Gotta Traverse 'Em All: Application of Directed Weighted Graphs in Pokémon Team Optimisation

Ariq Ulwan Hammam - 13525084
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: ariq.hammam@gmail.com , 13525084@std.stei.itb.ac.id

***Abstract*— In competitive Pokémon, team building remains one of the most pivotal elements dictating strategic performance. With the hard limit of 6 Pokémon per team and 2 types per Pokémon, players are forced to make significant tradeoffs with regards to type coverage. To resolve this structurally, we abstract the elemental type chart into a weighted directed graph mapped onto a static 18×18 adjacency matrix, using the element-wise Hadamard product to resolve composite dual-type vulnerabilities. This paper attempts to utilize a greedy algorithm operating with linear time to explore optimal type combinations for all constituent members of Pokémon by sequentially searching for the ideal combination to cover the existing weaknesses of the team so far. Guided by a multi-criteria objective function, the algorithm dynamically applies calibrated rewards for switchability and steep penalties for compounding weaknesses to isolate local optimization peaks. Experimental execution demonstrates that by introducing a randomized initial seed, this framework successfully avoids rigid outputs to generate highly insulated, diverse, and defensively stable team cores in fractions of a second.**

Keywords—Pokemon; graph theory; Hadamard product; greedy algorithm; adjacency matrix

I. INTRODUCTION

Pokémon, created in 1996, has been widely considered to be the highest-grossing franchise of all time, with it encompassing media including but not limited to video games, anime, manga, and trading card games. This paper will mostly concern the video games.

Team building has been a crucial part of the competitive scene in Pokémon, affecting a player's performance significantly. One of the most essential aspects of this concerns itself with the typing of members of the team, whether it be defensive or offensive. The author will primarily focus on the defensive aspects of team building and primarily consider non-weather teams in this paper.

In Pokémon, Pokémon are divided into 19 types, each with its own unique interactions with each other. Pokémon could either be single-type or dual-type (except for the Stellar type, which could only ever be single-type), making for 172 unique combinations. Among these 172, 163 combinations are already used, leaving only 9 unused. However, they are not created equally, with some type combinations being considered more advantageous than others. However, considering its lack of traits in the defensive department and its rather niche usage, Stellar type would be ignored in this paper.

Types in themselves have four types of interactions, that being immunity (0x multiplier), resistance (0.5x multiplier), normal interaction (1x multiplier), and weakness (2x multiplier). For double-typed Pokémon, these interactions could be calculated by multiplying the interactions of its types, and thus resistance is split into single resistance (0.5x multiplier) and double resistance (0.25x multiplier), and weakness is likewise split into single weakness (2x multiplier) and double weakness (4x multiplier).

Teams are composed of up to six Pokémon each, forcing players to choose at most six unique type combinations. A good team is a team that is defensively strong, which means that it has more immunities and resistances than weaknesses, and complementary, which means that members in the team share as few weaknesses with each other as possible.

II. THEORETICAL BACKGROUND

A. Graph Theory

Graph theory is a branch of mathematics that concerns itself with graphs, which are defined as a mathematical structure used to model pairwise relationships between objects. They are composed of nodes/vertices representing the objects as well as edges representing the relationships between them. This property has made them among the most invaluable tools in abstracting complex systems by reducing them to a set of objects and relationships, making it easier to analyze and manipulate via algorithmic methods. Formally, graphs are defined as $G = (V, E)$, with V being a set consisting of all the vertices, and E being a set consisting of all the edges.

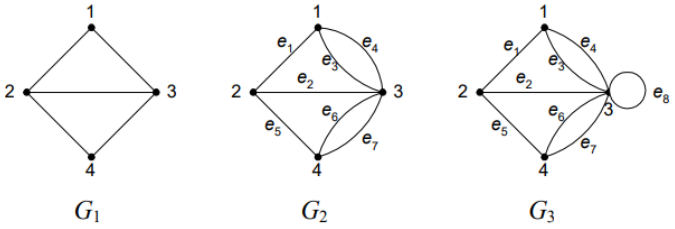

Fig. 1.1: A few examples of graphs
Taken from Dr. Ir. Rinaldi Munir, M.T.'s website, at
(https://informatika.stei.itb.ac.id/~rinaldi.munir/)

The graphs that are used in the analysis of Pokémon team building are weighted directed graphs. Weighted, meaning that each edge has a numerical value (weights) attached, and directed, meaning that vertices are connected by edges representing one-way relationships as opposed to mutual ones by default, represented as arrows.

