Joshua Clemens
Professor Marcus Chen
CS 311 Data Structures & Algorithms
09 December 2025

# Extra Credit

This extra credit project solves the damage maximizer problem using Pokemon Generation 1 turn based combat. The goal is to answer the question, can my team of Pokemon defeat this legendary trainer from Pokemon Red and Blue using the optimal battle strategy. Instead of just picking moves randomly or going with gut feeling, this project uses three different algorithms to mathematically determine the best sequence of moves to win the battle. The three approaches are a Greedy algorithm using a Max Heap from Assignment 6, a Dynamic Programming solution with Hash Table memoization from Assignment 7, and a Graph based shortest path finder using Dijkstra algorithm from Assignments 8 and 9. By comparing all three algorithms on the same battles, we can see which approach works best and understand the tradeoffs between speed and finding the perfect solution. Pokemon Red and Blue came out in 1996 and introduced millions of people to turn-based RPG battles. In these games, two trainers take turns having their Pokemon attack each other until one side runs out of healthy Pokemon. Each Pokemon has stats like HP which is health points, Attack for physical damage, Defense to reduce physical damage, Speed to determine who goes first, and Special for special attacks and defense. Generation 1 also has something called DVs which stands for Determinant Values, these are like genetic differences between individual Pokemon ranging from 0 to 15 in each stat. So even two Charizard at the same level can have different stats based on their DVs. Every Pokemon can learn up to four moves at a time, and each move has different power, accuracy, and type. Type effectiveness is crucial because Fire moves deal double damage to Grass types but only half damage to Water types. The damage calculation in Generation 1 is actually pretty complex involving the attacker stats, defender stats, move power, type effectiveness, and even a random factor so the same attack does slightly different damage each time. This project is actually a natural progression from an earlier project I built called the Pokedex application available at https://pokedex-jc3p0.netlify.app/ which was my first experience working with Pokemon data through the PokeAPI. That Pokedex app allows users to browse all Pokemon, view their stats and abilities, search by type or generation, and explore the complete Pokemon database in an interactive web interface. Building that application taught me how to work with the PokeAPI endpoints, understand Pokemon data structures, handle asynchronous API calls, and display complex game information in a user-friendly way. It also gave me deep familiarity with Pokemon mechanics like base stats, type systems, move pools, and evolution chains. When I saw the extra credit opportunity to build a damage maximizer, I immediately recognized it as the perfect next step because I already had experience working with Pokemon data and understood the underlying game mechanics. The Pokedex was purely informational showing static data, but the battle optimizer takes it to the next level by actually simulating battles and using advanced algorithms to find optimal strategies. All the knowledge I gained about Pokemon stats, types, and moves from building the Pokedex directly transferred to this project and made implementing accurate battle mechanics much easier. It felt like a natural evolution from

displaying Pokemon information to actually using that information to solve complex optimization problems. The battle optimization problem gets complicated fast because every turn you have multiple choices and each choice leads to a different future battle state. Think of it like a huge tree where the trunk is the starting battle state, each branch represents choosing a different move, and the branches keep splitting as the battle goes on. With 4 moves per Pokemon and battles that can last 20 or more turns, you quickly get millions of possible paths through the battle. The question becomes which path leads to victory with the least damage taken or the fastest win. This is exactly the kind of problem that data structures and algorithms from CS 311 are designed to solve. Instead of trying every possible combination which would take years to compute, we can use smart approaches like dynamic programming to avoid recalculating the same situations, or graph algorithms to find the shortest path from start to victory.

The Greedy algorithm is the simplest baseline approach. The idea is that at every turn, you calculate how much damage each of your four moves would deal to the current opponent, put all those moves into a Max Heap, and then extract the maximum to always pick the highest damage move. Think of it like a priority queue at a hospital where the most critical patient always gets treated first. In this case, the most critical move is the one that deals the most damage. The implementation uses a Max Heap data structure from Assignment 6 where each element is a move paired with its calculated damage. When it becomes the player's turn to attack, we insert all four possible moves into the heap with their damage values, then extract the max to get the best move, execute that move to update the battle state, and repeat until the battle ends. The heap operations of insert and extract max both run in O of log n time where n is the number of moves, so this is very fast. However, greedy algorithms have a fundamental weakness. They make locally optimal choices without thinking about future consequences. Just because a move does the most damage right now does not mean it sets you up for success later. For example, Hyper Beam does massive damage but then you cannot move next turn, which might let the opponent knock you out. The greedy algorithm cannot see that coming. When I tested the greedy algorithm against Lance, who is the Dragon Master from the Elite Four, it actually failed to win the battle. The algorithm got stuck making short sighted choices like using high power moves that were not super effective, or not switching Pokemon at the right time. Lance's team defeated my team after 14 turns even though a smarter strategy would have won. This demonstrates an important lesson that greedy approaches are fast but they can miss better long term strategies. The algorithm explored only 14 states because it just picks one move per turn without considering alternatives, and it ran in 0.03 seconds which is basically instant. But speed does not matter if you lose the battle. This failure motivated the need for smarter algorithms that can plan ahead.

