Approximation of the Optimal Badge Collection Order in Pokemon Scarlet and Violet Using a Modified Greedy Traveling Salesman Problem Algorithm

Benedict Darrel Setiawan - 13524057
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: benedictdarrel572006@gmail.com , 13524057@std.stei.itb.ac.id

***Abstract*—Pokémon Scarlet and Violet are the latest mainline games in the Pokémon Series. Being the series' first ever delve into the open-world genre, the game brings a new and improved gameplay format that lets the player progress through the storyline in any order. This change inevitably results in the sudden spike in difficulty for players who try to find the most optimal path in order to speedrun the game. This paper determines the true optimal route to completing the game's three storylines by using a weighted directed graph representation. This is done utilizing a modified version of a greedy Travelling Salesman Problem algorithm, with the Nearest Neighbor Algorithm as the base, that the author constructed to adjust accordingly to the flow of gameplay.**

Keywords — Pokémon, Graph, Travelling Salesman Problem, Greedy Algorithm, Nearest Neighbor Algorithm, Speedrunning, Hamiltonian Cycle

I. Introduction

Fig 1. Pokémon Scarlet and Violet banner (Source: https://www.nintendo.com/ph/switch/scarlet_violet/index.html)

Pokémon Scarlet and Violet are the latest mainline games in the Pokémon series, being released on November 18, 2022. Like most new entries in the series, these games feature a brand new region, the Paldea region, which is based on the real-life Iberian Peninsula. However, what sets these games apart from previous installments is that they are the Pokémon series' first dive into the open-world genre.

The open-world nature of Scarlet and Violet allows players to progress through the game however they please. Usually, Pokémon games follow a formula of needing to gather all of the region's 8 Gym badges in a specific order to be able to enter the Pokémon League and beat the game. However, Scarlet and Violet break the tradition by not only letting the player gather the badges in any order they want, but also by increasing the number of badges they need to beat the game. There are a total of 18 badges that the player needs to collect, all of them representing the 18 Pokémon types. These badges are split into three different storylines, with 8 of them being the usual Gym Badges in the Path of Victory storyline, 5 of them being Star Badges from the Starfall Street storyline, and another 5 being Titan Badges from the Path of Legends storyline. While categorized as being different, these badges usually have the same method of obtaining them, which is typically by defeating a specific trainer or Pokémon.

This big shift from the usual format of Pokémon games has certainly made an impact in most aspects of the series. One of them is adding another layer of complexity to speedruns. Usually, speedruns of Pokémon games require the speedrunner to know the optimal path to get through each location in the game. However, in Scarlet and Violet, the speedrunner now also has to consider what the optimal badge-collecting order is. There are also a multitude of other factors that complicate this process even more, such as the absence of level scaling.

This paper will determine the optimal route to complete all three storylines in the game, assuming that the player doesn't have access to fast travel, utilizing a modified Traveling Salesman Problem (TSP) algorithm on a graph with dynamic adjacency. The modifications are used to accommodate factors such as levels and prerequisite objectives for completing another objective. The application of TSP is used due to the starting point of all three storylines and the final point of two of the storylines are within the location known as Mesagoza on the map. Therefore, we can conclude that the optimal route for completing the game is indeed a Hamilton cycle, which means it can be solved by application of TSP.

II. THEORETICAL FRAMEWORK

A. Graph

Graphs are generally used to represent discrete objects and the relations between said objects. A graph is defined as $G(V, E)$, where V is a non-empty set of vertices and E is a set consisting of edges that connect the vertices. Based on the way that the direction of the edges is oriented, graphs can be classified into two types, which are:

1. Undirected Graphs, which have no discernible orientation of direction

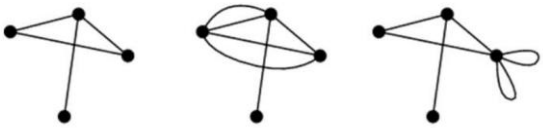

Fig 2. Undirected Graphs (Source: [1])

2. Directed Graphs, a graph in which every edge is given a direction

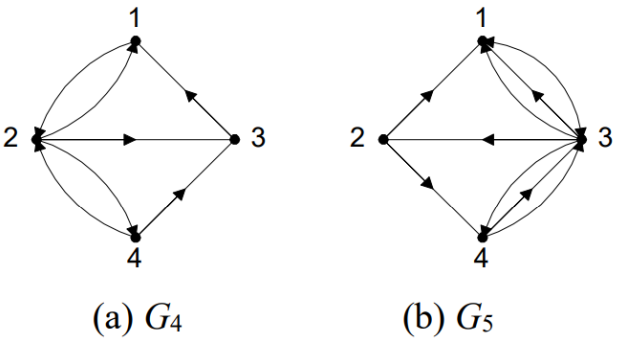

Fig 3. Directed Graphs (Source: [1])

Some of the various Graph terminologies that are relevant to this paper are as follows:

1. Path

A sequence of vertices in which each vertex is adjacent to the one before and next after it.