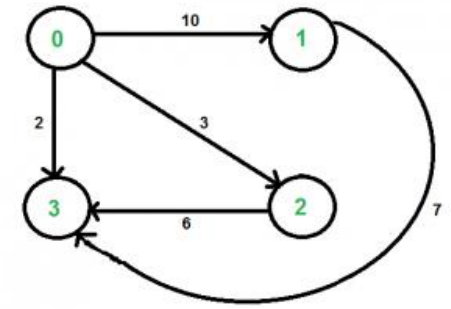

Fig. 1.2: An example of a weighted directed graph
Taken from GeeksforGeeks

To represent Pokémon type effectiveness, each type except for Stellar (for reasons outlined in the introduction) will be represented as vertices, with 18 outgoing edges and 18 incoming edges each. The multipliers for immunity, resistance, normal interaction, and weakness will serve as the weights of each edge.

B. *Matrix*

To define a matrix, Let $I_n \coloneqq \{1, \cdots, n\}$

An n times m matrix with coefficients in a ring R is a function $a : I_n \times I_m \to R$. We define $a_{i,j} \coloneqq a(i,j)$. Or simply put, it is a rectangular array of numbers, expressions, and symbols within n rows and m columns.

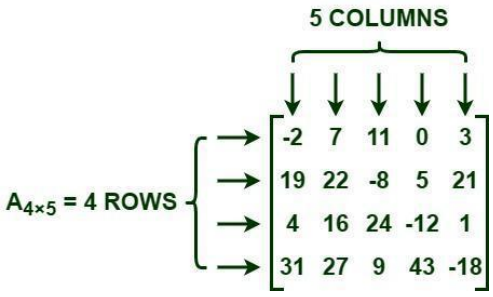

Fig. 1.3: An example of a matrix
Taken from GeeksforGeeks

Within graph theory, a square matrix called an adjacency matrix is used to represent adjacencies—in other words, vertices that are connected to each other by edges.

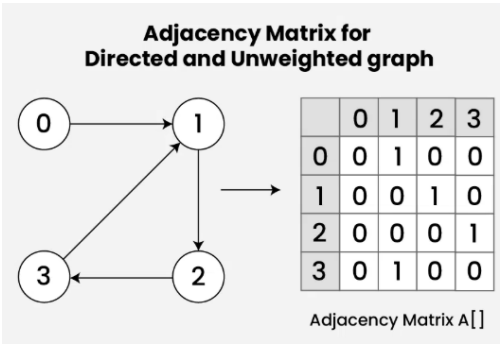

Fig. 1.4: An example of a graph and the adjacency matrix representing it
Taken from GeeksforGeeks

In directed weighted graphs such as ours, adjacency matrices aren't just simple 1s for connected and 0s for unconnected. Rather, initial and terminal vertices are differentiated, making the matrices not necessarily symmetric, unlike in undirected graphs. By convention, the initial vertex is represented by the row, whilst the terminal vertex is represented by the column, such that the edge (u, v)'s weight would be the element $M_{(u,v)}$.

Weights are quite simple, with the weight of each edge (u, v)'s weight being represented as the element $M_{(u,v)}$. Representation of non-adjacent nodes are complex and diverse, with some pathfinding algorithms like Dijkstra's opting to represent them with ∞, some others representing it as 0, among many other solutions. However, as we are dealing with a K_{18}^*, representation of non-adjacent nodes is a complete non-concern to this paper.

