Determining Optimal Solutions for Pokémon Go Gym Battles

Noah Lichtenstein and Nicholas Seidl
College of Computer and Information Science
Northeastern University,
Boston, USA

Abstract

Gym Battles in Pokémon Go consist of attacking Pokémon fighting defending Pokémon, where there exists an optimal attacking team for a given defending team. Greedy best first search (GBFS) and Monte Carlo Tree Search (MCTS) were used to determine optimal attacking Pokémon teams. Greedy best first search found valid solutions quickly, while MCTS found better solutions given more computation time.

Introduction

Pokémon Go is a game played globally, where players catch Pokémon and level them up. Pokémon Go Gyms are locations in the game where players can add their Pokémon to, and are consequently rewarded with in-game currency for defending that gym. Other players attack the Gym, using their Pokémon to fight the defending Pokémon, with the goal of replacing the existing defenders with their own.

Pokémon Go Gyms have multiple defending Pokémon, so a player attacking must choose multiple attacking Pokémon with the goal of defeating all of the defenders. Finding the "best" attackers is a non-trivial problem, due to many nuances with how Pokémon are constructed.

In order to solve this problem of finding a "best" set of attacking Pokémon, given a set of defending Pokémon, we implemented two algorithms: greedy best first search, and Monte Carlo tree search.

Background

State Formulation

In our formulation of the problem, a Pokémon Gym Battle consists of six defending Pokémon and six attacking Pokémon. Formally, a State in our formulation of the problem consists of the attacking Pokémon:

$$Attackers = \{ a_1 \ldots a_6 \} \, where \; a_i \in AllPokemon$$

Each State has an *Attackers* set of attacking Pokémon. *Attackers* consists of six variables, Pokémon a_1 through a_6. Each Pokémon/variable has a set of characteristics (such as health and base statistics, described later).

$$AllPokemon = \{ Bulbasaur \ldots Dragonite \}$$
$$|AllPokemon| = 144$$

The domain of each variable a_i is of size 144, because there are 144 different Pokémon. States impose no constraints on the domain of the variables: multiple a_i in the same *Attackers* set may be the same Pokémon.

A Gym Battle is the execution of a six versus six battle between an *Attackers* set and a similarly constructed *Defenders* set. Defending Pokémon are held constant so that the performance of different attacking sets can be evaluated against each other. The Gym Battle proceeds by pairing up attacking and defending Pokémon in order: a_1 vs. d_1 (say a_1 wins), a_1 vs. d_2 (d_2 wins), a_2 vs. d_2, etc.: a Pokémon stays as a team's contender until it's health hits zero. Health is not reset in between individual battles. Therefore, even though two *Attackers* sets may contain the same Pokémon, the order may be different, which will affect the outcome of the Gym Battle differently. The formula for determining the outcome of any one versus one battle will be explained after defining Pokémon, Moves, and Types.

Pokémon Data

A Pokémon has base statistics, moves (charge and quick), types, individual values (IVs), and a level. In order to appropriately simplify the Pokémon Gym Battles from the battles in the real game, we made the following assumptions: all Pokémon will have the same individual values (15 out of 15, for each of Attack, Stamina, and Defense) and level (40, the maximum possible level). These simplifications allow us to focus on the AI algorithms solving the problem, rather than small details that don't perfectly match the real game. All Pokémon Go game source data used in this project is from a public GitHub repository where game source code is extracted from the app (Barnes and Livio, 2018).

```
{
     "dex": 1,
     "stats": {
         "base_stamina": 128,
         "base_defense": 111,
         "base_attack": 118
     },
     "name": "Bulbasaur",
     "charge_moves": [
         "SLUDGE_BOMB",
         "SEED_BOMB",
         "POWER_WHIP"
     ],
     "quick_moves": [
         "VINE_WHIP_FAST",
         "TACKLE_FAST"
     ],
     "types": [
         "POKÉMON_TYPE_GRASS",
         "POKÉMON_TYPE_POISON"
     ]
 }
```