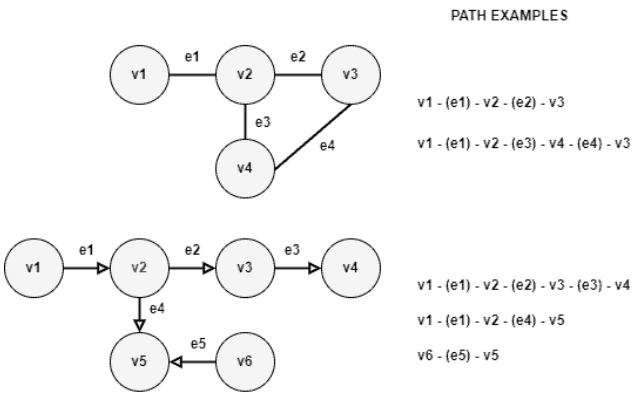

Fig 4. Path Examples (Source: [2])

2. Cycle

A path that starts and ends on the same Vertex

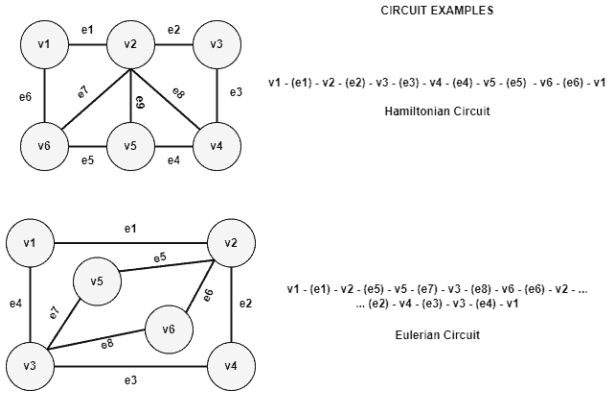

Fig 5. Cycle Examples (Source: [2])

3. Weighted Graphs

A graph in which each edge contains a value or weight

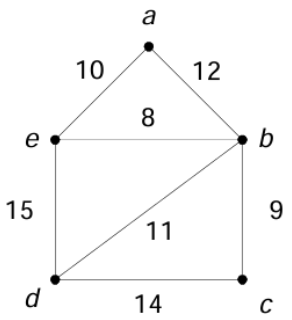

Fig 6. A Weighted Graph (Source: [1])

4. Hamiltonian Graphs

A Hamiltonian path is a path that visits every single vertex on the graph exactly once. A Hamiltonian circuit is a path that visits every single vertex on the graph exactly once and returns to the starting vertex at the end of the path. A graph that contains a Hamiltonian circuit is defined as a Hamiltonian graph, while one that contains a Hamiltonian path is regarded as a semi-Hamiltonian graph.

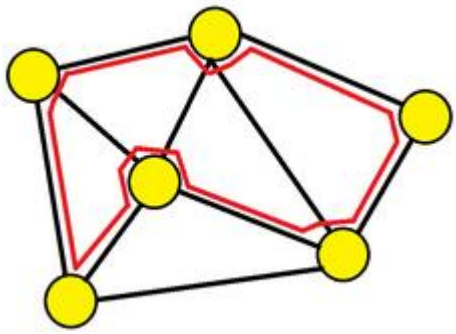

Fig 7. A Hamiltonian Graph (Source: https://en.wikipedia.org/wiki/Hamiltonian_path)

B. Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a theoretical problem that states of a salesman trying to visit several cities exactly once and then return to back to the city they started at. The goal of this problem is to find the shortest possible route for the salesman to travel across. The problem practically implores to find the shortest Hamiltonian cycle in a graph where the vertices represent each city the salesman has to visit.

TSP has become a popular problem in computation due to there being almost no time-effective method to definitively find the shortest possible route. Two of the most popular approaches to solving the TSP are as follows:

1. The Brute Force Method

The brute-force approach calculates and compares every permutation of the routes and picks the one that has the least distance value. This method is popular in picking apart TSP with a small number of cities/vertices, due to its objective 100% accuracy. However, due to its time complexity of $O(n!)$, this method is extremely impractical and unfeasible when tackling TSP with a large number of vertices.

2. The Nearest Neighbor Algorithm

The Nearest Neighbor Algorithm (NNA) is a greedy approach to solving TSP. It is a relatively simple algorithm to comprehend and quick to execute due to its time complexity of $O(n^2)$. However, the trade-off is that the algorithm is prone to faults and blind spots that affect the seemingly shortest possible cycle it produces. The step-by-step algorithm of NNA is as follows:

1. Pick a starting city.
2. Set it as the current city and put it in the list of currently visited cities.
3. Move to the city closest to the current one.
4. Repeat 2-3 until all cities have been visited.
5. Return to the starting city.

C. Badges in Pokémon Scarlet and Violet

Pokémon Scarlet and Violet have a total of 18 badges to obtain, each one belonging to one of the game's three storylines.

1. Victory Road (Gym Badges)

There are a total of 8 Gym Badges in Pokémon Scarlet and Violet. They can be obtained after defeating a Gym Leader in a battle. Obtaining Gym Badges will increase the level cap that gives the player the ability to command a Pokémon that is caught at said level cap or below. However, the player is still able to command Pokémon that are above the level cap if the Pokémon is caught at a lower level. After obtaining all 8 Gym Badges, the player can challenge the Pokémon League, which is located in Mesagoza.

Badge Attributes		
Gym Location	**Type Specialization**	**Obeying Pokémon Level Cap**
Cortondo	Bug	25
Artazon	Grass	30
Levincia	Electric	35
Cascaraffa	Water	40
Medali	Normal	45
Montenevera	Ghost	50
Alfornada	Psychic	55
Glaseado	Ice	100

Fig 8. List of Gym Badges and Their Attributes (Source: [4])

2. Path of Legends (Titan Badges)

There are a total of 5 Titan Badges in Pokemon Scarlet and Violet. They can be obtained after defeating a certain Titan Pokémon. Obtaining Titan Badges will give the player unique traversal abilities. After obtaining all 5 Titan Badges, the player can initiate the final battle of the storyline in Poco Path Lighthouse.