Matrices are used to make the calculation of type interactions of dual-type Pokémon easier. Defensively speaking, the interactions of dual-type Pokémon with the 18 types could be represented as a Hadamard product (that is, the result of a binary operation that takes in two matrices of the same dimensions and returns a matrix of the multiplied corresponding elements) of the two 18×1 column vectors extracted from the adjacency matrix that corresponds to the Pokémon's component types, where each row index represents the source attacking type and the resulting vector entries dictate the final damage multipliers.

C. *Greedy Algorithm*

A greedy algorithm is defined as an algorithmic paradigm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum. In the context of combinatorial optimization, rather than evaluating the entire solution space of all possible multi-node combinations simultaneously, a greedy strategy operates sequentially. At each iteration, the algorithm queries an objective function, isolates the single candidate that offers the maximum immediate utility, and appends it to the active cluster.

While highly efficient, a pure greedy algorithm possesses distinct mathematical boundaries and structural properties. The fundamental trade-off of a greedy heuristic is that it lacks foresight. Because it commits to the best local choice at step n, it cannot backtrack or alter previous selections if a different combination would have yielded a superior overall structural profile at step $n+1$. Consequently, while it is mathematically guaranteed to find a highly optimized, stable solution in linear time, it does not guarantee the absolute global optimum of the system. For a six-member team selected from 171 possible type combinations, a brute-force global search requires calculating:

$$\binom{171}{6} \approx 2.4 \times 10^{10} \text{ combinations}$$

The greedy paradigm reduces this solution space down to a fraction of the computational cost, trading absolute global perfection for practical viability.

Because greedy choices are entirely dependent on the current state of the system, the algorithm is highly path-dependent. If the system is entirely deterministic, it will generate the exact same solution every run. To explore different regions of the graph, a randomized initialization step is introduced. By selecting the very first vertex cluster uniformly at random, the algorithm establishes a unique baseline of flaws and strengths. The subsequent greedy iterations then dynamically pivot to complement that specific initial seed, allowing the program to function as an exploratory generator capable of outputting a wide variety of structurally distinct, balanced configurations across different execution runs.

III. Method

A. Graph Formulation and Structural Node Resolution

To transition from the theoretical abstraction of a weighted digraph into a functional computational model, the graph must be explicitly initialized within code structures. The system maps the type interactions using an 18×18 static adjacency matrix $M \in \mathbb{R}^{18\times18}$. Within this matrix, rows represent source vertices (attacking types) and columns represent target vertices (defending types).

Because this paper primarily concerns the defensive optimization of a team, the system must isolate inbound arc vectors. When calculating defensive attributes, the algorithm relies on a precise combination of coordinate mapping, matrix column slicing, and element-wise matrix algebra (Hadamard product).

The set of vertices V representing the 18 distinct Pokémon types, is mapped to a zero-indexed, ordered linear array. This defines a bijective mapping function $f: V \rightarrow \{0,1,\dots,17\}$, assigning each unique type string a fixed structural identity:

