VSV VCE Algorithmics SAT U4 AOS 1

Previously, a set of algorithms was developed to aid the Victorian government's response to a widespread infestation of Pangobats, which are known to spread itchy Nose Syndrome (INS). The two algorithms designed for this purpose were:

1. Single Source Shortest Path Algorithm: To route the response force from Bendigo to the target site in the minimum possible time.
2. Travelling Salesman Problem (TSP) Algorithm: To route the medical team, visiting all towns within a specified radius.

This report analyses these algorithms in terms of their time and space complexities and discusses the implications of their time complexities on real-world applications.

The algorithm for finding a route for the medical team is colloquially known as a "Travelling Salesman Problem" (TSP). The TSP is a classic problem in the field of computer science and operations research. The problem is defined as follows: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? The TSP is classed as a NP-hard problem, meaning that no polynomial-time algorithm is known to solve it. The TSP is a well-studied problem in the field of computer science and operations research, and many algorithms have been developed to solve it. The most common algorithms for solving the TSP are based on dynamic programming, branch and bound, and genetic algorithms. The algorithm developed in this case was a combination of a Breadth first search, to find all the relevant nodes in the provided radius, followed by Floyd-Warshall algorithm to generate a matrix of the shortest paths between all nodes, concluded with a brute force algorithm to find the shortest route that covers all nodes.

To find the nodes in the radius a Breadth First Search was taken. The function initialises a dictionary for visited nodes and a queue for nodes to be processed. Then adding the origin node, start, to both the queue and the visited dictionary. All four of these initialisation steps are completed in constant time, O(1). The function then proceeds to execute through the main while loop for as long as there are nodes in queue. The function design means that the loop iterates through every node in the graph, providing no opportunity for the loop to exit early, leading to *O(V)* operations, where *|V|* is the number of vertices in the graph. For each node dequeued, the algorithm iterates over its neighbours, regardless of if the node has already been visited. This results in *O(2E)* operations where E is the number of edges in the graph, this is because each edge has two nodes connected to either side, which means each edge is checked at most twice by the algorithm. When each neighbour is searched, the BFS function computes the haversine distance, checks if it exceeds the radius, and updates the visited dictionary and queue if necessary. These operations all complete in *O(1)* operations. The overall sum of time complexities for this algorithm is *O(4+V(2E+5)),* which can be reduced to the asymptotic notation, *O(|V|*|E|)*.

Breadth-First Search (BFS), with its time complexity of *O(V+E),* is well-suited for various real-world applications. In the context of the Pangobat infestation response, BFS was used to identify all towns within a specified radius from a central point. This efficient traversal ensures that all relevant nodes are found quickly, which is crucial. The required number of operations only increases linearly with the number of nodes and edges meaning that the algorithm is also very scalable if the Victorian government were to create more roads or establish new towns.

Once the relevant nodes in the radius are determined, the program executes a Floyd Warshal algorithm on the input node list, returning an *n*n* matrix of distances. This function initialises using three constant time declarations, *O(1)* and two quadratic time default value assignments *O(n^2)*, getting the number of nodes inputted, n, creating a dist array of size *n*n* with the default value of infinity and creating a previous matrix of size *n*n* with the default value of null. To set these default values, the program iterates over each row and column in each matrix setting the value, this means that the arrays are iterated over a total of *O(n^2)* times. The algorithm then iterates over each row and column again, assigning the adjacency matrix values.

The function that gets the edge time attribute, get_time, works by iterating over all nodes in the graph, $|V|$, to find the relevant edge between the two given nodes. This is completed using one loop. Due to this function call within a double nested loop, the combined time complexity of this loop is *O(n^3)* as it expands out to be a triple nested loop. Floyd-Warshall then executes the main loop, which is a triple nested loop, iterating over each nn item in the nodes list. The time complexity of this loop is also *O(n^3)*, making the overall worst case time complexity equation for the function *O(3n^3+3n^2+3)*, or in simplified asymptotic notation, *O(n^3)*.

