POKEMON CATCH

ETHAN ZHANG

The problem

6 random pokemon are chosen from a list of pokemon that can be caught. The goal is to find the least time path to catch these 6 pokemon. The player will start in a random region which will be their “home”. After catching all 6 pokemon the player needs to return back home; the length of the trip back home must be accounted for. To catch the pokemon, the player must first navigate to certain regions containing some of the 6 pokemon. Once they enter the region, they will start from the location of a random pokemon that will now be another home. They will catch each of the required pokemon within the region in the fastest time, then return back “home”. Similarly, the return trip to the home node will be accounted for.

The modelling of the problem may seem unrealistic to suggest that the starting pokemon whenever a region is entered is random, however, this is actually a realistic feature that reflects the movement of pokemon. It also accounts for entering regions from different directions such that the starting pokemon encountered coming from a north region will be different to the pokemon encountered when entering from a south region. So instead of the player being randomly spawned, it is instead the pokemon that are actually being randomly scattered.

Solution

Algorithm overview

My solution of solving this problem involves taking a modular approach, tackling a key aspect at a time.

1) First we need to find which pokemon can be caught. To do this we can iterate a Depth first search algorithm over each region sub-graph. We can then use a dictionary with keys as required regions and values as sets of available pokemon to record the information.

Keys: Dictionary → List

Where in our problem, the dictionary keys are regions and the outputted list would be a list of these regions.

We can then find every permutation of the regions that need to be traversed. This is shown in detail below where ABC are the regions that need to be reached.

(A,B,C) →[(A, B, C), (A, C, B), (B, A, C), (B, C, A), (C, A, B), (C, B, A)]

One of the methods this can be done is a module itertools which contains a ADT specification “permutations”. The specification is as follows:

Permutations: Iterator x Integer→ List

Where an interator is any ADT that can be iterated such as lists, tuples, dictionaries, and sets. Integer would specify the size of permutations. For our algorithm we would input the integer size of the inputted iterator as we want permutations containing all required regions. The outputted list would be a combination ADT of multiple tuples within the list. Each tuple will contain a permutation.

2) Our next step is to begin the algorithm of finding the shortest path between the six pokemon. We first turn the keys of the dictionary of regions that need to be found into a list. The python list ADT specification to do this is as follows.
3) Next, for each permutation, we partition the list of pokemon to multiple pairs of pokemon.

   In my code I appended each pair of nodes to a queue represented below by the square brackets. The purpose of doing this was for the ease of retrieving items and error prevention. As a queue acts as FIFO and only the end of the queue can be retrieved, I would remove all items from the queue until isempty returned true. This ensured that the path order was retained and makes the queue adt the best option. The queue ADT specifications used were as the following:

Create: → Queue

qsize: Queue → Integer # return the size of the queue

#put each pair of nodes
Put : Queue x element → queue # put an element into queue

#Retrieve each pair of nodes
Get : Queue → Element # retrieve the next element in queue

#Iteratively DeQueue items from queue until it is empty
Empty: queue → Boolean # check if queue is empty

(A,B,C) →[(A,B),(B,C)]

4) For each of the pokemon pairs, we can now use dijkstra's algorithm to first initialise every node distance from the source node. With this information we can use another reverse search algorithm to navigate from the goal node to the starting node, continuously selecting the incoming neighbouring node closest to the starting node. This provides us with the shortest path.

(A,B) →(A…B)

5) We now need to take into consideration that there is a randomised node that the player must start from. This is done by finding the shortest path from the home node to the first pokemon of the permutation, prepending this to the overall path list. We then find the shortest path from the last pokemon in the permutation to the home node, appending this to the overall path list.

[(A…B),(B…C)]→[(Home…A),(A…B),(B…C),(C…Home)]

6) Accumulating the path costs for each of the subpaths we can find the overall path cost.
[(Home…A),(A…B),(B…C),(C…Home)] → [(Home,A….C,Home),COST]
To figure out which ADT specification to use for storing the overall path, i realised i would need a ADT that could handle being appended to in an ordered fashion to preserve the order of the paths. In general, any ordered ADT would work such as a stack, list, array, queue, because we do would not need to access any of the items

However a tuple will not work as it cannot change after being created, thus it would be unsuitable. I ended up deciding to use a python list as it would prove me with the greatest flexibility if I ever ran into an error and needed to access certain elements in the ADT. This would be difficult with queues and stacks as when python prints these ADT's, it only prints an object, not all the items inside. The list ADT's i used in my code were the following.

create: → list
isempty: list → boolean
append: list x element → list
remove:list x element → list
index: list x integer → element

#join the pair node path to overall path
extend: list x list → list

7) We repeat this process for each permutation of pokemon, which provides us with a number of paths that all traverse the pokemon. This generates a great number of possible shortest paths with their associated costs. To decide which ADT to store all these in, I realised that the ultimate goal of this algorithm was to find the least cost path. So I used a priority queue that would use its key operations to naturally sort the entries from least cost to highest cost. Through this method, although not required, would also allow me to find the second shortest path, third shortest path and so on. The last entry in the priorityqueue, however, will not be longest path. This is because there will always be a longer path possible by doing another loop within the graph. This is a similar scenario to the example shown in proof 1.0. However, the highest cost path in the priorityqueue reveals the longest possible permutation of the 6 pokemon where for each sequential pokemon, the shortest path is utilised.

Let us now explore the scenario where we wished to use find information about the nth shortest path of the shortest paths between each node. We now encounter a problem as there are no ADT's that act as a priority queue whilst being able to be accessed like an array. This is an **ADT limitation** that could only be fixed by introducing a new ADT operation that would sort a list or array by a priority integer.

[(Home,A1….C1,Home),COST1] ……..[(Home,A6….C6,Home),COST6]

Creating visual graph

The entire library of pokemon gen 1 is present in the program code.

1) Generate the main region graph with the nodes representing the type of pokemon contained, and the edges representing the time required to travel between the regions.

create: → graph
add_node: graph x element → graph
add_edge: graph x element x element x boolean → graph
add_all: graph x list → graph

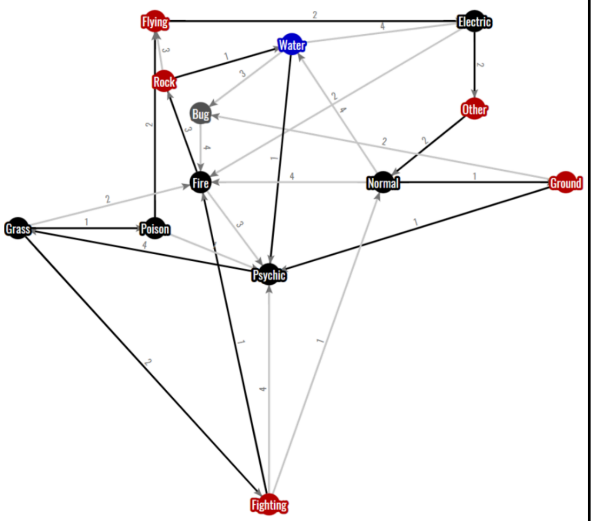
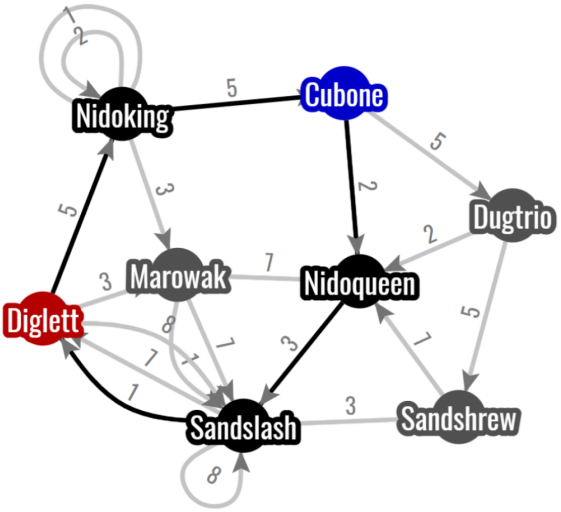
2) For each pokemon type, select a random amount. Storing this in a dictionary with the keys being regions/types and the values being lists of each pokemon type.