$$f(\text{NORMAL}) = 0, \quad f(\text{FIRE}) = 1, \quad f(\text{WATER}) = 2, \quad \dots, \quad f(\text{FAIRY}) = 17$$

```python
TYPES = [
    "NORMAL", "FIRE", "WATER", "ELECTRIC", "GRASS", "ICE", "FIGHTING", "POISON", "GROUND",
    "FLYING", "PSYCHIC", "BUG", "ROCK", "GHOST", "DRAGON", "DARK", "STEEL", "FAIRY"
]
TYPE_TO_IDX = {t: i for i, t in enumerate(TYPES)}

# Type Effectiveness
ADJ_MATRIX = np.array([
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0.5, 0, 1, 1, 0.5, 1],       # NORMAL
    [1, 0.5, 0.5, 1, 2, 2, 1, 1, 1, 1, 1, 2, 0.5, 1, 0.5, 1, 2, 1],    # FIRE
    [1, 2, 0.5, 1, 0.5, 1, 1, 1, 2, 1, 1, 1, 2, 1, 0.5, 1, 1, 1],      # WATER
    [1, 1, 2, 0.5, 0.5, 1, 1, 1, 0, 2, 1, 1, 1, 1, 0.5, 1, 1, 1],      # ELECTRIC
    [1, 0.5, 2, 1, 0.5, 1, 1, 0.5, 2, 0.5, 1, 0.5, 2, 1, 0.5, 1, 0.5, 1], # GRASS
    [1, 0.5, 0.5, 1, 2, 0.5, 1, 1, 2, 2, 1, 1, 1, 1, 2, 1, 0.5, 1],    # ICE
    [2, 1, 1, 1, 1, 2, 1, 0.5, 1, 0.5, 0.5, 0.5, 2, 0, 2, 2, 2, 0.5],  # FIGHTING
    [1, 1, 1, 1, 2, 1, 1, 0.5, 0.5, 1, 1, 1, 0.5, 0.5, 1, 1, 0, 2],    # POISON
    [1, 2, 1, 2, 0.5, 1, 1, 2, 1, 0, 1, 0.5, 2, 1, 1, 1, 2, 1],        # GROUND
    [1, 1, 1, 0.5, 2, 1, 2, 1, 1, 1, 1, 2, 0.5, 1, 1, 1, 0.5, 1],      # FLYING
    [1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 0.5, 1, 1, 1, 1, 0, 0.5, 1],        # PSYCHIC
    [1, 0.5, 1, 1, 2, 1, 0.5, 0.5, 1, 0.5, 2, 1, 1, 0.5, 1, 2, 0.5, 0.5], # BUG
    [1, 2, 1, 1, 1, 2, 0.5, 1, 0.5, 2, 1, 2, 1, 1, 1, 1, 0.5, 1],      # ROCK
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 2, 1, 0.5, 1, 1],          # GHOST
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 0.5, 0],          # DRAGON
    [1, 1, 1, 1, 1, 1, 0.5, 1, 1, 1, 2, 1, 1, 2, 1, 0.5, 1, 0.5],      # DARK
    [1, 0.5, 0.5, 0.5, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 0.5, 2],       # STEEL
    [1, 0.5, 1, 1, 1, 1, 2, 0.5, 1, 1, 1, 1, 1, 1, 2, 2, 0.5, 1]       # FAIRY
])
```

Fig. 2.1 Representation of the type effectiveness graph as an adjacency matrix in Python

To preserve the inherent asymmetry of a directed graph, edge directionality dictates how these coordinates are accessed. Querying across row index i $M_{i,*}$ extracts an offensive profile, while querying down column index j $M_{*,j}$ extracts a defensive profile. Thus, to evaluate the incoming damage vulnerabilities of any specific single type T, the system performs a column-slicing operation to yield an 18×1 inbound arc vector, denoted as:

$$v = M_{*,f(T)}$$

While single-type nodes map directly to a single column slice, dual-type configurations require resolving two target columns simultaneously. In game mechanics, dual-type defensive multipliers are calculated by multiplying the individual vulnerabilities of both component types. This relationship is mapped using the Hadamard product (element-wise multiplication). Let T_1 and T_2 be the primary and secondary types of a candidate Pokémon and let v_1 and v_2 be their respective 18×1 column vectors. The composite defensive vector $v_{composite}$ is resolved as:

$$v_{composite} = v_1 \odot v_2$$

$$\text{where } v_{\text{composite},i} = v_{1,i} \cdot v_{2,i} \quad \forall \quad i \in \{0,1,\dots,17\}$$

```python
def get_defensive_profile(combo):
    t1, t2 = combo
    v1 = ADJ_MATRIX[:, TYPE_TO_IDX[t1]]
    if t2 is None:
        return v1
    v2 = ADJ_MATRIX[:, TYPE_TO_IDX[t2]]
    return v1 * v2
```

Fig 2.2 The implementation of the Hadamard function into Python

To maintain uniform structural dimensions throughout the execution of the main algorithm, single-type candidates bypass this operation and return their sliced column vector directly. This formulation is scaled to accommodate all 171 legal type combinations while automatically preserving underlying game rules.