Badge Attributes		
Titan Pokemon	**Type Specialty**	**Ability Given**
The Stony Cliff Titan	Rock	Dash
The Open Sky Titan	Flying	Swim
The Lurking Steel Titan	Steel	High Jump
The Quaking Earth Titan	Ground	Glide
The False Dragon Titan	Dragon	Climb

Fig 9. List of Titan Badges and Their Attributes (Source: [4])

3. Starfall Street (Star Badges)

There are a total of 5 Star Badges in Pokemon Scarlet and Violet. They can be obtained after defeating the Team Star Squad Bosses. Obtaining Star Badges will allow the player to make a greater variety of TMs using the TM Machine. After obtaining all 5 Star Badges, the player can initiate the final battle of the storyline in Mesagoza.

Badge Attributes	
Star Squad	**Type Specialty**
Segin Squad	Dark
Schedar Squad	Fire
Navi Squad	Poison
Ruchbah Squad	Fairy
Caph Squad	Fighting

Fig 10. List of Star Badges and Their Attributes (Source: [4])

III. Method and Results

A. Initial Graph Representation

The first step in modeling the graph is to establish what each of the graph elements represents. In this case, each node in our graph represents a location where the player can obtain a certain badge or complete one of the three storylines. Below is an image that details where each of the 18 badges in the game can be obtained.

Fig 10. A Map that Shows Where All 18 Badges Are Located (Source: https://www.mandatory.gg/en/pokemon/ecarlate-violet/la-carte-de-paldea-le-monde-de-pokemon-ecarlate-et-violet/html)

From there, we can add two more notable locations to the node list. One being Poco Path Lighthouse, which is where the player can complete the Path of Legends storyline. The other being Mesagoza, which is not only the starting location for all three storylines, but also where the player is supposed to finish the Victory Road and Starfall Street storylines.

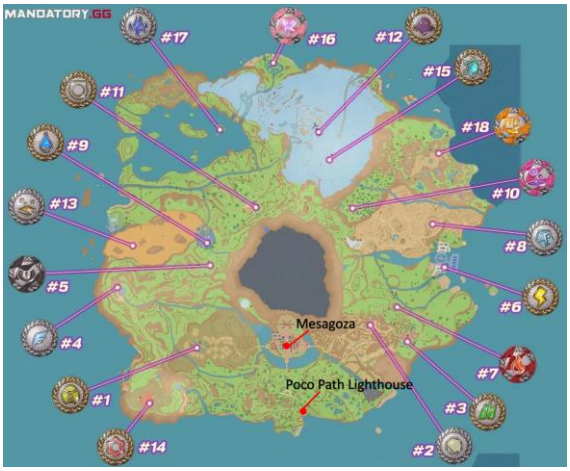

Fig 11. An Updated Map that Details Mesagoza and Poco Path Lighthouse

No.	Node Attributes		
	Location	**Pixel Coordinates (X, Y)**	**Highest-level Pokemon**
0	Mesagoza	516, 618	10
1	Cortondo Gym	352, 623	15
2	Stony Cliff Titan's Nest	667, 582	16
3	Artazon Gym	735, 611	17
4	Open Sky Titan's Nest	208, 515	19
5	Segin Squad Base	376, 473	21
6	Levincia Gym	803, 476	24
7	Schedar Squad Base	716, 549	27
8	Lurking Steel Titan's Nest	780, 399	28
9	Cascarrafa Gym	371, 434	30
10	Navi Squad Base	636, 370	33
11	Medali Gym	461, 368	36
12	Montenevera Gym	573, 231	42
13	Quaking Earth Titan's Nest	236, 439	44
14	Alfornada Gym	266, 723	45
15	Glaseado Gym	592, 282	48
16	Ruchbah Squad Base	490, 107	51
17	False Dragon Titan's Nest	394, 226	55
18	Caph Squad Base	790, 271	56
19	Poco Path Lighthouse	547, 743	63

Fig 12. Details of Each Node to be Used in the Graph

Next, we need to establish what the adjacency between two nodes in the graph represents. The simplest option would be to establish that an adjacency represents the ability to move from one location to another. Since the game is open-world, this would mean that our graph is a complete one, should we follow this route. However, that option would be unfeasible due to the game's lack of level scaling.

The lack of level scaling in Pokémon Scarlet and Violet results in every battle having a fixed level of difficulty. For instance, if the player begins in Mesagoza with the highest-level Pokémon on their team at level 10, they are able to attempt to earn the Glaseado Gym Badge in theory. However, winning a battle becomes impractical because their team is 38 levels lower than the Gym Leader's highest-level Pokémon.

To counteract this problem, the assumption is made that the player's current 'Highest-level Pokémon' is always set to the 'Highest-level Pokémon' of the latest obtained badge. It is also assumed that the player can obtain a badge that has a 'Highest-level Pokémon' value of at most L levels higher than the

player's current 'Highest-level Pokémon' value. From these assumptions, the following relation between two nodes in set P can be defined.

$R_1 = \{(u, v) \mid$ 'Highest-level Pokémon' of $u \leq$ 'Highest-level Pokémon of v' $\leq$ 'Highest-level Pokémon' of $u + L, u \neq v\}$

To determine the value of L, we need to simply find two consecutive nodes with the largest difference in their highest-level Pokémon. In this case, it's Caph Star Base and Poco Path Lighthouse, with their difference being 7. These assumptions are made so that the algorithm will be able to traverse the graph according to how the game intends the player to progress.