#add element to dictionary with associated key
Add: Dictionary x element x element → Dictionary

3) For each main graph region, if a node is clicked, the main graph will be cleared and using the dictionary containing the pokemon of the specific region, it will generate a subgraph containing pokemon.
clear: graph → graph
add_all: graph x list → graph

4) If the button to return to the region is clicked, the pokemon nodes and edges will be removed and the main region added again with the same set positions.
set_position: graph x element x element → graph

Graph Features

MAIN REGION GRAPH	POKEMON SUB-GRAPH
	

Graph traversal

1) Upon running the Pynode file, a number of graphs will be rapidly generated. This is normal and is just the DFS, Dijkstra's Algorithm and Shortest path Algorithm being performed on each of the pokemon sub-graphs and main region graph.

2) Once the algorithm has completed, we will return to the main region graph and a single node will be coloured blue. The blue nodes in my graphs represent the "home" node that the player will start from. The player will also return to this node once all pokemon have been caught.
set_color: graph x element x element → graph

3) The program will print the pokemon that the player is trying to catch, and also the regions that these pokemon are located in.

4) To represent the player's movements between nodes, if a player traverses from one node to another, the edges will be coloured black, whilst a small traversal animation will also happen.

5) If the player lands on a region that does not contain pokemon to catch, the node will simply change from grey to black.

6) If the player lands on a region that contains pokemon to catch, the node will be expanded and turn red. The main graph will then be removed and replaced with the region node that it entered.

7) Entering the region node, the player will start from another "home" node that will be coloured blue. After collecting the required pokemon on the sub-graph, the player will have to return to this blue node to exit back to the main graph.

8) Similar to the main-graph, nodes of interest that contain the required pokemon will turn red when traversed to symbolise the player catching the pokemon. The traversed edge will be black and nodes traversed that do not contain required pokemon will be black..

9) However, if the starting node is also the node containing the pokemon to catch. The node will simply be illuminated red, before the sub-graph is exited and the main graph is recreated.

10) Once all pokemon have been caught, the player returns to the starting blue "home" node located on the main region graph. The program will then print the overall length of the path in minutes and sequence of nodes it took.

To recognize if a region contains the required pokemon, a set is used. The python set ADT specifications that i have used in my algorithm are as follows:

Create: → Set
#When the 6 pokemon and their corresponding regions were randomly selected, **Add** the nodes to a set.
Add: Set x element →Set
#during a path traversal if the set of pokemon and regions **contains** the current node, change the colour of the node to red.
contains: Set x Element→ Boolean
#clear all items in set
Clear: Set →Set

Graph Features

A main directed and connected graph includes 13 region nodes: flying, bug, electric, fighting, fire, grass, ground, normal, poison, psychic, rock, water, other.

Each main region has a subgraph containing 6-17 pokemon of the corresponding region type. The subgraphs have a randomised amount of random pokemon derived from a list of every gen1 pokemon of each type pre-existing within the code. Edges are also randomly generated.

Possible features of my subgraphs with their definitions are shown in the table below.

Feature	Definition	Purpose
Directed edges	Edges have a defined direction associated with them indicating that you can only travel along them in the given direction	Edges in my graph represent the time in minutes required to traverse from one pokemon/region to another. The direction shows that a player can only walk in one direction
Strongly connected	All nodes are connected to another node via an edge. Such that for any two nodes, there exists at least two paths that can be used to traverse from a first node to second node or second node to first node.	The starting nodes are randomised so to ensure every pokemon can be reached from any other pokemon by an edge, the graph had to be strongly connected.
Cyclic graph	The graph contains a cycle	To allow for the graph to be strongly connected.
Weighted	Edges have weights on them indicating size or value	Use of BFS is not possible. The weights indicate the travel time between regions.
Labelled	Nodes are labelled with some attribute relating to them	The nodes in my model represent either pokemon in the subgraphs or regions which are defined by the pokemon type they contain. So they are named either the pokemon name or pokemon type.
Loops	An edge whose target and source is the same node. Or both ends are connected to the same node	My graphs may contain single or multiple loops on same node.
Edges	Two nodes can be connected by an edge which can represent some connection they share with each other. This could be distance, time, cost.	There may be more than one edge between a pair of nodes. These edges may also be of the same direction. The consequence of this is during the traversal animation of edges, the shortest path does not store

		information on the edges, so we must insert each edge from a starting node to the next node in the shortest path to a **priority queue**. We then retrieve the edge with the lowest weighting.

GRAPH ADT'S
outgoing_edges: graph x element → list
target: element x graph→ element
weight: element x graph → element
clear: graph → graph
set_attribute: graph x element x element → graph
get_attribute: graph x element → graph
set_position: graph x element x element → graph
set_color: graph x element x element → graph
set_label: graph x element x element → graph
edges_between: graph x element x element → list
node_id: graph x element → element
graph_node: graph x element → element
predecessor_nodes: graph x element → list
set_value_style: graph x element x element x element → graph
source: graph x element→ element

Finding which pokemon can be caught.

Modelling, Real World Applications, Modifications

To better model this problem as a real world situation, I made the presence of pokemon in some regions not always available for reasons such as being caught by other players or located in unreachable locations such as underground. My implementation of this is to randomise the availability of pokemon per region by selecting a random amount of pokemon from a list. The region graphs on the other hand remain constant, similar to the real world where geographical regions do not move. The random selection of elements from a list is an ADT operation that was imported from the random module, the signature specification would be in the form:

random.sample: list x integer→list

Where "list" is the list of pokemon to select from and "integer" would be the number of random elements. The result would be a list with the specified amount of random pokemon. The availability of this new signature operation will not assist with the algorithmic solution of finding the shortest path between 6 pokemon, however it will improve the modelling of the problem by increasing the real world applicability of the problem, hence the viability of implementing the corresponding algorithmic solution.

Directed edges simulate one way paths and loops symbolise paths that lead to no pokemon but can be retraced back to the node. Edges and edge weightings are also randomised to simulate the changing of location of pokemon and ability to navigate between them. This is done through another extension list ADT from the random module:

shuffle: list → list

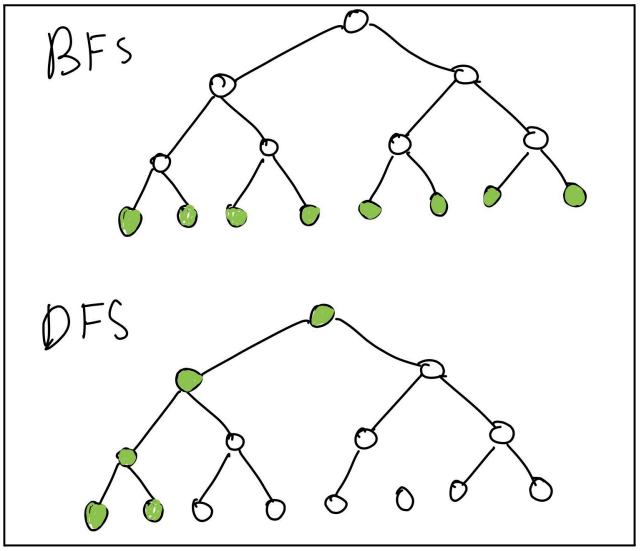
where the new list will contain the same elements as the first, however the order is randomised. This ensures that the final edge connections are not dependent on the order that the pokemon have been pre-placed in the code.

