PSEUDOCODE ETHAN ZHANG

```
FOR each of the region nodes, ADDNODE to graph.
FOR each of the edges between region nodes, ADDEDGE to graph, as directed with weights
INITIALISE lists and sets with all nodes and edges
```

DIJKSTRA'S ALGORITHM

```
FUNCTION dijkstra(starting_node):
    searched_nodes = set.create()
    node_distance = PriorityQueue.create()

    SET weighting attribute of all nodes to infinity
    CHANGE weighting attribute of starting node to 0

    INSERT starting node to node_distance with a priority of 0 as the distance from itself is 0

    LOOP until node_distance priority queue is empty:
        USE FUNCTION dijkstra_algorithm with the next element in node_distance

FUNCTION dijkstra_algorithm(main_node)
    node_dictionary = dictionary.create()
    ADD main_node to searched_nodes

    FOR each outgoing edge from the main_node, ADD the target node as a KEY and edge as VALUE to node_dictionary

    LOOP for each neighbouring node IN node_dictionary:
        edge_weight = weight of corresponding outgoing edge in node_dictionary
```

```
        new_weight = main_node weighting attribute + edge_weight

        IF new_weight < weighting_attribute of neighbouring node:
            SET weighting_attribute of node = new_weight
            INSERT neighbour node to node_distance with a priority of the new_weight
        ELSE:
            INSERT neighbour node to node_distance with a PRIORITY of its current weight attribute

    CONTINUOUSLY remove elements from node_distance until we reach a node that has not been searched
```

DIJKSTRA ALGORITHM EXTENSION FOR SHORTEST PATH

```
FUNCTION reverse_search(node)
    RECURSE THIS FUNCTION UNTIL weight attribute of current node is 0:
        neighbours = list of nodes that are pointed toward the current node
        weighting_PQ = PriorityQueue.create()

        FOR each neighbour node:
            INSERT neighbour node to weighting_PQ with a PRIORITY of the weight attribute of node

        next_node = weighting_PQ.peek()
        with the next element in weighting_pq: PUT it in path_stack THEN RECURSE with the element


FUNCTION path_distance(first_node, last_node):
    path = create list with last_node as element
```

```
    USE FUNCTION dijkstra to find distance to all nodes from first_node
    USE FUNCTION reverse_search from the last_node to find the shortest path

    RETURN list(path,weighting of last_node which is the path distance)
```

BRUTE FORCE PERMUTATIONS

```
FUNCTION list_queue(pokemon_list):
    # permutated = [[A,B,C],[A,C,B],[C,A,B]....]
    USE FUNCTION permutations on pokemon_list and INITIALIZE VARIABLE permuted to equal this
    queue_list = list.create()

    FOR each permutation_list in permuted:
        USE FUNCTION path_divide on permutation_list and APPEND result to queue_list
        #(A,B,C,D) → ([A,B],[B,C],[C,D]) where the round brackets represent a queue

    RETURN queue_list

FUNCTION path_divide(node_permutations):
    two_node_path = queue.create()

    IF there is only one pokemon to find, such that len(node_permutations) = 1:
        ENQUEUE the pokemon into two_node_path

    ELSE:
        FOR every permutation in node_permutations:
            ENQUEUE a list of consecutive pairs of nodes from the permutations to two_node_path
            permutation = (A,B,C,D) → ([A,B],[B,C],[C,D])

    RETURN two_node_path
```

```
FUNCTION possible_shortest_paths(pokemon_list):
    queues = USE FUNCTION list_queue on pokemon_list
        # (A,B,C,D) → ([A,B],[B,C],[C,D])
    final = list.create()
    FOR each pokemon_pair_queue in queues: #[A,B]
        path_weight= list.create()
        REPEAT UNTIL pokemon_pair_queue is empty:
            node_pairs = pokemon_pair_queue.serve()

            IF length(node_pairs) = 1, such that there is only one node to reach the pokemon:
                APPEND an array of size 2 to path_weight: [the one node, a distance of 0]

            ELSE:
                USE FUNCTION shortest_distance to find the path between the pair of nodes # [A,B] → [A…B]
                THEN APPEND the path with corresponding distance to path_weight
        APPEND path_weight to final_list
    RETURN final_list
```