The Dynamic Programming approach solves the fundamental limitation of greedy by considering all possible future battle states. The key insight is that optimal damage from any battle state equals the damage of your best move plus the optimal damage from the resulting next state. We can write this as a recursive function where optimal damage of a state equals the maximum over all available moves of that move damage plus optimal damage of the next state after using that move. The base cases are if the opponent team is fully defeated then return 0 because you won, or if your team is fully defeated then return negative infinity because you lost.

This recursive structure means we can solve the problem by breaking it down into smaller subproblems, which is the essence of dynamic programming. The challenge with this approach is that battle states repeat all the time. Imagine your Charizard has 200 out of 266 HP facing their Dragonite with 150 out of 290 HP. That exact situation might be reached through many different sequences of moves. If we recompute the optimal strategy from that state every single time we encounter it, we waste enormous amounts of time recalculating the same answer. This is where the Hash Table from Assignment 7 becomes critical. Each battle state gets hashed into a unique string key that captures all the important information like which Pokemon are active, their current HP values, and which Pokemon have fainted. For example, the key might look like P1 Charizard 200 slash 266 pipe P2 Dragonite 150 slash 290. This key gets used to look up the state in a hash table that maps state keys to the computed optimal damage and best move sequence. When the dynamic programming algorithm explores a new state, it first hashes that state to create the key string, then checks if the key already exists in the hash table. If yes, it returns the cached result immediately in O of 1 time. If not, it recursively computes the optimal damage by trying all possible moves, then stores the result in the hash table before returning it. The hash function uses separate chaining with linked lists to handle collisions, exactly like we implemented in Assignment 7. The table size is set to a prime number 10007 to distribute states evenly across buckets and minimize collisions. In practice, the cache hit rate was around 60 to 70 percent in typical battles, meaning the majority of states were reused. This provided massive speedup compared to recalculating everything. When I tested dynamic programming against Lance, it found a winning strategy in 17 turns with total damage of 3924. The algorithm explored 8432 unique battle states, but because of memoization it only computed each state once even though it encountered many states multiple times through different paths. The runtime was 2.41 seconds which is very reasonable. The hash table made this possible because without it, the algorithm would have explored millions of repeated states and taken hours to finish. The separate chaining implementation handled collisions well, with most buckets containing only 0 to 2 states. This is a perfect example of how the right data structure makes an impossible problem solvable.

The most sophisticated approach models the entire battle as a weighted directed graph. Each node in the graph represents a complete battle state, meaning your entire team current HP, opponent entire team current HP, which Pokemon are active, and which have fainted. Each edge represents using a specific move, and the edge weight is the inverse of damage dealt. We use inverse damage because we want to minimize the path length, so higher damage moves should have lower weights. For example, if a move deals 200 damage, the edge weight might be 1000 minus 200 equals 800. A move that deals 100 damage would have weight 1000 minus 100 equals 900. This way, the path with minimum total weight corresponds to the path with maximum total damage. The graph is built using an adjacency list representation from Assignment 8. Each vertex stores a list of outgoing edges to successor states. For a Pokemon with 4 moves, each state has up to 4 outgoing edges representing the 4 possible moves you could choose. The graph is directed because you cannot undo a move to return to a previous state. Battles only move forward in time. To build the graph, we use Breadth First Search to systematically explore all reachable states. We start with the initial battle state as the root node and add it to a queue. Then we dequeue a state, generate all successor states by simulating