However this only takes a snapshot of the locations of pokemon at any time. To fully integrate this problem for real life purposes, real time dynamic changes to the graph would have to be implemented. Unfortunately this would be beyond the scope of this course, however it would be an interesting concept to explore. A third variable of time could dynamically change the presence of pokemon in a certain location. Instead of nodes representing the location of pokemon and edges being the distance/time, nodes can entail an probabilistic approximation of the likelihood of a pokemon being in an area at a certain time whereas edges can be an estimate of distance. The path that ensures the greatest likelihood of catching every pokemon in the least amount of distance/time could then be explored.

DFS

As a result of randomisation of pokemon and edges, a search algorithm must be utilised to map out the possible pokemon that can be caught by the player. Two search algorithms from the course can be used, depth first search and breadth first search. Comparison of the two algorithms for the basic use of mapping a graph, leads to DFS being the slightly better option. However, although this is the case, for other applications with these algorithms, the result may be different.

Memory: In certain circumstances, dfs will require less memory than bfs. An example is provided below where bfs must store 8 nodes whereas dfs is only storing 5 at one time. Nonetheless this would be extremely unnoticeable for small graphs and where there is lots of available memory.



Ultimately there is very little disparities between the two algorithms when used for their most basic purpose of searching nodes. However I decided to use DFS for the extremely slim advantage. The

function I implemented must use an ADT to store nodes. The DFS algorithm I programed, recursively pushes neighbour nodes to a stack, recursing with the next unexplored element in the stack. This is repeated until the base case is satisfied when the ADT operation "**Empty: LIFOqueue→ Boolean**" returns True.

In the case of DFS, the most optimal ADT would be a stack for its FILO properties. This is because DFS traverses as deep as it can within a graph, until it either reaches a dead end or all surrounding nodes have been marked. The way the algorithm chooses the next node is based on an ADT specification, so it would only be reasonable to use a stack so that the nearest neighbours of any current node the DFS is on are the only nodes it will choose. The imported LIFOqueue ADT specifications are the following:

Create: →LIFOqueue

#Recursive function until stack is empty
Qsize: LIFOqueue →integer

#push unmarked neighbour nodes to stack
Put: LIFOqueue x element → LIFOqueue

#retieve element at top of stack
Get: LIFOqueue → element

#return true if stack is empty
Empty: LIFOqueue → Boolean

Other ADT's such as a queue would not be suitable as it functions on a FIFO basis. This will cause the DFS to continuously retrace its path to scan the neighbours of previous nodes. This would no longer be DFS and depending on the use of DFS would be highly inefficient to continuously backtrack on paths.

To store data on the locations of pokemon, we need an efficient and reliable ADT that has the necessary ADT specifications. The goal of the ADT is to be able to append pokemon to

The way the algorithm works is first we set the weighting attribute of the distance of every node from the source node as infinity. Then logically, we make the source node have a distance of 0 from itself. Starting from the source node, for each of the neighbour nodes that can be reached, we sum the source node weight attribute with the weight of the outgoing edge to the neighbour. If the calculated value is less than the neighbouring node's weight attribute, then update it. Once this process is done on the starting node, mark it in a way such that we do not repeat this process for the same node.

From the list of nodes that do not have a weight of infinity, but not marked, apply this process again to the least weight attribute node. End this recursion algorithm when all nodes have been marked.

Algorithm for shortest distance node/s from a source node and graph generation

Dijkstra vs Bellman Ford vs BFS

Algorithm definitions

Dijkstra's algorithm is a greedy algorithm that finds the shortest length between a source node and another node on the graph.

Bellman ford algorithm
Bellman ford on the other hand is a non-greedy algorithm that finds the shortest length between a source node and every other vertex on the graph.

Breadth first search is another potential algorithm that can be used for finding shortest distance between a source node to another single node on the graph. On the first instance that BFS enters the required node, the number of recursions would be the distance from the starting node.

The task I require the algorithm to do is find the shortest path between two nodes of a weighted graph with no negative edges.

Negative edges

The key differentiating feature of the bellman ford algorithm is its partial ability to be used on graphs with negative edges. However, like dijkstra's algorithm, it too cannot deal with negative cycles otherwise it will become stuck in an infinite loop. And for BFS, the very presence of weights, let alone negative cycles, removes its ability to find the shortest path However, it is the case that a shortest path can never be found for any graph with negative cycles due to its very nature.

This can be proven very easily by contradiction.

Let G be the graph that contains a negative cycle and two nodes S and E. We are proving that there exists no shortest path from S to E.

For the sake of contradiction, let's assume that there exists a shortest path from S to E.

Since G contains a cycle, the graph can be traversed endlessly in a loop where there are no restrictions on the path. Let (S,….,E) be the shortest path from S to E. From the path (S,….,E), let us traverse the negative cycle one more time. As the length of negative cycles are always negative, this new path will have a length less than (S,....,E). However this is a contradiction as (S,....,E) was defined as the shortest path.

Hence, it can be concluded by contradiction that if a graph has a negative cycle, there will never exist a shortest path between any two nodes in that graph.

Proof (1.0)

The problem of negative cycles, however, can be solved by disallowing for intersection paths such as a hamiltonian path.

Algorithm to generate random graph

To increase complexity my graphs have only directed edges and they are randomly generated.

A complexity I added to my graphs was that the initial node when entering any region is always randomised. Therefore, to ensure pokemon can always be caught from any starting node, my graphs must be connected. With an undirected graph, this could have easily been done by creating a set of edges between a small number of nodes, then connecting these subgraphs together. However, to further increase complexity, I decided to use directed edges; the method detailed prior is no longer possible without consideration of edge direction as it is not enough for the graph to be weakly connected.

Thus, an alternative algorithm must be used to generate the random graphs.
From the list of randomly selected pokemon, I iterate through each node, connecting each indexed element with its next adjacent index element with direction. The last item of the list is connected with the first element. This is done with an python list ADT observer operation that retrieves the item at a specified index position.

```
#list is list of pokemon, integer is index position, element is item at index position
Retrieve_index: List x Integer→ Element
```

Doing this operation will make the graph cyclic as visually a circle has been formed. The purpose of doing this is to ensure strong connectedness between every node to every node. A number of random edges with random weights are then formed between each node to increase possible paths between every node to every node.

We will now consider the case if negative weights were introduced to my graph and explore the advantage and coherence of using the bellman ford algorithm with relation to its ability to deal with negative edges over DFS and Dijkstra algorithms.