For instance, if a component type possesses a total immunity $(v_{1,i} = 0.0)$, the element-wise multiplication guarantees that the final entry is nullified, ensuring that immunities structurally override any compounding weaknesses or resistances inherited from the secondary type. Double weaknesses and resistances are likewise derived from situations where $(v_{1,i} = 0.5 \; and \; v_{2,i} = 0.5)$ resulting in $0.5 \times 0.5 = 0.25$. For double weaknesses, let $v_{1,i} \; and \; v_{2,i} \; both \; be$ 2.0, resulting in 4.0 (double weakness). A type resisted by one type of the composite whilst dealing double damage by another type of the composite will cancel out, causing a normal interaction as $0.5 \times 2 = 1$.

B. *Solution Space*

To begin, we must establish the solution space of 171 type combinations.

```
DRAFT_POOL = []
for i in range(18):
    DRAFT_POOL.append((TYPES[i], None))
    for j in range(i + 1, 18):
        DRAFT_POOL.append((TYPES[i], TYPES[j]))
```

Fig 2.3 Implementation of the draft pool in Python

We use combinations rather than permutations as, for example, Fire/Flying type Pokémon have the complete same type effectiveness profile with their Flying/Fire type counterparts. To represent single-type Pokémon, the code treats a “None” second type as a possibility. By the end of the double loop, the array would be filled in by the combinations.

C. *Objective Heuristics*

To find the most defensively optimal group configuration, the algorithm relies on a dynamic multi-criteria objective heuristic scoring function. For each iterative slot placement (from members two through six), the system determines the existing team's total exposure vector and calculates a scalar fitness score for every remaining candidate node within the solution space.

C.1. Mathematical Team Exposure Tracking

Before a candidate can be evaluated, the program computes the current collective vulnerability profile of the active team. This is achieved by aggregating the individual inbound arc vectors of all currently drafted members:

$$\mathbf{v}_{\text{team}} = \sum_{m \in P} \mathbf{v}_m$$

Let N represent the number of components currently present in the active list P. Because an unmitigated standard damage path introduces a structural multiplier of $1.0\times$ per node, a path index i is flagged as an active, unmitigated team vulnerability if its aggregated weight exceeds the current team size threshold:

$$\text{Vulnerable Condition: } \mathbf{v}_{\text{team}}[i] > N$$

This baseline register ensures that the objective function shifts its priority dynamically depending on whether a specific type path is currently exposing the cluster to danger or if the cluster is already sufficiently insulated down that path.

C.2. Environment Evaluation Scenarios

Every candidate defensive vector $v_{candidate}$ is passed through an evaluation loop across all 18 type indices. At each index i, the scoring mechanism updates the fitness register based on two mutually exclusive environmental conditions:

```
def evaluate_candidate_fitness(candidate_vector, active_team_vectors):

    # Calculate cumulative incoming path weights across the active team subgraph
    cumulative_team_exposure = np.sum(active_team_vectors, axis=0)
    current_team_size = len(active_team_vectors)

    score = 0
    for i in range(18):
        # Scenario A: Active cluster has an unmitigated shared vulnerability
        if cumulative_team_exposure[i] > current_team_size:
            if candidate_vector[i] == 0.0:
                score += 15  # Critical Immunity Cover
            elif candidate_vector[i] < 1.0:
                score += 8   # Critical Resistance Cover
            elif candidate_vector[i] > 1.0:
                score -= 12  # Compounded Vulnerability Penalty

        # Scenario B: Active cluster is defensively insulated down this arc path
        else:
            if candidate_vector[i] == 0.0:
                score += 4   # Luxury Immunity Buffer
            elif candidate_vector[i] < 1.0:
                score += 2   # Luxury Resistance Buffer
            elif candidate_vector[i] > 1.0:
                score -= 2   # Minor Exposure Penalty

    return score
```

Fig 2.4 Implementation of objective heuristics

Any new entry into the team is judged based on its capacity to actively balance the aggregate defensive profile of the existing cluster. The scoring mechanic does not evaluate the candidate vector in isolation; instead, it performs a comparative analysis against the active team's cumulative vulnerability vector across all eighteen type indices. If the system detects that the existing team members share an unmitigated vulnerability to a specific attacking element, the candidate is heavily rewarded for introducing a hard immunity $(+15)$ or a scaled resistance to that specific path $(+8)$. Conversely, if a candidate introduces a redundant weakness to an already exposed path, it receives a heavy penalty to prevent structural failure (-12). When the existing team is already structurally insulated against an

attacking type, any additional immunities or resistances provided by the candidate are classified as minor luxury buffers, receiving significantly lower priority scores, while minor exposure risks are met with nominal penalties.