Figure 1: JSON representation of a Pokémon

As seen in Figure 1, a single Pokémon may have multiple quick moves and charge moves, and multiple types. A type, such as Water, identifies the Pokémon's specialty. When a Water Pokémon uses a Water move, the damage of that attack move is higher when used against a Fire Pokémon (which is weak to Water) than when used against a Grass Pokémon (which is strong against Water).

```
{
     "id": "POKÉMON_TYPE_GRASS",
     "damage": [
         {
             "id": "POKÉMON_TYPE_FIRE",
             "multiplier": 0.714
         },
         {
             "id": "POKÉMON_TYPE_GRASS",
             "multiplier": 1.4
         },
         {
             "id": "POKÉMON_TYPE_FLYING",
             "multiplier": 0.714
         },
         ...
}
```

Figure 2: JSON representation of a Type

As seen in Figure 2, when a Grass Pokémon attacks a Fire Pokémon, the damage of its attack is multiplied by 0.714: this means that Grass is weak against Fire. However, when a Grass Pokémon attacks a Water Pokémon, the damage of that attack is multiplied by 1.4: this means that Grass is strong against Water.

```
{
     "damage_start_ms": 900,
     "power": 130,
     "stamina_loss": 0.11,
     "duration_ms": 3300,
     "critical_chance": 0.05,
     "energy_delta": -100,
     "type": "POKÉMON_TYPE_WATER",
     "id": "HYDRO_PUMP",
     "accuracy": 1
 }
```

Figure 3: JSON representation of a Move

As seen in Figure 3, a Move has a Type (just like how a Pokémon has Types), an energy cost (how much energy is consumed upon using the move), and a duration (how long the move takes to execute).

Determining the Winner of a 1 vs. 1 Battle

At a high level, each one versus one battle has an attacker and a defender. The winner of a one versus one Pokémon battle is simply the Pokémon that survives the longest. The damage a Pokémon's Move deals is calculated as such (Omu and Q, 2016):

$$damage = floor\left(\frac{1}{2} \times power \times \frac{atk}{def} \times stab \times effectiveness\right) + 1$$

Equation 1: Damage of a Pokémon's Move

In Equation 1, *damage* is the total damage of a move, *power* is the base power of a move, *atk* is the attack statistic of the attacking Pokémon, *def* is the defense statistic of the defending Pokémon, *stab* is 1.4 if the Pokémon Type and Move Type match and 1 otherwise, and *effectiveness* is the effectiveness of the Move Type against all the Defending Pokémon's Types.

So, a Move's damage depends on several factors, one of which is the Pokémon that it is attacking. For example, a Bulbasaur would deal different damage against a Charmander than it would against a Squirtle.

To calculate the total damage per second that a Pokémon deals when using its moves, the damage per second and energy cost per second of each attack are combined (Felix, 2016):

$$dps_{total} = \frac{\left(dps_{quick} \times eps_{charge}\right) + \left(dps_{charge} \times eps_{quick}\right)}{eps_{charge} + eps_{quick}}$$

Equation 2: Total damage per second of a Pokémon.

$$dps_{charge} = \frac{damage_{charge}}{duration_{charge}}$$

Equation 3: Damage per second of a charge or quick move.

$$eps_{charge} = \frac{abs(energy_{charge})}{duration_{charge}}$$

Equation 4: Energy cost per second of a charge or quick move.

The pseudo-code for determining the total damage per second can be found in Appendix A.

The highest damage per second pairing of a quick move and a charge move is selected for both Pokémon in a one versus one battle. In order to determine the winner, the duration that it would take each Pokémon to eliminate the other is calculated, and the Pokémon that eliminates the other quicker is deemed the winner.

The winner of a one versus one battle is always deterministically decided. Ties, which occur when the attacking Pokémon and defending Pokémon are exactly the same and start at the same health, end in the Attacker's favor (however, both are eliminated). The pseudo-code for determining the winner of a one versus one battle can be found in Appendix A.