Bellman Ford negative edges

Given node weights were still randomised, the formation of a negative cycle is not impossible as there will always exist at least one cycle that was created during the production of the random graph, and every edge in this cycle can be negative as edge weights are also randomised. We can now see that the implementation of negative edges in my algorithm may cause negative cycles, and looking back to proof(1.0) a graph with a negative cycle will never have a shortest path. Hence, it can be concluded that bellman Ford's ability to find shortest paths in negative edge graphs will remain redundant even if negative edges were introduced to my graphs. Thus this feature is no longer an advantage to consider for my choice of algorithm. Nonetheless due to real world applications and my knowledge of pokemon,

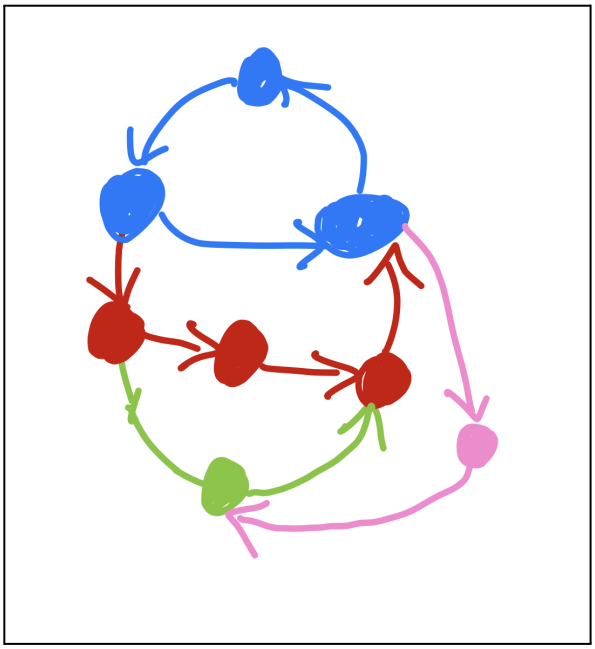
as long as edge weightings follow conventional values such as length and time, they should not be negative in the first place.

Two more different Algorithmic Approaches

We can use the statement that a directed graph is strongly connected if and only if it has an ear decomposition. Therefore, an algorithm motivated by ears of graphs, would be started by first forming a small cycle such as the 3 blue nodes shown in the image below. A recursion function can then incrementally layer nodes on the existing graph until a base case that all nodes have been added is satisfied. Each layered node however, must be directed in a straight line to ensure connectedness. The implementation of this is shown below with the red, green and pink edge and nodes.

A different brute force method that is beyond the scope of the course is to continuously generate random graphs, each graph will be tested for ear decomposition and when the result of one graph returns true, the algorithm will stop and the graph is returned.

Comparing these two algorithms to the one used in the final program, they are all equally effective at forming a strongly connected directed graph.



Back to comparison of algorithms

In terms of efficiency and speed, Modified BFS is definitely the best option as it immediately stops once the required node is reached. This aspect is lacking for dijkstra's where it will always perform a recursion for every node, even if the required node is reached. Bellman ford, as a non-greedy algorithm, also completely evaluates the whole graph, relaxing every edge until either the number of edges-1 or no more changes occur with weights. For my algorithm I require the shortest path between 2 nodes. This is

exactly the function that BFS performs, however unfortunately my graph also contains weights, rendering it unfit for purpose. Thus I must select from dijkstra's or bellman's ford to find the shortest path/distance from a node to another node.

Another argument that can be made against bellman ford is that the algorithm functions by relaxing edges. Thus the efficiency of the algorithm scales with the number of edges. My model of the problem has allowed loops, which will cause redundant calculations. This is in comparison to Dijkstra's algorithm which will ignore loops as the algorithm is finding the distance from one node to every other node. So the distance from the node to itself will never have to been calculated. This makes dijkstra a better option. BFS would also work quite well with loops as it only needs to traverse to every node not edge, however as previously indicated remains unsuitable for the problem.

Ultimately, I chose Dijkstra's for its versatility and efficiency compared to BFS who is restricted to unweighted graphs and bellman ford which requires much larger processing.

So how does dijkstra run?

The way the algorithm works is first we set the weighting attribute of the distance of every node from the source node as infinity. Then logically, we make the source node have a distance of 0 from itself. Starting from the source node, for each of the neighbour nodes that can be reached, we sum the source node weight attribute with the weight of the outgoing edge to the neighbour.

If the calculated value is less than the neighbouring node's weight attribute, then update it. Once this process is done on the starting node, mark it in a way such that we do not repeat this process for the same node. This is done with by storing each starting node in a set.

Add: set x element → set

In another iteration, if we find that the node has been marked. Skip it.

In: element x set → boolean

From the list of nodes that do not have a weight of infinity, but not marked, apply this process again to the least weight attribute node. End this recursion algorithm when all nodes have been marked.

set_weight: graph x element x element → graph
weight: graph x element → element

Implementation of Dijkstra's algorithm required a ADT to store all neighbour paths. We can compare which ADT specification to use.

Stack, Queue, Lists : The key feature of stacks,queues, and lists is that they are ordered in a certain way. Stack: FILO and Queue: FIFO and List is ordered in appended order. Although we do need sorted

items, we need it sorted by weight attribute. The order of these 3 ADT's are dependent on the order that items are put in.

Sets and Dictionary: Both ADT's are unsorted, furthermore items of sets cannot be retrieved, and dictionary values act on a key, value basis which is not coherent in this scenario.

Array: Can be used in conjunction with a sorting algorithm that determines which element of array has the least weighting. However this is not efficient as it requires the use of an extension ADT operation.
PriorityQueue: This ADT would be the most suitable as it contains an operation that retrieves the item with lowest priority. The ADT specifications used are the following:

Create: Iterator → heapq
Where an iterator would be any ADT that can be iterated. This could be an array or list or something else
heappush: heapq x element x integer → heapq
In our problem we can set the weight_attribute as the priority integer in the following heapq ADT specification, and the element would be the node.

After each recursion of searching a node, we can retrieve and pop the next node through:

heappop: PriorityQueue → element

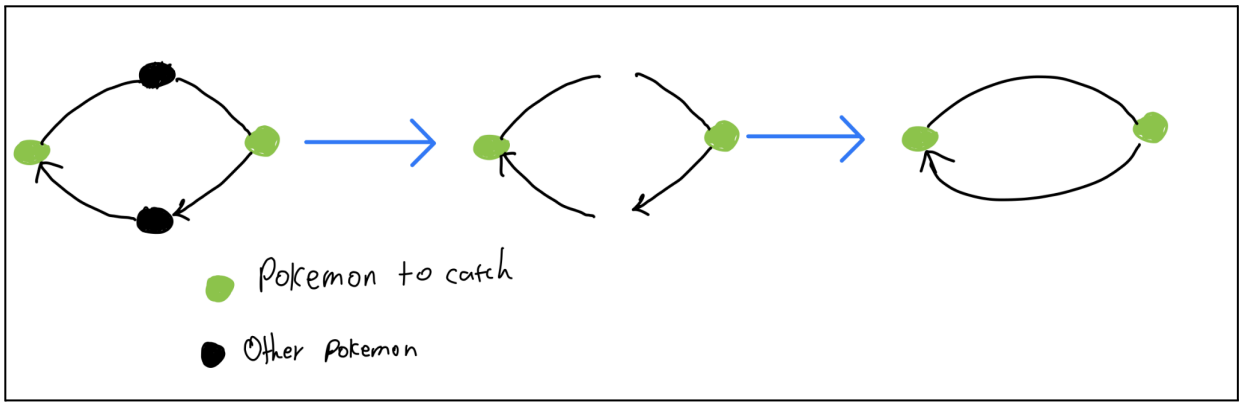
Where the element retrieved is a node.