The Python code to initialize the current graph is as follows,

```
import networkx as nx
import os, math, copy
import matplotlib.pyplot as plt
script_dir = os.path.dirname(os.path.abspath(__file__))

nodefile = file_path('Badge_Data.txt')

LevelRange = 7


def initializeNodes(file, G, pos) :
    data = open(file_path(file))
    line = data.readline()[:-1]
    n = 0
    while line != '======================' :
        name, attr = line.split(' : ')
        coords, lvl = attr.split(' ; ')
        nodecoords=[]
        for x in coords.split(',') :
            nodecoords.append(int(x))
        G.add_node(n, name=name, level=int(lvl),
                   x=nodecoords[0],
                   y=nodecoords[1])
        pos[n] = nodecoords
        n += 1
        line = data.readline()[:-1]

def addInitialEdges(G) :
    for i in (G.nodes) :
         offset = 0
         for j in (G.nodes) :
            if    (G.nodes[j]['level'] <=
                     G.nodes[i]['level'] + LevelRange)
                     and (G.nodes[i]['level'] <=
                     G.nodes[j]['level']) and (i != j):
                  distance = pathDistance(G,i,j)
                  G.add_edge(i, j, weight=distance)
                offset += 1
```

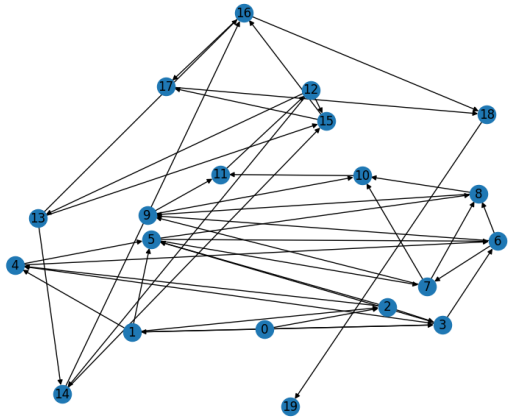

Fig 13. Initial Graph Representation

To prove the existence of a Hamiltonian Circuit in our future graph, we need to confirm that the above graph has at the very least one Hamiltonian Path. This is provable by the fact that the nodes in the initial graph are numbered based on the order of the highest-level Pokémon attribute, starting from lowest to highest. So, a path created from following the order of the numbered nodes (0, 1, 2, 3, etc.) will lead us to the Hamiltonian Path shown below

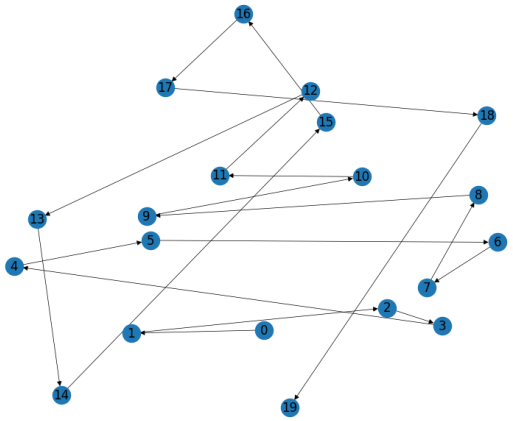

Fig 14. A Hamiltonian Path in Fig 13

B. Determining The Distance Between Nodes

To fully realize our graph, we must determine the weight that is given to each edge of the graph. In this problem, the weight represents the distance between the two nodes that the edge connects. However, a problem arises when we consider the map layout further.

1. Area Zero

Area Zero is a giant crater located in the middle of Paldea. Due to its tall and unclimbable nature, crossing it would be either difficult or even impossible to do.

Fig 14. Area Zero Border

To counteract this, we simply need to find the shortest path by going around the border. The steps of the algorithm to do so are as follows:

1. Define Area Zero as a polygon
2. Determine points *A* and *B*, and put them in list *P*
3. Create a line from point *A* to point *B*
4. If said line intersects through two points in the Area Zero polygon, find the vertex of the polygon that is closest to the middle point of the line. And then put said vertex as *C* into list *P* in between *A* and *B*
5. Create a line from every consecutive Point in *P*
6. Repeat 4-5 until no line is formed from P that intersects through two points in the Area Zero polygon
7. Delete any points in list *P* that are between two points that create a line that doesn't cut through the Area Zero polygon
8. Sum up the distance between each consecutive point in list *P*

Below is the implementation of said algorithm in the Python language using the Shapely library and a visualization of the result.

```
from shapely.geometry import Point, Polygon

AreaZeroPoints = [
    (404, 491), (409, 502), (401, 503), (395, 515),
    (400, 522), (413, 528), (413, 541), (433, 558),
    (449, 578), (482, 586), (499, 594), (496, 582),
    (519, 564), (556, 600), (576, 594), (606, 601),
    (615, 589), (617, 547), (619, 474), (590, 416),
    (569, 388), (547, 397), (518, 379), (495, 389),
    (469, 391), (448, 407), (439, 436), (415, 453),
    (412, 479)
]

def isPathIntersectingAreaZero(point1, point2) :
    line = LineString([point1, point2])
    AreaZeroPolygon = Polygon(AreaZeroPoints)
    if line.intersects(AreaZeroPolygon) :
        if line.intersection(AreaZeroPolygon).geom_type ==
           'Point' :
            state = False
        else :
            state = True
    else :
        state = False
    return state


def pathDistanceAroundAreaZero(point1, point2) :
    CurrPath = [point1, point2]
    while True :
        state = False
        for I in [0, len(CurrPath) - 2]:
            if isPathIntersectingAreaZero(CurrPath[i],
               CurrPath[i+1]) :
                state = True
                middle =  idpoint(CurrPath[i], CurrPath[i+1])
                closestPoint = AreaZeroPoints[0]
                for x in AreaZeroPoints :
                    state2 = True
                    for y in CurrPath :
                        if (x == y) :
                            state2 = False
                            break

                    if state2 and (pointDistance(closestPoint,
                       middle) > pointDistance(x, middle)) :
                        closestPoint = x
                CurrPath.insert(I + 1, closestPoint)
        if not(state) :
            break

    while not(isPathIntersectingAreaZero(CurrPath[0],
              CurrPath[2])) :
        CurrPath.pop(1)
    while not(isPathIntersectingAreaZero(CurrPath[len(CurrPath) -
              1], CurrPath[len(CurrPath) - 3])) :
        CurrPath.pop(len(CurrPath) - 2)

    distance = 0
    for I in range(len(CurrPath) - 1) :
        distance += pointDistance(CurrPath[i], CurrPath[I + 1])

    return distance

def pathDistance(G, node1, node2) :
    nodepoint1 = (G.nodes[node1]['x'], G.nodes[node1]['y'])
    nodepoint2 = (G.nodes[node2]['x'], G.nodes[node2]['y'])
    if isPathIntersectingAreaZero(nodepoint1, nodepoint2) :
        distance = pathDistanceAroundAreaZero(nodepoint1,
                                              nodepoint2)
        path_shape = 'modified'
    else :
        distance = pointDistance(nodepoint1, nodepoint2)
        path_shape = 'straight'

    return distance, path_shape
```