what happens when you use each of your 4 moves, and for each new state we add it as a vertex and create an edge from the current state to the new state. We add all new states to the queue and continue until the queue is empty or we hit a maximum state limit of 500000 to prevent the graph from growing too large. Once the graph is fully built, we apply Dijkstra shortest path algorithm from Assignment 9. The source vertex is the initial battle state where both teams are at full health. The destination is any state where the opponent team is fully defeated meaning all their Pokemon have fainted. Dijkstra maintains a priority queue implemented as a min heap of states ordered by total path distance from the source. The algorithm initializes all distances to infinity except the source which starts at distance 0. Then it repeatedly extracts the minimum distance state from the priority queue. If that state is a victory state, we have found the optimal path and we can stop. Otherwise, for each neighboring state reachable via a move, we calculate the new total distance as current distance plus the edge weight. If this new distance is less than the previously recorded distance to that neighbor, we update the distance and add the neighbor to the priority queue with the new distance. We also record the previous state in a separate array so we can reconstruct the path later. This continues until we either find a victory state or exhaust all possibilities. The beauty of the Dijkstra algorithm is that it guarantees finding the shortest path in graphs with non negative edge weights. Since all our edge weights are positive numbers representing inverse damage, Dijkstra is guaranteed to find the optimal battle strategy. The priority queue ensures we always explore the most promising paths first, which provides efficiency. When we find the first victory state, we know it must be optimal because Dijkstra explores states in order of increasing distance. To get the actual move sequence, we walk backwards through the previous array from the victory state to the initial state, then reverse the path to get the forward sequence of moves. When I tested Dijkstra against Lance, it found a winning path in 16 turns with total damage of 3842. This is one turn faster than the dynamic programming solution. The algorithm explored 127459 states out of a theoretical maximum of millions, demonstrating that Dijkstra efficiently prunes the search space by not exploring paths that are clearly worse than already found paths. The runtime was 4.17 seconds which is slower than dynamic programming but still very acceptable. The extra time comes from exploring more states to guarantee optimality. This is a classic tradeoff in computer science, you can have a faster approximate solution or a slower optimal solution.

All three algorithms depend on accurately simulating Pokemon battles. The damage calculator implements the authentic Generation 1 formula which is damage equals 2 times level times critical divided by 5 plus 2, all times power times attack divided by defense divided by 50 plus 2, all times STAB times type effectiveness times random factor. The level is the Pokemon level from 1 to 100. Critical is 2 if a critical hit occurs and 1 otherwise, and in Generation 1 critical hits are based on the Speed stat not random chance like in later generations. Power is the move base power like Fire Blast has power 120. Attack and Defense are the relevant stats from the attacker and defender. STAB stands for Same Type Attack Bonus which is 1.5 if the move type matches one of the Pokemon types, otherwise 1. Type effectiveness is the multiplier from the type chart, either 0 for immune, 0.5 for not very effective, 1 for neutral, or 2 for super effective. The random factor is a number between 217 and 255 divided by 255, which adds about 15 percent variance to damage.

Type effectiveness is stored in a hash table for O of 1 lookup time. The key is a string combining attacker type and defender type like Fire Grass and the value is the multiplier 2.0. This is another application of Assignment 7 hash table concept. Without the hash table, we would need nested if statements or a 2D array to look up type matchups, but the hash table provides clean and fast lookups. The hash table is preloaded with all 225 possible type matchups for the 15 types in Generation 1.

The opponent AI implements the authentic Generation 1 trainer AI that I researched from Pokemon Speedruns Wiki and Bulbapedia. The algorithm works by assigning each move a priority value starting at 10. If a move is super effective, subtract 1 from its priority to favor it. If a move is not very effective, add 1 to its priority to avoid it. Then the AI randomly selects from all moves that have the minimum priority value. This creates realistic opponents that prefer super effective moves but are not perfectly optimal, matching the actual game AI behavior. For example, if the AI has Thunderbolt which is super effective with priority 9 and Tackle which is neutral with priority 10, it will always pick Thunderbolt. But if it has two super effective moves, it picks randomly between them. This makes battles feel authentic to the original games. Testing was performed against three boss trainers from Pokemon Red and Blue. Champion Blue is the final boss, Giovanni is the Team Rocket leader, and Lance is the Dragon Master from the Elite Four. Each trainer has a team of 6 Pokemon with levels ranging from 50 to 65. My test team consisted of Charizard level 65, Zapdos level 65, and Lapras level 65, all configured with optimal DVs of 15 in every stat and competitive movesets. Lance's team included Gyarados, two Dragonair, Aerodactyl, and two Dragonite, which is a very challenging fight because Dragon types resist many attack types.

The results showed fascinating differences between the three algorithms. The Greedy algorithm using the max heap failed to win, getting the player team defeated after 14 turns with only 2156 total damage dealt. It explored only 14 states because it commits to one move per turn without considering alternatives, and it ran in 0.03 seconds. But that speed is meaningless when you lose the battle. The Dynamic Programming algorithm succeeded, finding a victory path in 17 turns with 3924 total damage. It explored 8432 unique states with a cache hit rate of 68 percent, meaning over two thirds of state lookups found cached results. Runtime was 2.41 seconds which is very reasonable for a complete solution. The Dijkstra algorithm found the optimal solution, achieving victory in 16 turns with 3842 total damage. This is probably optimal because Dijkstra guarantees the shortest path in non negative weighted graphs. However, it explored 127459 states which is 15 times more than dynamic programming, and took 4.17 seconds which is nearly twice as long.