Algorithm Evaluation

Each algorithm used to determine a "good" *Attackers* set of attacking Pokémon, given a *Defenders* set of defending Pokémon, will be evaluated using the following metrics: compute time, attackers remaining, health lost, battle time.

> Compute time is the amount of time spent computing before a solution is produced; a lower compute time score is better.
>
> Attackers remaining is the number of attackers (an integer from 1 to 6 inclusive) left after the attacking Pokémon defeat the defending Pokémon; a higher attackers remaining score is better.
>
> Health lost is the sum total of health lost on the *Attackers* team over the duration of the Gym Battle; a lower health lost score is better.
>
> Battle time is the duration of the whole Gym Battle where the *Attackers* team defeats the entire *Defenders* team; a lower battle time score is better.

However, we did not come up with a single comprehensive function that combines the four evaluation metrics with respective weights. This is because different human players prefer to optimize for different things: if a player is in a rush, they may disregard how many potions they need to use to heal Pokémon and how many revives they need to use to revive eliminated Pokémon, and prefer a lower total duration over the other metrics. But, if they're not in a rush, the total duration may be totally irrelevant, and they'd prefer to minimize resources.

Searching for a Solution

Our initial solution to the problem involved implementing a Breadth First Search. One action/transition from a State to a different State is defined as the change of a single Pokémon, such as changing *Attackers*[2] from a Bulbasaur to a Charmander. The domain of each of variable in the Attacking set has a size of 144. So, there are $144^6 \cong 8.9$ *trillion* different unique Attacking sets.

Even if the evaluation of a full six versus six Gym Battle takes 0.01s, evaluating all the immediate successors of a state (all sets that are one substitution away, 6 slots × 144 options per slot = 864) would take about 8.64 seconds. Evaluation of the second ply (states that are two substitutions away) would take 864 successors × 864 successors × 0.01s evaluation time per successor ≅ 7465s ≅ 124 minutes ≅ 2 hours. Therefore, even if a winning *Attackers* set is only two substitutions away from the original guess, it would take maximally 2 hours to compute. The BFS algorithm is intractable due to how long it takes to arrive at solutions.

Related Work

A utility already exists to simulate one versus one Pokémon battles (PokéBattler, 2018). The simulation utilizes an adversarial Monte Carlo Simulation to determine the winner. However, it does not extend this functionality to six versus six Gym Battles. In addition, only about 200 simulations are run. Finding this motivated us to leverage Monte Carlo Simulation ourselves to simulate full six versus six Gym Battles.

Local Search algorithms such as Simulated Annealing and Hill Climbing could also be used to solve the problem. If we were to use a Local Search algorithm, we would first find a reasonably good initial Attacker set, and then continuously make Pokémon substitutions and re-evaluate the full Gym Battle outcome. However, the performance of these algorithms could be diminished since they converge only towards local maximums. There are many solutions in the state space, and therefore many local maximums.

Project Description

To search for solutions to the gym problem, we formulated the problem both as a basic greedy best first search (GBFS), and as several different implementations of a Monte Carlo tree search (MCTS).

Greedy Best First Search (GBFS)

Greedy best first search is a method for exploring a state space that derives its ability from how it chooses the next state to explore (Coles and Smith, 2007). Each state is assigned a heuristic value, which communicates that state's value relative to other states, where lower heuristic values are better. At a high level, the algorithm expands all successors of the current state, and then adds them all to a Priority Queue, where node priorities are the state heuristic scores. This way, successors with lower heuristic scores will always be processed before successors with higher heuristic scores. Figure 4 below shows the algorithm in more detail.

```
function GBFS(state, defenders)
  initialize explored set, E
  initialize priority queue, Q

  if state == terminal, return state

  priority ← H(state)
  Q.insert(priority, state)

  while not Q.empty do
    state ← Q.dequeue
    if state ∈ E
      continue
    E.add(state)

    if state == terminal, return state

    for successor in Successors(state) do
      if successor is terminal, return successor

      priority ← H(successor)
      Q.insert(priority, successor)

  return null
```

Figure 4: Pseudo-code of the greedy best first search algorithm.