SELECT SHORTEST PATH

```
FUNCTION shortest(path_list):
    all_path = PriorityQueue.create()

    FOR each all_node_path in path_list: # [[A…B],[B…C]]

        home_to_first = USE FUNCTION path_distance to compute the shortest path from "home" node to first node in path
        PREPEND home_to_first to path_list
        path = INITIALISE list with home_to_first path
        length = INITIALISE list with length of home_to_first

        FOR two_node_path ([A...B],length) in all_node_path [([A...B],length),([B...C],length)], EXCLUDING first and last elements:
            GET the path of two_node_path : [A...B]
```

```
        THEN REMOVE first index of the path : [...B]
        THEN APPEND elements of this new path to path: [G,H,I...B]

        INCREMENT length by path length

    IF length(all_node_path) != 1,# to consider for cases of one pokemon:
        GET last path of all_node_path: [B...C]
        THEN APPEND elements of this new path without the first node in the path to path: [G,H,I...C]
        INCREMENT length variable by the length of the last path

    SET VARIABLE end_to_home, to output of path_distance from the last node of all_node_path to "home" node
    GET the path of end_to_home and APPEND all nodes of path EXCEPT the first node, to path
    INCREMENT length by the length of end_to_home path
    INSERT shortest path of permutation to all_path with PRIORITY length of the path

RETURN all_path.GETMIN()
```

GRAPH GENERATE ALGORITHM

```
FUNCTION graph_generate(pokemon_list):
    RANDOMISE pokemon in pokemon_list
    pokemon_edges = list.create()
    pokemon_nodes = list.create()
    pairs = set.create()

    APPEND every item in pokemon_list to pokemon_nodes

    ADD array of consecutive pokemons in pokemon_nodes with a random edge weighting integer to pairs:
         (xth item in pokemon_nodes, x+1th item in pokemon_nodes, random number from 5 to 8)

    ADD array with last pokemon_node and first pokemon_node with random weight ininteger to pairs
```

```
    FOR each pokemon in pokemon_nodes, ADD to pairs an array of size 3 with the pokemon, another random pokemon from pokemon_nodes and a
          random number:
          (xth item in pokemon_nodes, random item from pokemon_nodes,random number from 5 to 8  )

    APPEND each array in pairs to pokemon_edges
    RETURN a list of: (pokemon_node,pokemon_edges)
```

SELECT RANDOM POKEMON FOR EACH REGION

```
region_dictionary = dictionary.create()

FOR each list containing every possible pokemon in each region:
    CREATE a new array with a random slection of pokemon with array size greater than 6 and less than the amount of pokemons in list
    THEN USE the graph_generate function on the new list to return a list containing pokemon_nodes and pokemon_edges
    region_dictionary.ADD(key:region,Value: list of pokemon_nodes and pokemon_edges)
```