C.3. Heuristic Weight Justification

The specific integer assignments function as calibrated scalar weights designed to force the greedy selection engine to mimic professional human team-building strategies. The rationale for the massive discrepancy between Scenario A and Scenario B points is based on two core meta-theoretical principles.

Under the Principle of Switchability, a defensive core in competitive team design is effective only if an incoming node can safely switch into a hostile arc intended for a vulnerable ally. By granting a maximum reward for an immunity and a scaled down reward for a resistance exclusively when the team is endangered, the algorithm prioritizes filling immediate critical defensive gaps over stacking unnecessary resistances on paths where the team is already stable.

Conversely, the compounding weakness boundary condition addresses the danger of stacking multiple shared weaknesses down a single damage path, which allows an opposing attacker to sweep the entire team structure using a single elemental coverage vector. The heavy negative penalty applied under Scenario A acts as a mathematical barrier within the loop, severely dropping the fitness profile of any candidate that would introduce an overlapping weakness to an already exposed path, thereby satisfying historical building constraints which state that a balanced cluster should never compound vulnerabilities beyond a manageable threshold.

D. The Greedy Algorithm and Execution Sequence

Once the finite solution space of 171 unique type combinations and the conditional objective scoring parameters are established, the global search sequence operates as an iterative greedy optimization loop. Rather than wasting computational resources processing billions of global permutations simultaneously, the algorithm scales linearly by sequentially appending the single most complementary type node to the active team cluster. The progression of this automated selection loop is divided into three functional phases that transform the raw matrix data into a balanced, cohesive six-member team. This iterative approach balances computational efficiency with structural depth, mirroring the analytical process a human analyst uses when building a team from scratch.

During the first phase, which handles randomized seed initialization, the optimization loop begins by deliberately forcing an exploratory step to maximize sample diversity across different program executions. Instead of starting from a fixed, hardcoded baseline, the system selects the very first team member from the pool of 171 candidates uniformly at random. This initial selection establishes a unique baseline vector profile with a localized set of strengths and flaws. The remaining five team slots are then filled deterministically to directly compensate for and insulate the specific structural vulnerabilities inherited from that original randomized seed, ensuring that every run yields a distinct yet structurally optimized configuration. By introducing this random seed, the program avoids repetitive, predictable outputs and forces itself to solve unique defensive puzzles every time it runs.

The second phase execution governs the localized heuristic search for slots two through six, where the system enters a sequential evaluation loop. The program replicates the active team state, aggregates the current team-wide exposure vector, and calculates a fitness score for every remaining available candidate in the pool. By parsing each candidate's resolved Hadamard vector through the multi-criteria objective heuristic rubric, the program determines exactly how well that candidate covers the team's active blind spots. This reduces the search space for each slot down to a simple linear evaluation of the remaining array elements, drastically improving processing speeds. Rather than stumbling blindly through permutations, the algorithm takes a measured look at its immediate choices, picking the option that provides the most immediate relief to the team's current defensive weaknesses.