In our formulation, states are simply complete assignments of six Attacking Pokémon. The heuristic used is *num_defenders_left*—after simulating the six on six Gym Battle, the defenders left will be a number between 0 and 6 inclusive, and lower is better. So, states and successors states are continuously generated and scored with the *num_defenders_left* heuristic, and the very first state encountered that passes terminal criteria (0 defenders left: *Attackers* beat *Defenders*) is returned. Unique states are never processed more than once; we ensure this by maintaining and checking a separate explored set.

The *num_defenders_left* heuristic is neither admissible nor consistent. This is because it may in fact overestimate how many states are in between the current one and a terminal state. For example, consider a state that has three remaining defenders left. If a substitution occurs where a really good Pokémon is swapped in, the successor state may immediately be terminal. This would mean the heuristic value of original state is 3, even though it is only 1 transition away from a terminal state, and because 3 > 1, our heuristic overestimates. In our case, a heuristic that is neither admissible nor consistent still produces valid results, because the number of defenders left stays constant or decreases, which brings our candidate *Attackers* sets closer to an *Attackers* set that completely eliminates the *Defenders* set (by eliminating all 6 defenders such that *num_defenders_left* = 0). The results from GBFS will be suboptimal due to the inadmissible heuristic, but will still be valid *Attackers* sets.

The general run time of this greedy best first search algorithm is *O(n)*, where *n* is the number of states in the state space. This is because even though at each step we evaluate all of the immediate successors, those successors get added to the work queue, and the work queue will never have a duplicate of an already processed state. Even though at first glance this runtime may appear none better than a simple linear search through all possible states, the ordering of states that are evaluated is significant, in that the states that are closest to being terminal (the ones with the lowest number of defenders left) are evaluated earlier than worse ones.

Monte Carlo Tree Search (MCTS)

MCTS is a method for exploring and expanding a tree by sampling the most valuable nodes through many "playouts," or simulations, of the search tree. Upon reaching a terminal state, MCTS back propagates the value of the result to the intermediate nodes visited to reach the solution. A Monte Carlo simulation has four distinct stages, explained and shown in pseudo-code in Figure 5 on the next page. (Browne et al., 2012).

(1) Selection: starting the simulation at the root node S, successor nodes S' are selected (by methods described below) until a leaf node L with no previously recorded simulations is encountered

(2) Expansion: expand the tree from L, generating successors and adding them as children, and choose one child L' to expand

(3) Simulation: playout the game by choosing random successors from L' until a terminal state T is reached

(4) Back-prop: evaluate the reward at T, and back-propagate the information to all predecessor states from L' to S

Choosing which successor state to select in the selection phase can be determined in many ways: in this paper, we utilize both uniform random selection, as well as the UCB1 algorithm (described later) which selects nodes based on their estimated value as well as their upper confidence bound. In order to apply this algorithm to our problem, simulations are run *n* number of times and then the final state is retrieved by selecting the successor with the highest expected value based on the back-propagated results at each intermediate state. When we reach a terminal state (after selecting the best six Pokémon for each variable) the complete attacking set is returned.

MCTS Using States (MCTS 1)