CALLABLE FUNCTION TO NAVIGATE BETWEEN GRAPHS

```
FUNCTION graph_generate(region):
    REMOVE all edges and nodes from current graph

    IF region is the clickable button node to return to region:
        ADD nodes and edges of region graph
        SET position/coordinate of each node in region graph
```

```
ELSE:
    everything = list.create()
    RETRIEVE pokemon_nodes and pokemon_edges from region_dictionary.get(region)

    CHANGE pokemon_edges from tuple of adjacent nodes and the edge weighting to edge object
    APPEND pokemon_nodes and edge objects to everything

    ADD ALL nodes and edges in everything list
    ADD clickable button node to return to main graph

IF shortest path between 6 pokemon algorithm has been run:
    search_set = list.create()
    path_traverse = list.create()

    IF region is the clickable button node to return to region:
        IF length of first item/path of region_path_length is 1 AND first node of region_path_length is IN region_starting_node:
            SET COLOR of the one pokemon to catch/"home" node to red

        ELSE:
            for each key/region in find_dictionary:
                append region node to search_set

            animate(region_path_length,search_set)

    ELIF region is IN the keys of find_dictionary:

        IF find_dictionary(KEY: region) returns a list with only one node, and that node is in the set of starting_pokemon:
            set_color of that one node to RED

        ELSE:
            FOR each pokemon in pokemon_list:
                append pokemon as node object to search_set

            FOR each node in the first item/node list in path_dictionary(key: region):
```

```
                    append node as a node object to path_traverse

                using animate function ((0,path_traverse),search_set), show path traversal of path_traverse, marking search_set nodes

FUNCTION region_button(region_node):
    only function if algorithm has finished AND node clicked is in region_set or clickable button to return to main graph
        graph_generate(region_node)

use pynode register_click_listener function to input all node clicks to region_button() and change graph
```

DFS

```
FUNCTION dfs_algorithm(node):
    ADD node to visited
    ADD node to pokemon

    FOR each outgoing neighbour node:
        IF the neighbour NOT in visited
            PUSH NEIGHBOUR TO STACK

    IF stack is not empty:
        recurse dfs_algorithm with next element in stack

FUNCTION dfs(start_node):
    visited = set.create()
    pokemon = set.create()

    stack = stack.create(start_node)
    dfs_algorithm(start_node)

    RETURN pokemon
```

PERFORM DFS ON GRAPH

```
FUNCTION region_search(regions):
    pokemon_dictionary={} #create pokemon dictionary to record locations of each pokemon

    for region in regions:
        graph_generate(region)
        all_nodes=graph.nodes()
        all_nodes.remove(graph.node("RETURN TO REGION"))
        pokemon_dictionary[region]=dfs(random.choice(all_nodes))

    #restore region graph
    graph_generate("RETURN TO REGION")

    return pokemon_dictionary

pokemon_dictionary = region_search(region_list)
```

PATH ANIMATION

```
FUNCTION animate(shortest_distance_node, searching)
    ITERATE through every node in first item/node list of shortest_distance_node EXCEPT the last node:
        edge_pq = priorityqueue.create()

        IF it is the first node:
            colour first node in path of shortest_distance_node blue

        edges = list.create(edges between(current node, next node in path))
        INSERT_WITH_PRIORITY each edge in edges to edge_pq : (priority: edge.weight(),value: edge)
```

```
        shortest = GET lowest weighting edge between the two adjacent nodes with edge_pq.get()
        final_length += weighting of shortest edge

        SHOW traversal animation between the two nodes via shortest edge

        IF the next node in path is in searching and is not the "home" node:
            colour the next node red
        ELSE:
            colour the next node black
```

GENERATE POKEMON TO FIND AND LOCATE

```
findable_pokemon=list.create()
find_dictionary= empty dictionary for locations of 6 pokemon

FOR region node in pokemon_dictionary:
    APPEND to findable_pokemon: the pokemon of the corresponding region from pokemon_dictionary
pokemon_list = randomly select 6 pokemon from findable_pokemon list

FOR each region and for each pokemon to be catched:
    IF the pokemon is present in the region:
        ADD a new entry to dictionary with key:region, value: pokemon
    HOWEVER, if the key already exists, update the region in find_dictionary with a new list with the new pokemon
```

COMPUTE SHORTEST PATH BETWEEN POKEMON IN SUBGRAPHS

```
path_dictionary = dictionary.create()
main_nodes = set.create()
starting_pokemon = set.create()

FOR each region from the keys of find_dictionary:
    region_pokemon = list.create()
    pokemon_id_list = list.create()
    graph_generate(region)

    starting_node = random node from list of available pokemon from pokemon_dictionary(key:region)
    ADD starting node to starting_pokemon set

    FOR each pokemon that needs to caught derived from find_dictionary(key:region):
        APPEND the pokemon to region_pokemon list
        ADD pokemon to main_nodes set

    pokemon_path = shortest(possible_shortest_paths(region_pokemon))

    CONVERT each pokemon in the path of pokemon_path to graph object:
    THEN the nodes to pokemon_id_list

    UPDATE path_dictionary(key:region) = list(pokemon_path, pokemon_id_list)
```