Fig 15. An Example of a Modified Path Visualized

2. Alternate Alfornada Route

Alfornada is a city located in the most southwestern part of the Paldea region. What makes it different from most cities in the game is that the main path to the city, which is Alfornada Cavern, can only be traversed once the player has obtained the high jump or the climb ability. However, there exists a longer alternate path to the city that doesn't require any abilities to be traversed.

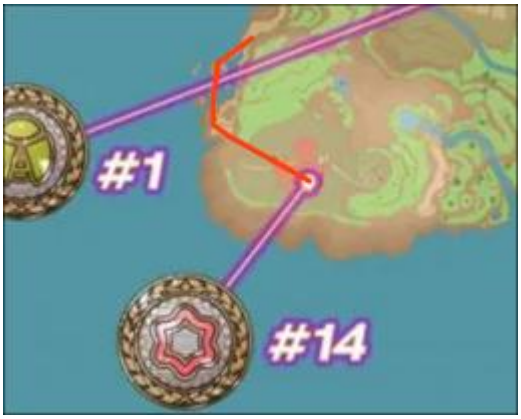

Fig 16. An Alternate Route to Alfornada

The solution to this problem implemented in the Python language is as follows:

```
AlfornadaAlternatePathPoints = [
    (236, 654),
    (221, 666),
    (220, 683),
    (219, 696),
    (227, 701)
]

def AlfornadaAlternatePathDistance(G) :
    AlfornadaCoords = (G.nodes[14]['x'], G.nodes[14]['y'])
    distance = 0
    for i in range(len(AlfornadaAlternatePathPoints) - 1) :
        distance += pointDistance(AlfornadaAlternatePathPoints[i],
AlfornadaAlternatePathPoints[i + 1])
    distance +=
pointDistance(AlfornadaAlternatePathPoints[len(AlfornadaAlternatePathPoints) - 1],
              AlfornadaCoords)
    return distance


def pathDistance(G, node1, node2, visitedNodes) :
```

```
    nodepoint1 = (G.nodes[node1]['x'], G.nodes[node1]['y'])
    nodepoint2 = (G.nodes[node2]['x'], G.nodes[node2]['y'])
    climb_obtained = False
    alternate_distance = 0
    for x in visitedNodes :
        if (x == 17) or (x == 8) :
            climb_obtained = True

    if (node2 == 14) and not(climb_obtained) :
        nodepoint2 = AlfornadaAlternatePathPoints[0]
        alternate_distance += AlfornadaAlternatePathDistance(G)

    if isPathIntersectingAreaZero(nodepoint1, nodepoint2) :
        distance = pathDistanceAroundAreaZero(nodepoint1, nodepoint2)
        path_shape = 'modified'
    else :
        distance = pointDistance(nodepoint1, nodepoint2)
        path_shape = 'straight'

    if alternate_distance > 0 :
        path_shape = 'modified'
        distance += alternate_distance
    return distance, path_shape
```

C. *Dynamic Graph Progression*

To fully represent Pokémon Scarlet and Violet's gameplay flow, the graph needs to be dynamic. The reasons are as follows:

1. The player needs to be able to obtain badges, which have a 'highest-level Pokémon' value lower than their latest obtained badge, while also not lowering their actual 'highest-level Pokémon'.
2. The player only gains the ability to cross bodies of water after they obtain the Open Sky Titan's Badge (Node 4). Due to this, the player cannot obtain the False Dragon Titan's Badge (Node 17) until then, due to needing to cross a body of water.
3. The player is only able to do the Poco Path Lighthouse objective (Node 19) after they obtain all 5 Titan Badges (Nodes 2, 4, 8, 13, 17).
4. Alfornada's main route can only be traversed when the player has until the player has obtained either the high jump ability from the Lurking Steel Titan's Badge (Node 8) or the climb ability from the False Dragon Titan's Badge (Node 17). Before gaining either of those abilities, the player must take the alternate longer route to reach the city.
5. To ensure that every node has been visited before finishing the cycle, the player must only be able to come back to Mesagoza (Node 0) after visiting all other nodes.