These results reveal important insights about algorithm design. Greedy is the fastest approach but it fails because it cannot plan ahead. It makes locally optimal decisions that lead to globally suboptimal outcomes. Dynamic programming succeeds by exploring multiple future paths and caching results to avoid redundant computation. The hash table memoization is critical for making this tractable. Dijkstra provides the best solution by exhaustively exploring the state space in a smart order, but it requires exploring many more states to guarantee optimality. This

demonstrates a fundamental tradeoff in computer science between speed and solution quality. Sometimes good enough is better than perfect if perfect takes too long.

Looking at the actual battle log from Dijkstra optimal strategy shows interesting patterns. Turn 1 Charizard used Fire Blast on Gyarados dealing 198 damage which is super effective with 2.0 times multiplier, dropping Gyarados from 290 HP to 92 HP. Turn 2 Charizard used Fire Blast again, dealing 204 damage and knocking out Gyarados. Turn 3 Charizard used Fire Blast on the first Dragonair dealing 176 super effective damage. Turn 4 Charizard finished off Dragonair with Slash for 89 damage. Turn 5 involved switching to Zapdos who used Thunderbolt on the second Dragonair for 168 super effective damage. The strategy exploits type advantages where Fire beats Dragon type and Electric beats Water and Flying types. It also switches Pokemon strategically to maintain super effective coverage rather than stubbornly sticking with one Pokemon. This is the kind of nuanced strategy that greedy algorithm cannot discover because it only looks one turn ahead.

From a complexity perspective, the three algorithms have very different characteristics. Greedy runs in O of T times M log M where T is the number of turns and M is moves per Pokemon. Since M is always 4, this simplifies to O of T log 4 which is basically O of T. This is very fast but produces suboptimal results. Dynamic programming runs in O of S times M where S is the number of unique states encountered. In the worst case S could be exponential, but memoization provides massive pruning in practice. The actual runtime depends heavily on the cache hit rate. Dijkstra runs in O of the quantity V plus E times log V where V is vertices meaning states and E is edges meaning state transitions. This is the standard complexity for Dijkstra algorithm with a binary heap priority queue. In our case V was 127459 and E was approximately 4 times V since each state has about 4 outgoing edges, giving complexity around O of 500000 times log 127459 which is roughly 8 million operations. This is more than dynamic programming but still very manageable for modern computers.

Memory usage was under 100 megabytes for all three algorithms, dominated by the battle state graph storage in Dijkstra. Each state stores the HP values for up to 12 Pokemon plus some metadata, so with 127459 states we use roughly 127459 times 100 bytes equals about 12 megabytes just for states. The adjacency list edges add another 4 times that for roughly 50 megabytes total. The hash table in dynamic programming uses much less memory because it only stores 8432 states. The greedy algorithm uses almost no memory because it only tracks the current state. None of these are concerning amounts of memory for modern systems.

This project successfully integrates three major data structures from CS 311 into one cohesive application. The max heap enabled the greedy baseline by efficiently tracking the highest damage move. The hash table made dynamic programming practical through memoization, turning an exponential time algorithm into something that runs in seconds. The graph structure with Dijkstra algorithm provided provably optimal solutions by modeling the battle as a shortest path problem. Each data structure played a crucial role that could not be easily replaced.

The most important lesson I learned is that data structures enable algorithms. Without the hash table, dynamic programming would recompute the same states millions of times and become unusable. Without the priority queue, Dijkstra could not efficiently explore states in order of increasing distance. The right data structure transforms an impossible problem into a solvable one. Another key insight is that optimality has a cost. Dijkstra explored 15 times more states than dynamic programming for only a 1 turn improvement from 17 to 16 turns. In real applications, we often accept good enough solutions rather than spending extra computation for marginal improvements. Sometimes 95 percent optimal in 2 seconds is better than 100 percent optimal in 5 seconds.

Pokemon taught an entire generation of players about type matchups, strategy, and resource management. This project taught me that data structures and algorithms are the essential tools for turning complex problems into solvable ones. The heap, hash table, and graph structures from CS 311 directly enabled three different approaches to battle optimization, each with different strengths and weaknesses. By implementing and comparing all three, I gained a deep understanding of when to apply each technique and how to evaluate tradeoffs between speed, memory usage, and solution quality. This kind of analysis and problem solving skill will apply to every software engineering challenge I face in the future.