A possible **limitation** of PriorityQueue seen in other ADT specifications and the vce study design that could be improved is that it cannot peek and pop concurrently. This would reduce function call overheads and make the code more efficient. Fortunately this can be done in python.

Comparison to TSP and Possible Abstraction of Problem with Floyd-Warshall

The modified problem I ended up with was a modified travelling salesman problem(TSP). The reason it was not exactly the TSP was that it was not finding the least cost hamiltonian cycle. Instead, my problem involved finding the least cost cycle from a randomised starting node that traverses to select nodes with the required pokemon. If required, my problem can be abstracted to conform to TSP rules. Advantages of doing so would be a more vast collection of solutions to solve the algorithmic problem. This can be done with the use of a modified floyd warshall algorithm described in the next paragraph.

As a hamiltonian cycle requires visiting every single node exactly once, my graph must be changed to emulate that situation. Currently, only the inputted/randomised set of pokemon need to be visited/caught so it can be assumed that the traversal to any other node would be unimportant. Abstracting the original graph containing all pokemon, unnecessary pokemons can be removed. However the existing edges on these nodes are still of importance as they detail possible paths between 2 important nodes with a redundant middleman. An example of this is shown in the picture below.



Information on the preexisting edges can therefore be used to create a new set of edges. This is a similar process to the floyd warshall algorithm, where the distance of 2 nodes can be derived from a series of distances from middle-man nodes. A 3 dimensional array can be used to find the shortest distance between every node. The result of this array can then be used to develop a new graph with only the nodes with pokemon to catch and edges between them.

The benefit of this abstraction is the reduced edges and nodes which equates to increasing the speed of finding the shortest path between the 6 pokemon with the algorithm I used. However, due to time constraints, this method was not implemented but could be done for semester 2.

Dijkstra Algorithm Extension to find shortest path between 2 nodes

Why i used iterative vs recursive for finding the shortest path

After running a modified Dijkstra's algorithm on the pokemon and region maps, the distance of every node from an initial single node is initialised. We then start at the end node or pokemon that is to be traversed to from a starting node. Using information initialised by Dijkstra's algorithm, the incoming neighbour node with the lowest distance from the starting node will be selected.

As the edges of my graph are directed, only the incoming neighbours/nodes with edges directing away can reach this final node. Recursing this algorithm with the lowest distance node will eventually reach the starting node of distance 0, which defines the base case for this recursion function to stop the algorithm.

The use of an iterative algorithm or recursion would have little difference on performance, efficiency and speed. Although iterative functions are generally more efficient as they do not cause function call and recursion stack overheads, this is generally only for large proportions, so this would be negligible for smaller applications in cases of traversing 13 maps of less than 18 nodes/pokemon. However, if larger graphs needed to be analysed, the use of iterative would be more suitable. As there was little difference between the two types of functions, I chose to do recursion for easier readability and less complexity due to the nature of the problem being highly easy to implement as a recursive function.

Use of ADT's

Finding incoming neighbour node with least distance from starting node

How the algorithm finds the lowest distance neighbour node involves first finding the the incoming neighbour nodes. The pynode graph ADT operation predecessor_nodes will return a list in the following ADT specification.

predecessor_nodes: graph x node → list

The incoming neighbours then need to be stored in an appropriate ADT. The ADT I would use would need an operation that could sort by an integer that is the node's distance from the starting node. The ADT I chose to use is a priority queue; the ADT operation “getmin” will ensure I always retrieve the lowest distance node. The pseudocode will contain the sequence of operations, however the sequence of ADT specifications are detailed below.

#where iterator is any ADT that can be iterated through such as list
heapify: Iterator→ heapq
#Insert an entry with the Element: node, Integer: node distance from starting node
heappush: heapq x Element x Integer → heapq
#Get the node with lowest distance from starting node
Heappop : heapq→ Element

Although another ADT such as a list could be used alongside a sorting algorithm or custom specification, the inbuilt ADT specifications of a priority queue is the most optimal. Any other ADT in the algorithmics course will be unsuitable as detailed below.

Array: The required initialisation size of the array would be unknown, a workaround would be that a different algorithm is utilised to find this information, so it is inefficient. Furthermore, the key specifications of an array is the indexing which would be unnecessary for this application.

Queue: A queue is ordered and acts on a first in first out basis which would be unsuitable as a different order is required.

Dictionary or set: A sorting algorithm would be required to determine the lowest distance node from the starting node. This makes a dictionary ADT inefficient. Furthermore, the set is unordered so it would not be able to be sorted.

Custom ADT operation: A workaround is retrieving every item, storing each element to an ordered array with an corresponding array of distances. Then, getting the lowest number in the second array with a number analysis algorithm, using this index to get the corresponding shortest distance node from the first array. However this uses the assumption that the amount of neighbour nodes is an available ADT observer operation. The signature operations used for this are shown below

Custom ADT operation

#create two arrays with size: length of neighbours

Create: Integer→ Array

#for each incoming neighbour node, store the node into the next index position of an array and its corresponding distance from the starting node into the second array at the same index position.

set_value: Array x Integer x Element→ Array

#Use a custom ADT operation to find the index of the smallest number in the array

Smallest: Array → Integer

#Retrieve the node at the returned indexed position

Get: Array x Integer → Element

Storage of node distances from starting node

Another point to be explored is the storage of node distances from the starting node. This was done through the use of an inbuilt ADT specification of pynode **set_attribute**. In my algorithm the attribute used was “weighting”. I also refer to this attribute within my pseudo code and report as the weighting attribute.

#The first element is a node, the second element is usually a string that sets the attribute type, in the case of my algorithm, that would be the integer distance from the starting node. This returns a updated graph with attribute of the node changed.

Set_attribute: Element x element x Element → Graph

#The first element is a Node with the associated graph. The second element to input is the attribute type. The element associated with the specified attribute of the node is returned.

Attribute: Element x string x Graph → element

If this was not available, a dictionary could be used with keys: nodes, value: integer distance from starting node.

Add:Dictionary x Element x Element → Dictionary

Why i used a stack to store the shortest path

The storage of the shortest paths will also require an ADT. The implication of finding the shortest path from the final node is the path will be reversed. Consequently, a stack would be the most suitable ADT to store the sequence of nodes as stacks run on last in first out, so popping elements of the stack will naturally traverse a path from the starting to end node.

Other ADT's that are ordered such as a list or queue could also be used. A queue would have to be reversed by creating two queues. A list would have to have values prepended, then have its elements observed and removed from the Head. Any unordered ADT by itself cannot be used as a path is a sequence of connected nodes. An array would also be unsuitable as the length of the path is unknown. The sequence of ADT operations for a list is shown below whilst the sequence of ADT operations for the stack are shown in the pseudo code as it is my preferred choice of ADT.

```
path_list=list.create()
Until the starting node is reached:
        path_list.prepend(current_node)
#To retrieve path
Until path_list isempty:
        next_node=path_list.first()
        path_list.remove_first()
#where the sequence of next_node's is the path
```

Both ADT's function similarly however the function and basic operations of the stack ADT made it the optimal choice for its FILO properties that logically solve the problem of a reversed path.