The first formulation of MCTS uses a slight variation on the state formulation presented previously. In this formulation, a valid state $S = \{a_1 \dots a_6\}$ may have unassigned attacking Pokémon, so long as all $a_i > a_u$ are also unassigned, where $u < i$, and $1 \leq u, i \leq 6$. In this formulation then, each action is an assignment of the first unassigned variable a_u. A terminal state is reached when all a_i in S are assigned. The reward of a terminal state is 1 if the attacking set successfully defeated the defending set (as previously described), or else 0. The method for choosing a successor in the selection phase, as laid out above, uses a uniform random distribution to make a choice.

```
function MonteCarloTreeSearch(state, n)
   initialize node counter table N, node value table V
   root_node ← Node(state)
   for n loops do
       leaf_node ← Selection(root_node)
       terminal_reward ← Simulation(leaf_node)
       BackProp(leaf_node, terminal_reward)

   return BestPath(node)

function Selection(node)
   while node != terminal do
       if node == fully expanded
           node ← ChooseSuccessor(node)
       else
           return Expand(node)
   return node

function Expand(node)
   successors ← GenerateSuccessors(node.state)
   for each successor in successors do
       add Node(successor) as child of node
   return random node from Children(node)

function Simulation(node)
   state ← node.state
   while state != terminal do
       successors ← GenerateSuccessors(state)
       state ← random state from successors
   return Reward(state)

function BackProp(node, terminal_reward)
   while node is not root_node do
       N(node) ← N(node) + 1
       V(node) ← V(node) + terminal_reward
       node ← Parent(node)

function BestPath(node)
   while node != terminal
       node ← argmax(V(child) / N(child))
           for all child in node.children
   return node.state
```

Figure 5: Pseudo-code of the Monte Carlo Tree Search algorithm.

MCTS Using Actions (MCTS 2)

Reusing the same states and actions from MCTS 1, since each attacking Pokémon acts independent of the others, our second MCTS formulation assigns values to actions instead of to states. Since each action is state dependent (assigning a Pidgey to a_1 is distinct from assigning a Pidgey to a_2), this allows us to evaluate the effectiveness of placing a Pokémon in a particular position, regardless of the rest of the state. In the prior formulation, the states {Dragonite, Pidgey} and {Dragonite, Ekans} would be evaluated separately, and information would be backpropagated to each state individually. However, in this case, it is Dragonite (one of the strongest Pokémon) who is clearly the contributing factor to the state's success. Thus, this formulation attempts to generalize information gained from playouts by accumulating the information in the action assigning Dragonite to a_1. All other aspects of MCTS remain as before.

MCTS Using UCB1 and Expected Value (MCTS 3)

The final formulation of MCTS builds on the previous. In this formulation, the reward function $R(S)$ is dependent on a linear combination of the features of that state, based on the evaluation metrics listed above, and calculated according to Equation 5 below. We use the inverse for battle time and attacker health points lost, because for these metrics, lower values are better. The weights were determined by hand based on domain knowledge and experimentation to be 30.0, 300.0, and 1000.0 respectively.

$$R(S) = w_{AR} * AttackersRemain + \frac{w_{BT}}{BattleTime} + \frac{W_{AHPL}}{AttackerHPLost}$$

Equation 5: Reward function of a state.

The algorithm now effectively keeps track of the estimated expected value of each action, as opposed to the average rate of attacker success. This improvement is paired with the UCB1 algorithm, which is used during selection phase to select the best node to expand (Bradberry, 2015). This algorithm balances exploitation and exploration by selecting the node with maximum expected value estimate plus upper confidence bound, as outlined below in Equation 6. Through hand tuning, we arrived at a C value of 150, based on our reward function.

$$\frac{V(S)}{N(S)} + C \times \sqrt{\ln\left(\frac{N_t}{N(S)}\right)}$$

Equation 6: Upper Confidence Bound (UCB) Formula

In Equation 6, C is a tunable constant, $V(S)$ is the value of a state, $N(S)$ is the number of times a particular state has been visited, and N_t is the number of visits to all states.

$$BestState = \underset{s \in Children}{\text{argmax}} \; UCB(s)$$

Equation 7: UCB1 Selection Algorithm

In order to compute the best state to choose during selection phase, we must iterate over all children of a node (successors of a state) and choose the node (successor) with the highest UCB value, as outlined in Equation 7.

Experiments

To evaluate our algorithms, we performed a series of experiments in order to compare: performance of our different MCTS formulations, performance of the expected value MCTS with and without the UCB1 algorithm, performance of expected value MCTS with UCB1 for different numbers of simulations, and the performance of our MCTS formulations versus GBFS.