COMPUTE SHORTEST PATH BETWEEN REGIONS OF MAIN GRAPH

```
USE FUNCTION graph_generate to make main graph
```

```
FOR each region from the keys from find_dictionary:
    CONVERT region to node object of
    THEN APPEND node object to exploring_region list
    ALSO ADD node object to main_nodes set

starting_node = a single random converted node object from region_list
region_starting_node = starting_node

USE FUNCTION possible_shortest_paths on exploring_region to compute shortest paths of every combination of the 6 pokemon
THEN USE FUNCTION shortest on the result to sort out the minumum cost path with its corresponding path
SET variable region_path_length =  shortest path

region_path = first item of region_path_length which gets the path list
```

ANIMATE AND PRINT SHORTEST PATH

```
coloured=list.create()
notmain=list.create()
path_edges=list.create()
been=set.create()
final_path=list.create()
final_length=0

IF length(region_path)>1:
    FOR each region in region_path EXCLUDING last region:
        SET COLOUR of starting region to blue
        APPEND current region to final_path
```

```
IF current region is IN main_nodes such that it is a region with a pokemon to catch:
    REMOVE current region from main_nodes to prevent re-entering of pokemon subgraph
    ADD current region to been
    SET COLOUR of current region to red
    HIGHLIGHT current region(colour=Red, size = 3*current size)

    IF current path is not the starting region:
        APPEND current path to coloured

    graph_generate(current region)

    IF the length of path_dictionary(key:current region) is 0 such that the subgraph has only 1 node to be traversed:
        for the node, SET COLOUR to red, INCREASE SIZE to 50 and do highlight animation

    pokemon_nodes= list.create()

    FOR the pokemon in the path_dictionary(key:current region):
        CONVERT pokemon to node
        THEN APPEND node to pokemon_nodes and final_path

    search_set=set.create()
    FOR pokemon in pokemon_list:
        CONVERT pokemon to node
        THEN APPEND node to searched_set

    USE FUNCTION animate to show traversal of pokemon subgraph ([length,pokemon_nodes],search)
    graph_generate(main graph)

    FOR each node in notmain:
        set node colour to black
    FOR each node in coloured:
        set node colour to red


ELIF current node does not contain pokemon to be catched and is not the starting node:
```

```
            APPEND current node to notmain
            SET COLOUR of current node to black

        FOR edge in path_edges:
            set edge colour to black

        APPEND edge joining current node and next node in region_path to path_edges
        do traverse animation along edge joining current node and next node in region_path with COLOUR black
        final_length += weighting of edge joining current node and next node in region_path
    APPEND last node in region_path to final_path

OTHERWISE a case where there is only one region in region_path:
    APPEND the only node in region_path_length to final_path
    ALSO SET COLOUR of node to blue

    IF the only node in region_path is in main_nodes and contains a pokemon to catch:
        SET COLOUR of node to red
        graph_generate(current region node)

        IF length of path_dictionary(key: current region) is 1 or that the shortest path consists of only one node:
            highlight this one pokemon node red

        pokemon_nodes= list.create()
        FOR the pokemon in the path_dictionary(key:current region):
                CONVERT pokemon to node
                THEN APPEND node to pokemon_nodes and final_path

        search_set=set.create()

        FOR pokemon in pokemon_list:
            CONVERT pokemon to node
            THEN APPEND node to searched_set


        USE FUNCTION animate to show traversal of pokemon subgraph ([length,pokemon_nodes],search)
```

```
        graph_generate(main graph)

generated=True
PRINT total cost
PRINT sequence of nodes for shortest path
```