Discussion of greedy algorithm and possible improvements

As it always chooses the immediate lowest distance node it is greedy, however in the case of this algorithm, the shortest path is still guaranteed due to the nature of values initialised by Dijkstras. What the algorithm fails to achieve is determining whether the path selected is unique; when there are multiple neighbour nodes with the same minimal distance from the starting node, a priority queue will function as a normal queue with the first in first out basis, neglecting other nodes. Although all minimum paths have the same distance, the importance of finding all solutions is undermined by the possibility that there may exist a different route that could simultaneously pick up a different pokemon. Ultimately this may lead to the shortest overall path to catch the 6 pokemon being able to be found faster. This

could be a potential improvement that due to time constraints and complexity was difficult to implement as it would have required an additional iterative or recursion algorithm that must draw and relate to the overall path to find the optimal solution. It may also have hindered the efficiency of the overall algorithm due to the added combinations of paths to consider. Where this would be useful would be instances where the number of pokemon that need to be caught is overwhelming large, so finding the optimal path will be near impossible and dynamic printing of the current shortest path found would be required. Therefore the effectiveness may be a good tradeoff for efficiency due to the impossibility of considering every permutation.

Other Overall Algorithms

Some of the alternative methods of solving this problem that will be explored in this section:

- Prim's algorithm
- Brute force/Dijkstra and Search algorithm
- Nearest Neighbour
- Nearest Neighbour and Dijkstra combination algorithm
- Prim and Floyd Warshall combination algorithm

Recursively finding the next closest pokemon

If my graph was indirectly it would be efficient

However as my graph is indirectly a single bad choice can cause an extreme increase to shortest path

Difficult to model travelling salesman as it must return back to starting node, this approach would be better for being able to end anywhere

Prims algorithm

Prims algorithm is commonly used for finding the minimum spanning tree of a graph. It involves starting from any node, then continuously traversing along the lowest weight edge of any neighbouring nodes of nodes it has traversed. Ensuring that the target node of the edge has not been traversed, otherwise this will cause a cycle in the MST. This is not allowed as the definition of a tree is a connected graph containing no cycles.

Now we can consider how suitable it would be to apply Prim's algorithm to our problem.

Prim's algorithm tries to find the minimum spanning tree to every node on the graph. However for our problem there are only 6 pokemon nodes that need to be reached. Furthermore, every region sub-graph and the main graph has a minimum of 7 pokemon. Prim's algorithm therefore will almost never be a reasonable and coherent solution due to it attempting to traverse to every node.

Combination Algorithm of Floyd Warshall and Prim's

However, the unreliability can be improved through abstraction of our problem. Using the abstraction method previously mentioned, we can use a combination of Floyd Worshall and Prim's algorithm to solve this issue. The graph will become simplified to only contain nodes of required pokemon that we can solve before slowly re-adding nodes. This new algorithm that we have developed however, is still

not optimal for problems which only require the path between 6 pokemon. This is because for any graph, prim's algorithm will never guarantee the shortest path as it is simply an algorithm for MSP. A MSP means that there are no loops which may be a restriction that causes a less efficient route. Also, tree lengths are not calculated by the cost to traverse through the tree, but instead just the total of every edge weight. So the shortest path using a MSP will require the player to backtrack multiple times out of the leaves of the tree. However if we generalise our problem to finding the shortest path between the pokemon with no home node for a large number of pokemon. This solution becomes viable as compared to brute force methods and other approximation algorithms that require high computation time due to the high number of iterations/recursion, Prim's algorithm will have a maximum number of recursions as the number of nodes. On the other hand brute force will increase by a factorial rate and approximation methods will require exponentially larger amounts of trials to achieve a good solution due to a greater amount of randomness.

We now discuss another problem. Prim's algorithm fails when directed edges are introduced. No other listed algorithm fails with directed edges. Prim's failure can be easily shown by proof by counterexample.

Through the following proof we can **demonstrate** why prim's algorithm is not the preferred algorithm.

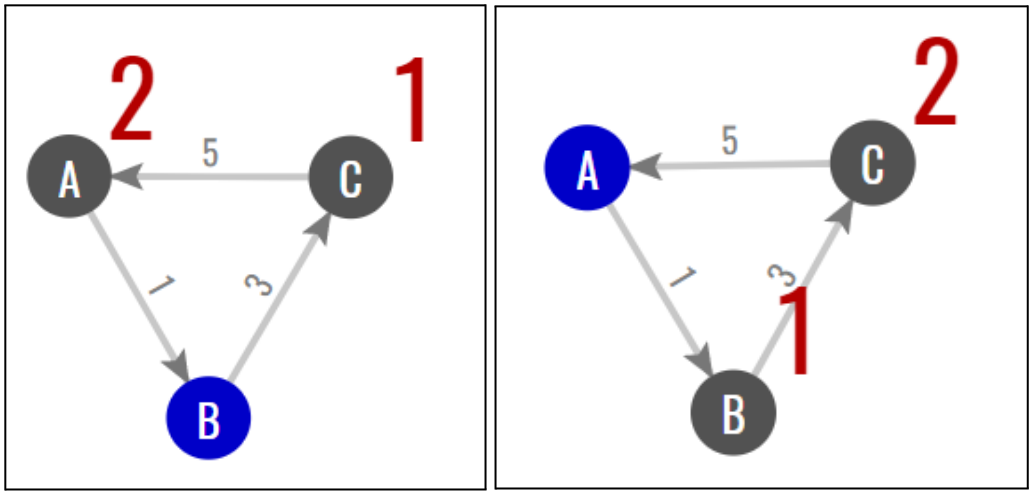
Let us disprove the conjecture: **For any weighted graph G, Prim's algorithm will find the minimum spanning tree from any starting node.**

On the top is Prim's algorithm coded with pynode with the starting node randomised. On the bottom, we have run Prim's algorithm with two different starting nodes A and B. The node letters represent names for the nodes. The node labels represent the order of the MSP. The blue node represent the starting node. The edge weights represent a value between the nodes. If the conjecture is true, both MSP's with a starting node on either A or B will result in the same minimum cost tree. However, we can see below that this is untrue as the left graph has a tree length of 8 and the right graph 4.

```
import random
import heapq
a=Node("A")
b=Node("B")
c=Node("C")
nodes=[a,b,c]

ab=Edge(a,b,weight=1,directed=True)
bc=Edge(b,c,weight=3,directed=True)
ca=Edge(c,a,weight=5,directed=True)

graph.add_all([a,b,c,ab,bc,ca])

start=random.choice(nodes)
start.set_color(color=Color.BLUE)

neighbours = []
heapq.heapify(neighbours)
n=0
```

```
def prims(node):
    global n
    if n<2:
        n+=1
        for edge in
node.outgoing_edges():

heapq.heappush(neighbours,[edge.wei
ght(),edge])

        closest_neighbour =
heapq.heappop(neighbours)[1].target
()


closest_neighbour.set_label(n)

closest_neighbour.set_label_style(s
ize=30, color=Color.RED,
outline=None)
```

```
            pause(1000)
            prims(closest_neighbour)
prims(start)
```



Therefore we can conclude by proof by counterexample of the two graphs shown above that Prim's algorithm will not always generate the MSP from any weighted graph from any starting node.

Proof (2.0)

Thus, the fitness for purpose of prim's would be extremely poor due to my modelled problem involving directed edges.