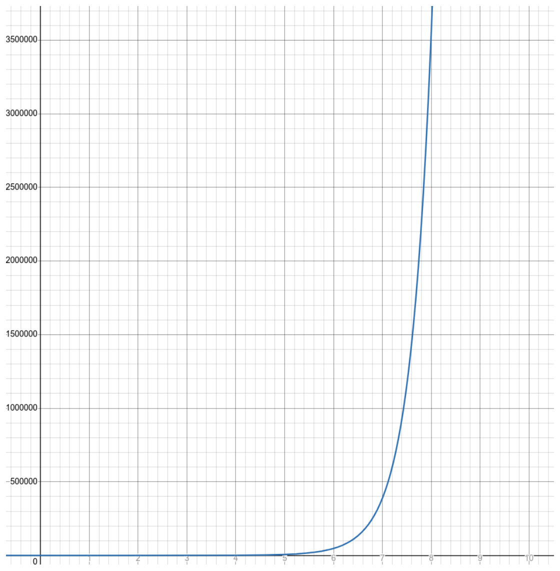
Because of the cubic nature of the Floyd Warshall algorithm, the number of operations required for any given set of nodes explands very quickly, although the dynamic programming features help. For graphs with a large number of vertices, the *O(V^3)* complexity can become impractical. For example, if a graph has 1,000 vertices, the algorithm might require on the order of 1 billion operations, which can be computationally expensive and time-consuming. This would not be ideal for a response to an outbreak of a pest or infection as it would lead to a large risk of further outbreaks and spread. Due to this it is typically only used on smaller graphs and where performance is not of concern.

The algorithm requires *O(V^2)* space to store the distance matrix, which can also be significant for large graphs. This space requirement can lead to memory constraints in systems with limited RAM. Although for this application it would not be relevant as you would expect that a government would have systems with 256gb to upwards of 3TB of memory.

Overall, while Floyd-Warshall is powerful for its ability to handle all-pairs shortest path problems and works well for small to medium-sized graphs, its cubic time complexity means that for large-scale problems or real-time applications, alternative algorithms or optimizations are often necessary.

These two functions are the building blocks primarily used by the TSP function. As previously stated TSP is an NP-Hard problem. Meaning that it can be checked in polynomial time but there is no solution that has been proven to exist in polynomial time. The implementation chosen simply brute forces all the possible permutations of paths in the graph to find the shortest path connecting all of the given nodes.

To find the Time complexity of the TSP algorithm, we start with the BFS operation. This above has been established to take _O(|V|+|E|)_ operations to complete, where V is the number of nodes and E is the number of edges. Floyd warshal is then executed on the return value of BFS, once again, as discussed above the Floyd Warshall implementation takes _O(n^3)_ operations to complete, where N is the number of nodes in the list passed to it — the return value of BFS. The TSP algorithm then continues to make four variables to hold relevant parameters, each taking _O(1)_, constant time, to create. The main loop is then executed, iterating over all permutations of numbers 0-n, This means that the outer loop always iterates _O(n!)_, where n is the number of nodes returned by BFS. This loop then has a nested for loop within it that uses the distances matrix returned by Floyd Warshall's algorithm to compute the travel time of the current path permutation and update the shortest path if the travel time is lower than the previous minimum. This inner loop iterates from 0 through n, therefore taking linear time to execute. Taking into consideration all constant time operations within this inner loop, such as if statements and creating and reading from variables, we can conclude that the inner for loop takes _O(11n)_ operations to complete. From these sub-analyses it can be observed that the overall algorithm takes _O(V+E+n^3+11n*n!)_ operations to complete, or _O(V+E+n!)_ in asymptotic form, where V is the total number of nodes in the graph, E is the total number of edges in the graph and n is the number of nodes returned by the BFS function.



The TSP algorithm's NP-hard nature and factorial time complexity, _O(n!)_, have significant implications for its real-world application. While the algorithm is theoretically sound and guarantees finding the optimal solution, its practicality diminishes rapidly as the number of nodes nn increases. For example, with a relatively small number of towns (e.g., 2-8), the algorithm can compute the shortest route within a reasonable timeframe, less than 1 million operations . However, as n grows, the computational resources and time required increase exponentially, making it infeasible for even small networks and radiuses that only include 10-20 towns. In real-world scenarios where the number of nodes can be substantial, heuristic or approximation algorithms, such as the nearest neighbour or genetic algorithms, are often employed to provide near-optimal solutions within acceptable timeframes.