The final phase resolves convergence and handles tie-breaking operations. The type combination that maximizes the total scalar fitness score is permanently appended to the chosen team list, and its corresponding defensive vector is committed to the active exposure matrix. Because of underlying symmetries in the type chart—where completely different dual-type combinations can yield identical defensive multiplier arrays—multiple candidates will frequently tie for the absolute highest local fitness score. The system resolves these deadlocks uniformly at random among the top-tier candidates, keeping the selection process dynamic rather than rigid. This complete sequence repeats recursively until the size of the chosen array equals six, terminating the greedy optimization loop and returning the fully insulated node cluster.

```python
def generate_optimized_team():
    pool = DRAFT_POOL.copy()
    chosen_combos = []
    chosen_vectors = []

    # Step 1: Randomized initialization step for exploratory diversity
    first_pick = random.choice(pool)
    chosen_combos.append(first_pick)
    chosen_vectors.append(get_defensive_profile(first_pick))
    pool.remove(first_pick)

    # Step 2: Iterative greedy candidate search selection loop
    while len(chosen_combos) < 6:
        max_fitness = float('-inf')
        best_candidates = []

        for candidate in pool:
            candidate_vector = get_defensive_profile(candidate)
            fitness = evaluate_candidate_fitness(candidate_vector, chosen_vectors)

            # Keep track of local optimization values
            if fitness > max_fitness:
                max_fitness = fitness
                best_candidates = [candidate]
            elif fitness == max_fitness:
                best_candidates.append(candidate)

        # Step 3: Handle structural convergence ties uniformly at random
        next_pick = random.choice(best_candidates)
        chosen_combos.append(next_pick)
        chosen_vectors.append(get_defensive_profile(next_pick))
        pool.remove(next_pick)

    return chosen_combos
```

Fig 2.5 Implementation of the greedy algorithm

```
==============================================
  COMPLEMENTARY GRAPH COMBINATION PROFILE
==============================================
Node Group 1: FIRE / GRASS
Node Group 2: NORMAL / STEEL
Node Group 3: WATER / GHOST
Node Group 4: GROUND / FLYING
Node Group 5: WATER / FAIRY
Node Group 6: STEEL / FAIRY
==============================================
```

Fig 2.6 Example of a successful run of the algorithm

CONCLUSION

This paper shows that competitive Pokémon team building can be successfully modeled and simplified using graph theory and basic matrix operations. By turning the type chart into a weighted directed graph and using the element-wise Hadamard product, we created a clear mathematical way to resolve dual-type calculations without messing up the dimensions of our data. The greedy algorithm completely solves the massive computational problem of team selection. Instead of wasting time calculating tens of billions of permutations, the sequential search loops through the 171 possible choices and finds highly optimized, complementary members one slot at a time in less than a second.

The actual success of the code comes down to how the objective function tracks weaknesses. By measuring team exposure against the current team size threshold, the program successfully flags when the team is in danger. The scoring system accurately mirrors the defensive priorities of a real human player. Giving high point values for immunities and resistances when a vulnerability is active forces the algorithm to build reliable defensive cores, while the heavy penalty for compounding weaknesses acts as a strict guardrail to prevent the team from being swept by a single attacking type. Finally, by starting the loop with a random choice and using random selection to break ties, the program avoids repetitive, rigid outputs and instead serves as a varied team generator across different runs.

While this non-weather defensive model is highly efficient, it leaves plenty of room for future upgrades. The solution space could easily be expanded beyond pure type combinations by linking a dataset of individual Pokémon species, their base stats, real abilities, and movepools. The algorithm also focuses on what is best at the time, not what is best overall, which is a flaw of greedy algorithms. Additionally, the objective function could be upgraded to handle a bipartite model that calculates offensive coverage and defensive stability at the same time. Ultimately, this implementation proves that straightforward computer science heuristics can easily take a complex, multi-layered competitive meta and reduce it to a fast, reliable, and automated system.

ACKNOWLEDGMENT

The author would like to express their sincere gratitude to **Dr. Ir. Rinaldi Munir, M.T.** for his consistent and earnest guidance throughout this semester. The author would also like to thank their family and friends for their outpour of support and inspiration throughout the process of writing this research paper.

REFERENCES

[1] H. Sahovic, “Algorithmic Pokémon Teambuilding,” 2025;

[2] T. Odland, “The best Pokémon party,” 2021;

[3] Reis, da Silva Oliviera, et al., “An Adversarial Approach for Automated Pokémon Team Building and Meta-Game Balance,” 2023.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 Juni 2025

Ariq Ulwan Hammam 13525084