So, each time a node is visited, the graph will update its adjacencies according to the rules above. This means the algorithm for finding the shortest possible path must save not only the currently visited nodes but also the 'highest-level Pokémon' value among the set of currently visited nodes. The implementation in the Python language are as follows:

```
def currentLevel(G, VisitedNodes) :
    Curr = G.nodes[VisitedNodes[0]]['level']
    for l in VisitedNodes :
        if (G.nodes[l]['level'] > Curr) :
            Curr = G.nodes[l]['level']

    return Curr

def updateGraphRoutes(G, VisitedNodes) :
    CurrentNode = VisitedNodes[len(VisitedNodes) - 1]
    CurrentLevel = currentLevel(G, VisitedNodes)
    state2 = False

    for Dest in (G.nodes) :
        state = True
        state1 = True
        state2 = True
        for (_,v) in (G.edges(CurrentNode)) :
            if (v == Dest) :
                state = False
                break
        for x in VisitedNodes :
            if (x == Dest) :
                if (len(VisitedNodes) != len(G.nodes)) or (x !=
                   VisitedNodes[0]) :
                    state = False
                    break

        if (Dest == 17) :
            state1 = False
            for y in VisitedNodes :
                if y == 4 :
                    state1 = True
                    break

        if (Dest == 19) :
            state2 = False
            z2 = 0
            for z in VisitedNodes :
                if (z == 2) or (z == 4) or (z == 8) or (z == 13) or
                   (z == 17) :
                    z2+=1

            if z2 == 5 :
                state2 = True
        if state and state1 and state2 and (CurrentLevel + LevelRange
           >= G.nodes[Dest]['level']) :
            distance, shape = pathDistance(G, CurrentNode, Dest)
            G.add_edge(CurrentNode, Dest, weight=round(distance),
                       shape=shape)
```

D. *Determining the Shortest Possible Hamiltonian Circuit*

To determine the shortest possible Hamiltonian circuit of the graph, the author has employed the use of the Nearest Neighbor algorithm as a basis to solve the Traveling Salesman Problem. The said algorithm is used due to its relatively simple, fast, and easily modifiable algorithmic framework. To accommodate the changes made due to the dynamic nature of the graph, the Nearest Neighbor algorithm used for this problem is as follows:

1. Pick a starting node.
2. Set it as the current node, and put it in the list of currently visited nodes
3. Move to the node with the lowest distance value that is connected to the current node.
4. If the node's 'highest-level Pokemon' value is higher than the current highest value, set it to said value.
5. Update the graph's adjacencies according to the current node.
6. Repeat steps 2-5 until all nodes are in the list of currently visited nodes.
7. Return to the starting node.

Using the algorithm above, the following Python code, used to find the shortest possible Hamiltonian circuit, is produced:

```
def closestPossibleNode(G, VisitedNodes) :
    if len(VisitedNodes) == len(G.nodes) :
        return VisitedNodes[0]
    CurrentNode = VisitedNodes[len(VisitedNodes) - 1]
    state = True
    for (_,x) in G.edges(CurrentNode) :
        state1 = True
        for y in VisitedNodes :
            if x == y :
                state1 = False
                break

        if state1 :
            if state :
                closestNode = x
                state = False
            else:
                if  (G[CurrentNode][x]['weight'] <
                     G[CurrentNode][closestNode]['weight']) :
                    closestNode = x

    return closestNode
```

```
def findShortestCycle(G) :
    G2 = nx.DiGraph()
    pos2 = {}
    initializeNodes(nodefile, G2, pos2)
    addInitialEdges(G)
    visitedNodes = [0]
    n = 0
    while len(visitedNodes) <= len(G.nodes) :
        n += 1
        CN = closestPossibleNode(G, visitedNodes)
        visitedNodes.append(CN)
        G2.add_edge(visitedNodes[len(visitedNodes) - 2], CN,
                    weight = G[visitedNodes[len(visitedNodes) -
                               2]][CN]['weight'],
                    shape = G[visitedNodes[len(visitedNodes) -
                              2]][CN]['shape'])
        updateGraphRoutes(G, visitedNodes)
    print(f'Total Distance :  {currentDistanceTraveled(G,
          visitedNodes)}')
```

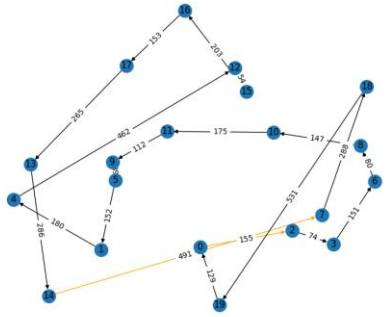

Fig 17. Shortest Possible Hamiltonian Circuit using the Nearest Neighbor Algorithm

The figure above shows the result of the Nearest Neighbor Algorithm trying to find the shortest possible Hamiltonian circuit with a total relative distance of 4127. However, said result is seemingly still not the shortest possible Hamilton cycle that the graph contains. This is due to the Nearest Neighbor Algorithm having a relatively shallow analysis of the graph. Since it will always go to the nearest possible node, other routes that are further in the short term but might be shorter in the long term are ignored.

To find the true shortest possible Hamiltonian Circuit of the graph, the author has modified and improved upon the Nearest Neighbor Algorithm. Instead of only analyzing the nodes that are adjacent to the current node, the algorithm will now analyze each possible option and iterate a route for each using the previous Nearest Neighbor Algorithm. It will then calculate the distance that each route has and pick the node that leads to the shortest route. The complete steps for the algorithm are as follows:

1. Set it as the current node and put it in the list of currently visited nodes.
2. Pick a node that is adjacent to the current node.
3. Iterate a cycle with said node as the next node using the previous Nearest Neighbor Algorithm.
4. Calculate the distance of said cycle.
5. Pick another node that is adjacent to the current node.
6. Repeat 4-5 until all possible options are explored.
7. Move to the node that leads to the cycle with the shortest distance.
8. Repeat 2-8 until all nodes are in the list of currently visited nodes.
9. Return to the starting node.