Furthermore, our problem involves returning back the home node which makes prims entirely unsuitable as this is not accounted for due to it being greedy. An example of this is if there are multiple minimum spanning trees. As Prim's is greedy, it will generally have a hard time finding all MSP's. For the two MSP's one may require a less distance to return back to the home node, making Prim's unreliable. We can draw comparisons with another proposed solution that suffers from the same issue: nearest neighbour.

Nearest neighbour and Modified Nearest Neighbour Algorithms

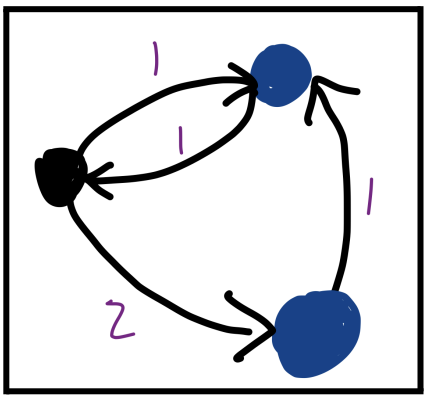
The nearest neighbour algorithm involves starting at a node, finding the closest neighbour and continuously traversing until all nodes have been seen. To have this algorithm be even realistically suitable, we would need to use similar abstraction methods to the modified Prim's algorithm. Using a modified Floyd warshall algorithm that was detailed in an earlier part of this report, the nearest neighbour algorithm can now only traverse to important nodes. It can now return to the original complex graph and find the shortest paths between each sequence of nodes based on the MSP of the simplified graph.

However, as both algorithms are greedy, they do not factor in the necessity to return back to the home node after traversing all required pokemon nodes. This is in contrast with non-greedy algorithms such as brute force and dynamic output approximation algorithms.

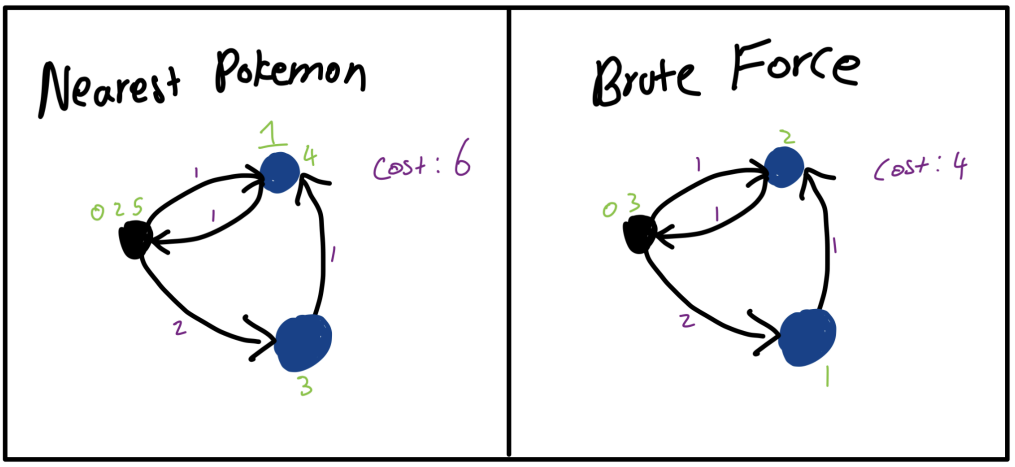
However, nearest neighbour would be more suitable to my model than Prim's as it allows for direction in graphs. Furthermore, construction of the solution to the nearest neighbour will be easier as we can adapt this algorithm to fit with unimportant nodes. Using dijkstra's or an alternative shortest distance algorithm we can find the closest pokemon that needs to be caught. This new algorithm will traverse to the closest pokemon and repeatedly find the closest pokemon until they have all been found. At the very end of the algorithm is where the problem of returning back home becomes present. As the algorithm was greedy it could very well provide a path that is not short at all as it never considered going back.

Using the graph below, we can also demonstrate that the punishment of greediness using a nearest pokemon algorithm is made even more apparent with directed edges:

Let the below image be the graph. Where the black node is the home node and the blue nodes are nodes containing pokemon. The purple edge weightings represent the length between nodes.



The side by side comparison below shows the solution paths outputted by nearest pokemon and brute force algorithms. The green text above nodes represent the order of traversal.



Prim's algorithm's failure to find the MSP above is a result of its greediness. The reason it is greedy is because it always chooses the closest neighbouring node, without considering the possibility that any

other choice will benefit in the long term. It is quite clear how its greediness will disadvantage the algorithm when there is direction for edges and the requirement the return home. This is in contrast to using a algorithm that simulates ant colonies which only decides the path after the recursive function has run a number of times.

Modified ant colony optimization algorithm(ACO):

From a skeletal graph of the pokemon graphs that contains the nodes values but not edge weights, we can initialise every edge weight to be the same and the sum of all edge weights to be 1. We can now deposit virtual ants from the source node. These ants then randomly navigate through the graph, their paths probabilistically influenced by the weighting of the edges. Once an ant has successfully fulfilled the criteria of finding the 6 pokemon and returning, we can analyse the path the ant took and increase the edge weights of the path it took, whilst reducing other edges. Ants who fulfil the criterion faster will have their path edge weights increased more. We can iterate this process as many times as we want, which allows high malleability and adaptability to the circumstances. Such is if a solution is required immediately, less trials can be used. This is a feature not present in any of the other algorithms. Once we decide to stop the trials, we can analyse the path edges, and from the starting node, recursively traverse the highest weight outgoing edges, storing the nodes it enters, before the algorithm loops back.

We can see that ACO only evaluates the shortest path between the 6 pokemon only once all calculations have been done. This is similar to the brute force algorithm. The brute force algorithm is done by considering every possible permutation of the 6 pokemon, calculating the cost of each of the paths before returning the most optimal path. We can then see that ACO and brute force produce non-greedy results. In most cases, non-greedy algorithms produce more optimal results which is especially true when these two algorithms are compared to Prim's and Nearest Neighbour/nearest pokemon. Prim

However, it should be noted that despite ACO being inherently non-greedy, it still only bases the shortest paths on the ant trials, which can have contrastingly different reliabilities depending on the number of trials. More ant trials can always be done that will make the result more optimal. This makes the algorithm require an infinite number of trials to find the best solution, however generally the shortest path will converge after some time. In terms of efficiency, for large scale graphs, ACO would have the most optimal solutions that can be calculated in a realistic time as it uses a dynamic algorithm, relying on previous trials to orient itself toward the best path. This is a stark contrast to brute force methods which, although will guarantee the best answer, take a very long time when the number of nodes increases. Prim's and neighbouring pokemon although could compute in a realistic amount of time, their solutions would be abysmal compared to ACO due to their unsuitability toward the problem.

Ultimately the answer to which algorithm is the best is convoluted, however we can consider the approaches to the brute force method that may make it more viable.

Brute force algorithm

The naive brute force algorithm would randomly create cycles from the starting node and ending in the starting node in hopes of finding a path to the 6 pokemon. The algorithm will continue to run for however long, maintaining a priority queue to keep track of the least cost path. Although technically, eventually the shortest path will be found, it will take an unreasonably long time.

Through functional abstraction, a better approach would be to focus on the pokemon that need to be caught. One method would be to compute the shortest path for every permutation of the 6 pokemon. We can then select the shortest path from these permutations.

permutations: list x element →list

#where the element is the size of each of the permutations and the output list is a list of tuples of permutations.