MCTS Formulations 1, 2, and 3

To compare MCTS algorithms, we used a dataset of 100 randomly chosen defender sets, and recorded for each algorithm: the chosen attacker set after 10,000 simulations, time to compute, number of attackers remaining after battle, amount of health points lost during battle, and duration of battle. Table 1 below summarizes our results.

MCTS 1, 2, 3 at 10,000 Simulations			
MCTS Version	1	2	3
Compute Time (s)	32.619	31.209	39.014
Attackers Remaining	2.242	4.111	4.404
Health Lost	624.633	421.093	357.217
Battle Time (s)	70.238	57.741	50.076

Table 1: Results of MTCS Version 1, 2, and 3 Comparison

The results of the MCTS Action algorithms (2 and 3) compared to MCTS 1 clearly return better solutions, showing 44.57% and 55.98% average improvement over all performance metrics respectively.

The improvement of performance metrics from the state-based MCTS algorithm to the action-based variants likely stems from two factors. Firstly, since Pokémon perform largely independently, a state-based algorithm that considers {Dragonite, Pidgey …} and {Pidgey, Dragonite …} two completely distinct states is unable to capitalize fully on the information it gains during playouts. Secondly, the number of actions in our formulation is significantly less (144 actions per variable assignment × 6 variables × 1 ordering of assignment = 864), meaning we can get away with running far fewer simulations. The MCTS 1 algorithm likely suffers from a lack of information, simulating only 10,000 times in the considerably larger set of states versus set of actions. Given the computation time for these number of simulations, going far beyond 10,000 states is impractical.

The improvement from MCTS 2 to MCTS 3 is easily explained by our algorithm's increased discerning power. Since MCTS 3 can evaluate the "goodness" of states with a complex linear reward function, it is more easily directed towards good solutions, versus simply solutions that are likely to result in a victory, as in MCTS 2. Further, since the UCB1 algorithm is used in MCTS 3, we make more efficient use of our playouts, by exploring better states on average than a random selection.

UCB1 vs. Random

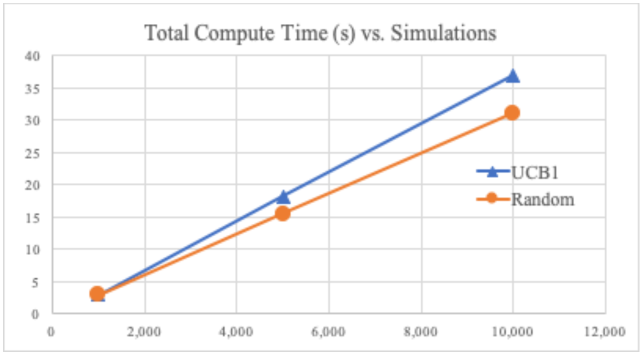

Figure 6: Total Compute Time vs. Simulations graph

To determine the effectiveness of the UCB1 algorithm, we used a dataset of 100 randomly chosen defender sets, and recorded the solution and metrics described earlier of MCTS 3 after 1,000, 5,000 and 10,000 simulations; once using UCB1 and once using Random selection. Table 2 below summarizes our results.

UCB1			
Simulations	1,000	5,000	10,000
Compute Time (s)	2.996	18.145	36.940
Attackers Remaining	4.030	4.364	4.374
Health Lost	411.776	360.746	355.395
Battle Time (s)	56.365	49.565	48.998

Random			
Simulations	1,000	5,000	10,000
Compute Time (s)	2.954	15.501	31.147
Attackers Remaining	4.030	4.485	4.485
Health Lost	415.291	364.403	360.701
Battle Time (s)	57.095	52.845	52.226

Table 2: UCB1 vs. Random successor choosing

Upon analysis, the UCB1 algorithm appears to show slightly improved metrics (1.5% decrease in HP lost and 6.2% decrease in battle time), at the cost of a 15.68% in

