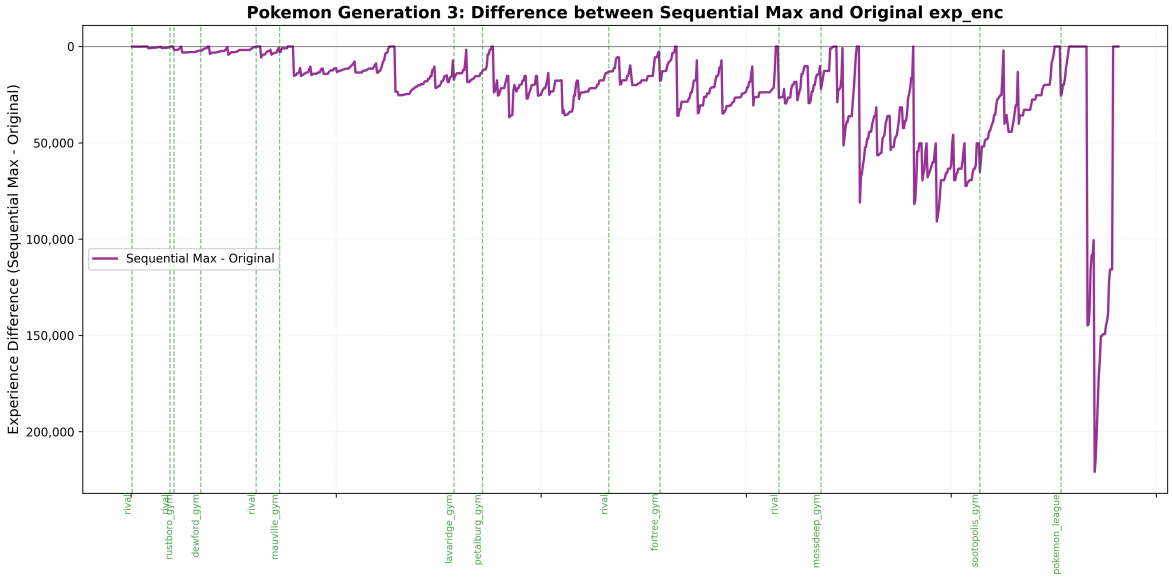


Figure 19: Difference Between Actual vs. Max Experience over Encounters Gen 3



References

[^1]: https://groups.google.com/g/alt.games.nintendo.pokemon/c/QodR2hhv4t4/m/AtyisHGVVdsJ?pli=1

[^2]: https://www.smogon.com/forums/threads/vgc-2025-regulation-j-metagame-discussion.3769052/

[^3]: https://www.merriam-webster.com/dictionary/heuristic

[^4]: https://www.merriam-webster.com/dictionary/deduction

[^5]: https://tommyodland.com/articles/2021/the-best-pokemon-party/index.html