The greatest advantage that this algorithm has with any of the others mentioned is that it will always guarantee the lowest cost path. However it comes with a caveat in that it becomes increasingly difficult to use for higher amounts of nodes. For just 6 pokemon there are 6!=720 permutations. However, I factored this in when choosing my algorithm and decided it would be a viable tradeoff for the guarantee of the best solution due to the small size. Demonstrating the viability of sacrificing the speed for accuracy, my code can run for a maximum of 120 seconds on my laptop. This is a worse case scenario where each of the 6 pokemon are each located in a different region, forcing the algorithm to calculate the maximum number of permutations. If even one pokemon shares the same region as another, we can reduce the number of iterations to 5! = 120 which usually only takes 10 seconds to compute. Reflecting the real life circumstances of this problem, I viewed this as reasonable, for my model has edge weightings of at least 1, so if through brute force methods, I manage to produce a lower cost path than another algorithm by more than one, it already has become more efficient.

However, similar to Prim's, the introduction of directed edges disallows a possible improvement for the problem. If we consider the case that the model had undirected edges, we can functionally abstract the brute force algorithm. As we must consider every permutation, not combination, order matters. From this we can consider the two permutations of 3 nodes (A,B,C) → (A,B,C),(C,B,A). The two permutations i have shown are in fact just the reverse of each other. Now, if the edges were undirected, the paths could similarly be reversed. However, as my problem contains direction, and not every pair of nodes share 2 edges, the path would be different. As the paths are simply the reverse of each other, the cost of the paths would also be the same. From this knowledge we could have then simply only considered half the permutations had the edges been undirected from 720 to 360. However, it is through such considerations that I decided to increase complexity to explore the viability of directed graphs.

As order matters and the size of the permutations are the same, the adt specification I chose to store each permutation was an array of size: number of pokemon to catch. This was due to the specifications of an array being superior to others, with its indexing abilities that can be iterated through for finding shortest paths, ordering, and set size which reduces memory usage.

Tuples in queues were then used to contain the node pairs. The python ADT Tuples, function similarly to lists and arrays however they are immutable, making them the ideal choice for data that will never be changed after initialisation. Another benefit is human error prevention as the original state of the tuple will always be kept, making accidental modifications not possible. Trying to modify a tuple is not part of its ADT specification so, it will result in an error that can be easily identified unlike other ADT's which would not return any errors and continue with the program, this also helps with debugging. The main ADT specifications i used were:

Create: ELEMENTS→ Tuple

Where the ELEMENTS represent a sequence of elements that the tuple is initialized with and cannot be changed.

In my algorithm this would be a pair of nodes.

Retrieve_Index: Tuple x Integer → Element

Where integer is the index position from which to return an element

As python indexing starts from 0, In my algorithm, integer = 0 would represent the starting node of a path. Integer =1 would represent the ending node of a path. Returning these two values, I can algorithmically construct the shortest path

For each permutation (A,B,C), we partition this into tuples of 2

(A,B,C) →[(A,B),(B,C)]

I described this process at the start of this report, however we can now consider a limitation and possible improvement from the queue ADT to store the tuples of nodes. We can recall that the sequence of node tuples are just the desired order of navigating to the pokemon. To truly find the shortest path in terms of all pokemon, we must find the shortest path between the pair of nodes. Ultimately, the efficiency of the algorithm suffers most from this part as it requires 2 algorithms to be run per tuple. So if we could use functional abstraction to reduce the amount of tuples it would need to iterate through, our algorithm could be improved to be more suitable for time-stressed situations and be a more viable option for larger graphs.

We can realise that each queue of tuples simply represents a single permutation. For greater amounts of pokemon that need to be found, the number of permutations would exponentially scale. We can first realise that the same tuple pairs may appear within multiple queues. This is obvious from the example shown below:

(A,B,C,D) →[(A,B),(B,C)....]

(A,B,D,C) →[(A,B),(B,D)....]

We see the node pair (A,B) repeated. Typically in our original algorithm we would separately compute the shortest path between (A,B) in both instances. Clearly there is computational effort being wasted that could be used on other tasks. However by reusing the solution previously found, we can make our algorithm dynamic. How we can implement this is through a dictionary that stores the solutions to each of these tuple pairs. The python ADT specification is as follows:

Create: → Dictionary
#Check if a key exists in dictionary
HasKey: → Dictionary x element → Boolean

#add element to dictionary with associated key
Add: Dictionary x element x element → Dictionary
#update the value of a inputted key with another element
Update: Dictionary x element x element → Dictionary
#retrieve the value from a inputted key
Get: dictionary x element → element
#retrieve a list of dictionary keys
Keys: dictionary →list

So whenever a new tuple needs to be calculated, it can be checked against the keys of the dictionary using the ADT specification **HasKey.** If this returns true, we know that we have previously found the shortest path between the tuple. We can **get** the corresponding path, completely removing the need to recompute the shortest path. This method of memoization will greatly improve the speed at which the algorithm will find the solution, making it even more viable for small to medium size problems.

Limitations of interface

A small issue I found was that sometimes when graphs were generated, it formed in a line, making it difficult to visualise. Despite efforts to solve this problem such as slowing down the generation of graphs with the inbuilt function “Pause”, the problem persisted. It seemed the problem occurred when multiple graphs were cleared and created. Eventually I deemed the malfunction as acceptable and would be at most an inconvenience. I found that to compromise, the user would either have to manually separate the graph by click and dragging or i must set positions for the nodes. I opted to not set positions due to the nature of my model which uses randomised graphs. This causes ambiguity with edges, making it extremely difficult to find the optimal placement of nodes to cause the least overlap of edges. However, this could be an area of improvement that due time restrictions was not able to be done. The general implementation of a Best Node Placement algorithm could first retrieve a 2 dimensional dictionary of adjacent nodes using the pynode ADT operation

A small issue I found was that sometimes when graphs were generated, it formed in a line, making it difficult to visualise. Despite efforts to solve this problem such as slowing down the generation of graphs with the inbuilt function “Pause”, the problem persisted. It seemed the problem occurred when multiple graphs were cleared and created. Eventually I deemed the malfunction as acceptable and would be at most an inconvenience. I found that to compromise, the user would either have to manually separate the graph by click and dragging or i must set positions for the nodes. I opted to not set positions due to the nature of my model which uses randomised graphs. This causes ambiguity with edges, making it extremely difficult to find the optimal placement of nodes to cause the least overlap of edges. However, this could be an area of improvement that due time restrictions was not able to be done. The general implementation of a Best Node Placement algorithm could first retrieve a 2 dimensional dictionary of adjacent nodes using the pynode ADT operation

Adjacency_Matrix: Graph → Dictionary

The nodes can then first be placed in a 2D grid

Nodes with the most edges can then be placed toward the middle.

Then the algorithm can move each node so that there is 1 less edge intersection. This process of relaxing the nodes can be recursed a number of times until the number of edge intersections becomes constant.

Although the modelling of the pokemon graph is impacted, the algorithm remains unaffected, and simply making the output interface full screen, then manually moving the nodes slightly will allow Pynode to naturally set the positions of the nodes.

A different solution for this problem for situations where the importance of the model is needed would be to use an alternative language to pynode. Mathematica would be a viable option as it contains extremely similar ADT specifications for graph theory that can be used to for calculations and implementation of algorithms. A Custom ADT or Pygame with collision ADT specifications would also make the proposed algorithm above viable.

Collision: object x object → Boolean