The realization of the algorithm above into Python code and the resulting visualization are as follows:

```
def nearestLoopLength(G, VisitedNodes, NextNode) :
    G1 = copy.deepcopy(G)
    CurrVisited = []
    for x in VisitedNodes :
        CurrVisited.append(x)
    DestNode = NextNode
    while len(CurrVisited) <= len(G1.nodes) :
        CurrVisited.append(DestNode)
        updateGraphRoutes(G1, CurrVisited)
        #print(CurrVisited)
        if (len(CurrVisited) <= len(G1.nodes)) :
            DestNode = closestPossibleNode(G1, CurrVisited)

    return currentDistanceTraveled(G1, CurrVisited), CurrVisited

def recommendedNextNode(G, VisitedNodes) :
    if len(VisitedNodes) == len(G.nodes) :
        return VisitedNodes[0]
    CurrentNode = VisitedNodes[len(VisitedNodes) - 1]
    state = True
    for (_,x) in G.edges(CurrentNode) :
        statel = True
        for y in VisitedNodes :
            if x == y :
                statel = False
                break

        if statel :
            if state :
                nextNode = x
                state = False
            else:
                if (nearestLoopLength(G, VisitedNodes, x)[0] <
                   nearestLoopLength(G, VisitedNodes, nextNode)[0]) :
                    nextNode = x

    return nextNode

def findShortestCycleVer2(G) :
    G2 = nx.DiGraph()
    pos2 = {}
    initializeNodes(nodefile, G2, pos2)
    addInitialEdges(G)
    visitedNodes = [0]
    n = 0
    while len(visitedNodes) <= len(G.nodes) :
        n += 1
        CN = closestPossibleNode(G, visitedNodes)
        visitedNodes.append(CN)
        G2.add_edge(visitedNodes[len(visitedNodes) - 2], CN,
                    weight = G[visitedNodes[len(visitedNodes) -
                               2]][CN]['weight'],
                    shape = G[visitedNodes[len(visitedNodes) -
                              2]][CN]['shape'])
        updateGraphRoutes(G, visitedNodes)
    print(f'Total Distance :  {currentDistanceTraveled(G,
          visitedNodes)}')
```

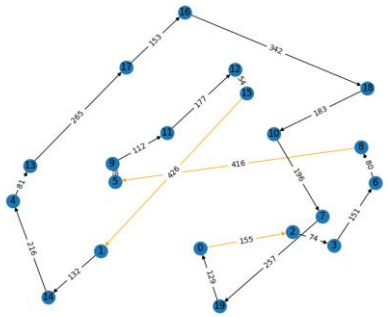

Fig 18. Shortest Possible Hamiltonian Circuit using the improved Nearest Neighbor Algorithm

The figure above shows the results of the improved version of the Nearest Neighbor Algorithm attempting to find the shortest possible Hamiltonian circuit, with a total relative distance of 3623. While significantly shorter than the previous cycle produced, further optimization is still possible. Upon observation, the author found that rearranging the order of node 10 from after node 16 to after node 8 results in an even shorter cycle. This change occurs because the sum of the weights on edge (8, 10), edge (5, 10), and edge (18, 7) is less than the sum of the weights on edge (8, 5), edge (18, 10), and edge (10, 7). Consequently, the algorithm still has some blind spots that need addressing, indicating that further improvements can be made.

After further analysis, the reason behind this specific blind spot is that the closest adjacent node to node 10 is node 11. So, when the algorithm is trying to find the best possible option to take when the current node is node 8, the node 10 operation will lead to a longer route due to its next closest node being node 11. When in actuality, the best possible route to take is indeed node 10, but with node 5 being the next node after it instead of node 11.

To solve this issue efficiently, the author has employed a similar method that was originally used to identify the problem in order to improve the previous version of the algorithm. This method involves inserting another node adjacent to the current one, positioned between it and the next planned node. The newest version of the algorithm, after applying this improvement, is as follows:

1. Pick a starting node.
2. Set it as the current node and put it in the list of currently visited nodes.
3. Pick a node that is adjacent to the current node.
4. Iterate a cycle with said node as the next node using the previous Nearest Neighbor Algorithm.
5. Calculate the distance of said cycle.
6. Repeat 4-5 until all possible options are explored.
7. Pick the node that leads to the cycle with the shortest distance and save a copy of said cycle.
8. Pick another node that is adjacent to the current node, delete it from the cycle, and re-insert it between the current node and the planned next node.
9. Calculate the distance of said cycle. If it is shorter than the previously saved cycle, then set it as the current cycle and set the inserted node as the planned next node.
10. Repeat 9-10 until the shortest possible cycle of the iteration is found.
11. Move to the planned next node.
12. Repeat 2-13 until all nodes are in the list of currently visited nodes.
13. Return to the starting point.