MCTS 3 Performance Over Simulations Run										
Simulations	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Compute Time (s)	2.838	6.408	10.064	13.702	17.338	20.961	24.579	28.196	31.808	35.406
Attackers Remaining	4.192	4.475	4.475	4.495	4.515	4.505	4.505	4.515	4.515	4.515
Health Lost	389.633	355.478	354.290	348.136	344.838	344.797	344.337	343.394	343.342	341.421
Battle Time (s)	53.812	50.291	49.319	48.563	48.626	48.601	48.493	48.372	48.169	48.083

Table 3: MCTS 3 performance over simulations run

creased compute time at 10,000 simulations. In the attackers remaining metric, the UCB1 algorithm actually performed worse than random, showing a 2.47% decrease.

Intuitively, running more simulations increases the total compute time. The graph on the previous page (Figure 6) shows that for both algorithms this increase happens linearly, as expected. The increased time overhead between the two algorithms; however, can be explained by the extra computation that must be done to compute the upper confidence bound of each action at each stage of the simulation phase. Each partially assigned state requires the computation of the UCB for 144 actions, which results in a non-trivial increase in time.

As for performance metrics, here our results seem mixed: while two of our three performance metrics increase, one (attackers remaining) actually performs worse. We believe this mixed result to be a consequence of the particular reward function implemented for MCTS 3. As discussed above, it is hard to judge which of the metrics is the "most important." In this case, our reward function values reduced battle time very highly. As a result, we see this trade off in our metrics: an exploitation of the preference for battle time at the cost of remaining attackers. It is also important to note that the remaining attacker's metric is bounded to be a natural number on the range $(0, 6]$. This small and restricted range of values may affect the algorithm's ability to balance preference of saving more attackers or improving other metrics.

Number of Simulations Effect on Performance

To determine how the number of simulations run before a solution is found effects the performance of our algorithm, we used a dataset of 100 randomly chosen defender sets, and recorded the same metrics as previous experiments, in 10 1,000 simulation intervals. Drawing on previous experiments, we opted to analyze MCTS 3, as we believe it to be the most complete and "real world" algorithm. Table 3 above summarizes our results.

The trend from the data above shows a rapid improvement initially as new values are explored, which quickly decreases to slowing, steady improvement as more simulations are added. In Figure 7 to the top right, the average battle time drops significantly over the first 4,000 simulations, before reaching a much slower steady decline.

These results suggest that there is a practical cutoff point in our problem where the trade-off of compute time vs. performance shows diminishing returns. We believe this is a natural consequence of the problem we are solving. As we continue to simulate, we approach asymptotically the optimal solution to the problem. As such, given the number of solutions in the space, it gets increasingly difficult to improve the solution found by our Monte Carlo method.

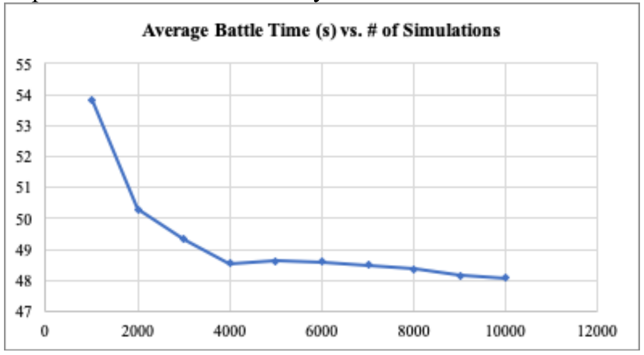

Figure 7: Average Battle Time vs. # of Simulations graph

MCTS 3 vs. GBFS

Lastly, we compared the average performance of GBFS over 1,000 runs to the average performance of MCTS 3 over 100 runs at 10,000 simulations. The table below shows the results from this experiment. Clearly, while GBFS returns rapid solutions that solve the gym problem, it performs nowhere near as well as MCTS 3. Further, MCTS 3 shows a clear advantage over the naive solution, even with few simulations. However, in a real world scenario, having a valid solution may be preferable to having no solution at all. In this case, the 2.132 second compute time of GBFS might be an advantage.

Algorithm	Compute Time	Attackers Left	HP Lost	Battle Time
GBFS	2.132	1.654	723.062	80.246
MCTS 3	35.406	4.515	341.421	48.083

Table 4: GBFS vs. MCTS 3

The results here are not surprising, namely because GBFS does not discriminate based on the goodness of states, it simply returns as soon as it finds a single solution. This is why we see such a difference in compute time between the two—Monte Carlo simulates many solutions, while GBFS finds only one.

Conclusion

Our results reveal that although methods exist to arrive at a solution to the gym problem in Pokémon Go quickly, and solutions that are "good," it is difficult to develop a method that can do both. Greedy search techniques were able to arrive at a solution in two seconds but with poor quality of solution. Monte Carlo methods run with a low number of simulations such that compute time was comparable to GBFS performed better, but nowhere near optimal. It was only after an order of magnitude more compute time that Monte Carlo began to return "good" solutions. Future work should pursue methods for improving state evaluation, and for optimizing battle simulation time, to maximize the number of states that can be sampled before a solution must be returned. Further, future work can add many optimizations, like local search layers, to find small improvements to already well optimized answers.

References

Barnes, D., and Livio, B. 2018. Pokemon Go JSON Pokedex. https://github.com/pokemongo-dev-contrib/pokemongo-json-pokedex

Bradberry, Jeff. 2015. Introduction to Monte Carlo Tree Search. https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/

Browne et al., 2016. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*. Vol. 4, No. 1.

Coles, A., and Smith, A. 2007. Greedy Best-First Search when EHC Fails. https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume28/coles07a-html/node11.html

Felix, D. 2016. How to Calculate Comprehensive DPS. https://pokemongo.gamepress.gg/how-calculate-comprehensive-dps#appendix

Omu, H., and Q, M. 2016. Damage Mechanics. https://pokemongo.gamepress.gg/damage-mechanics

PokéBattler. 2018. Gym Battle Simulator. https://www.pokebattler.com/fights

Appendix

A: Algorithms

Total DPS Pseudo-code

```
function total_dps(attacker, quick_move, charge_move, defender):
  atk ← attacker.attack
  def ← defender.def

  stab ← 1.4 if quick_move.type in attacker.types else 1
  effectiveness ← product(effectiveness[quick_move.type][defender.type]) for all defender.type
  quick_damage ← floor(0.5 * quick_move.power * atk/def * stab * effectiveness) + 1
  quick_dps ← quick_damage  / quick_move.duration * 1000
  quick_eps ← abs(quick_move.energy) / quick_move.duration * 1000

  stab ← 1.4 if charge_move.type in attacker.types else 1
  effectiveness ← product(effectiveness[charge_move.type][defender.type]) for all defender.type
  charge_damage ← floor(0.5 * charge_move.power * atk/def * stab * effectiveness) + 1
  charge_dps ← charge_damage / charge_move.duration * 1000
  charge_eps ← abs(charge_move.energy) / charge_move.duration * 1000

  qdps, qeps, cdps, ceps ← quick_dps, quick_eps, charge_dps, charge_eps

  total_dps ← ((qdps * ceps) + (cdps * qeps)) / (ceps + qeps)

  return total_dps
```

One vs. One Battle Pseudo-code

```
function one_vs_one_battle(attacker, defender):
  atk_quick, atk_charge ← best_moves(attacker, defender)
  def_quick, def_charge ← best_moves(defender, attacker)

  atk_dps ← total_dps(attacker, atk_quick, atk_charge, defender)
  def_dps ← total_dps(defender, def_quick, def_charge, attacker)

  atk_faints_in ← attacker.health / def_dps
  def_faints_in ← defender.health / atk_dps

  winner ← attacker if atk_faints_in >= def_faints in else defender

  return winner
```