The realization of the algorithm above into Python code and the resulting visualization are as follows:

```
def recommendedNextNode(G, VisitedNodes) :
    if len(VisitedNodes) == len(G.nodes) :
        return VisitedNodes[0]
    CurrentNode = VisitedNodes[len(VisitedNodes) - 1]
    state = True
    for (_,x) in G.edges(CurrentNode) :
        statel = True
        for y in VisitedNodes :
            if x == y :
                statel = False
                break

        if statel :
            if state :
                nextNode = x
                state = False
            else:
                if (nearestLoopLength(G, VisitedNodes, x)[0] <
                   nearestLoopLength(G, VisitedNodes, nextNode)[0]) :
                    nextNode = x

    newNextNode = nextNode
    plannedLoopPath = nearestLoopLength(G, VisitedNodes, nextNode)[1]
    while True :
        shortestLoopPath = plannedLoopPath
        for (_,x) in G.edges(CurrentNode) :
            statel = True
            for y in VisitedNodes :
                if x == y :
                    statel = False
                    break

            if statel  and x != nextNode :
                if (currentDistanceTraveled(G, shortestLoopPath) >
                    currentDistanceTraveled(G,
                    moveTo(plannedLoopPath, len(VisitedNodes), x))) :
                    shortestLoopPath = moveTo(plannedLoopPath,
                                              len(VisitedNodes), x)
                    newNextNode = x
        plannedLoopPath = shortestLoopPath
        if nextNode == newNextNode :
            break
        else :
            nextNode = newNextNode

    return nextNode
```

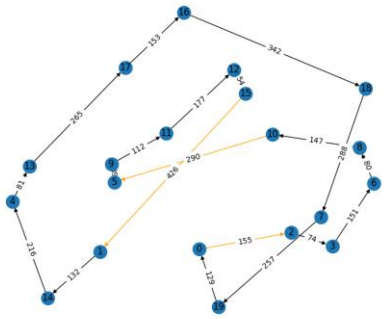

Fig 19. Shortest Possible Hamiltonian Circuit using the further improved Nearest Neighbor Algorithm

E. Discussion

Using the new and improved Nearest Neighbor algorithm, The author was able to determine the most optimal route to complete all three of the game's storylines. However, enhancing the Nearest Neighbor algorithm has significantly increased its time complexity. The original NN algorithm has a Big O notation of $O(n^2)$, where n represents the number of nodes in the graph. Since the second version integrates the original NN algorithm, it has an increased notation of $O((n^2)^2)$ or $O(n^4)$. And since the third version nests another process into the nested loop, it has a notation of $O((n^2 + n^2)^2)$ or $O(n^4)$. Nonetheless, this is still far more efficient than using the brute-force method, which has a notation of O(n!). The trade-off for the increased time complexity is deeper analysis and significantly better accuracy that the algorithm provides.

Another major factor in the algorithm's attempt to find the shortest possible path is *L*, or the level range in which decides which badge the player will go for next. The *L* value used in

this paper is 7, which is the minimal value needed for there to be at least one cycle. However, this value can always be adjusted accordingly to the player's skill, and can even lead to an even shorter route than the one found in this paper.

IV. CONCLUSION

Using an approach based on the Traveling Salesman Problem, the author has concluded that the optimal badge collection route to take to complete all three of Pokémon Scarlet and Violet's storylines is as follows:

1. Stony Cliff Titan Badge (Node 2)
2. Artazon Gym Badge (Node 3)
3. Levincia Gym Badge (Node 6)
4. Lurking Steel Titan Badge (Node 8)
5. Navi Squad Star Badge (Node 10)
6. Segin Squad Star Badge (Node 5)
7. Cascarrafa Gym Badge (Node 9)
8. Medali Gym Badge (Node 11)
9. Montenevera Gym Badge (Node 12)
10. Glaseado Gym Badge (Node 15)
11. Cortondo Gym Badge (Node 1)
12. Alfornada Gym Badge (Node 14)
13. Open Sky Titan Badge (Node 4)
14. Quaking Earth Titan Badge (Node 13)
15. False Dragon Titan Badge (Node 17)
16. Ruchbah Squad Star Badge (Node 16)
17. Caph Squad Star Badge (Node 18)
18. Schedar Squad Star Badge (Node 7)
19. Path of Legends Completion (Node 19)
20. Victory Road and Starfall Street Completion (Node 0)

This leads to a route that has a total relative distance value of 3568. In finding these results, the author was also able to improve upon the Nearest Neighbor Algorithm used to solve the TSP by finding its faults, such as:

1. The NN algorithm's inability to see past one route when other routes could lead to a shorter path/cycle
2. The second version of the author's NN algorithm's inability to reanalyze the order of the current route taken and whether it is truly the shortest route taken.

By enhancing the regular NNA, the author developed an algorithm that provides a deeper and more accurate analysis of the graph while also being significantly more efficient in terms of time compared to a brute force algorithm.

V. APPENDIX

GitHub repository for the source code made for this research paper:
https://github.com/BenedictD-RH/Badge-Collection-Optimization-Using-an-Improved-NNA

Explanation of the research paper in video form: https://youtu.be/luZBkQmU5y0

VI. ACKNOWLEDGEMENT

First of all, the author would like to give his deepest gratitude to the Almighty God for giving him the health and capacity to finish this paper and this semester. The author would also like to thank Mr. Rinaldi Munir for his guidance and teachings on Discrete Mathematics, which provided the foundations for the concepts used in this paper.

VII. REFERENCE

[1] R. Munir, "Graf (Bagian 1)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf. [Accessed: Jun. 14, 2025].

[2] R. Munir, "Graf (Bagian 2)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf. [Accessed: Jun. 14, 2025].

[3] W3Schools, "DSA The Traveling Salesman Problem", W3Schools [Online]. Available: https://www.w3schools.com/dsa/dsa_ref_traveling_salesman.php. [Accessed: Jun 15, 2025].

[4] Bulbagarden, "Badge", Bulbapedia. [Online]. Available: https://bulbapedia.bulbagarden.net/wiki/Badge. [Accessed: Jun 15, 2025]

STATEMENT

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 20 June 2025

Benedict Darrel Setiwan
13524057