Even assuming that the government running the TSP algorithm has the very best supercomputers, it may take days to weeks to compute the most optimal path for large graphs. This means that it is not suitable for nearly any time critical circumstances. Even in this case where there are only 62 nodes it could still take upwards of _3.14*10^85_ operations to decide on the optimal path if all nodes are required to be travelled.

The response team required an algorithm to route directly from Bendigo to a given target site in the shortest possible time. Algorithms that fit this task are known as Single Source Shortest Path algorithms. These algorithms belong to the P class of algorithms, meaning that they have been proven to be both checkable and solvable in polynomial time. The algorithm of choice in this case was a derivative of Dijkstra's algorithm with a priority queue optimisation.

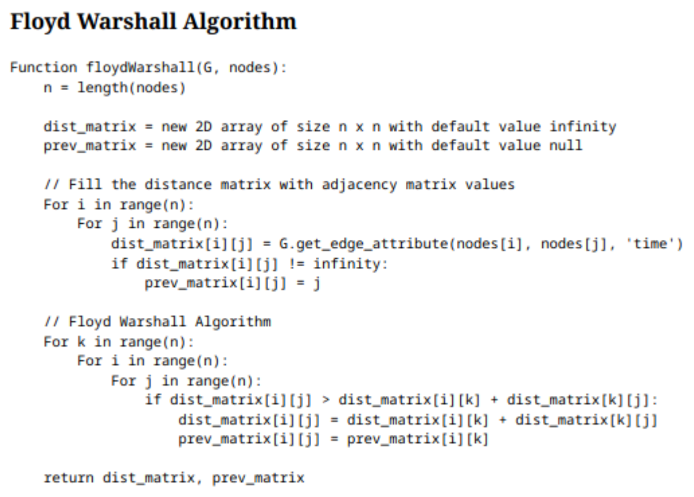
The Dijkstra implementation in this case takes a graph and target node as input. The function then initialises two dictionaries to store the distance to each visited node and the previous node in the shortest path alongside a minimum heap priority queue based on distance to store the nodes to be visited. The initialisation of these data structures is completed in *O(1)* operations. The function proceeds to add the start node to these data structures, adding elements to the dictionaries in *O(1)* operations by simply hashing the key and placing the value in the corresponding memory location. The priority queue works differently depending on the language. In Python, using the heapq library, the push operation takes *O(log n)* operations, where n is the number of elements in the heap. As the length of the heap can be V items long in the worst case, n is equal to V. Analysing the heapq Priority Queue operations is outside of the scope of this report; therefore, the time complexities will just be taken at face value. The function then iterates over all other nodes in the graph, setting the distances to infinity, which takes $O(|V|*(\log V+ 2))$ operations to complete.

The Dijkstra algorithm then iterates over the priority queue, popping the node with the smallest distance, iterating over its neighbours, and updating the distance and previous node if a shorter path is found. The overall time complexity of the main while loop simplifies to $O(|V| + 6|E| + 2\log V)$. Finally, the algorithm creates a path variable and reconstructs the path from the previous array, which, in the worst case where all nodes are travelled in the path, takes $O(|V|)$ operations. All of these time complexity operations combine to a total worst-case operation count of $O(|V|(\log V + 4) + 2|E|*\log V + 6|V|)$ or, in asymptotic behaviour form, $O(|V| + |E| * \log V)$ — excluding n because it is dependent on the priority queue’s size and not particularly relevant.

Dijkstra's algorithm, with its polynomial time complexity $O(|V|+|E| * \log V)$ is highly efficient and well-suited for real-world applications involving routing and pathfinding. Its ability to handle large graphs efficiently ensures that response times remain quick, which is crucial in emergency situations like the Pangobat infestation. The algorithm's scalability allows it to manage extensive networks, providing reliable and rapid route calculations. This makes Dijkstra's algorithm an ideal choice for scenarios where timely and accurate pathfinding is essential, such as in disaster response, transportation logistics, and network routing. The algorithm is suitable for even massive networks and is commonly used on GPS systems, network packet routing, and in various optimization problems. In these applications, the graph sizes are usually manageable with modern computational resources.

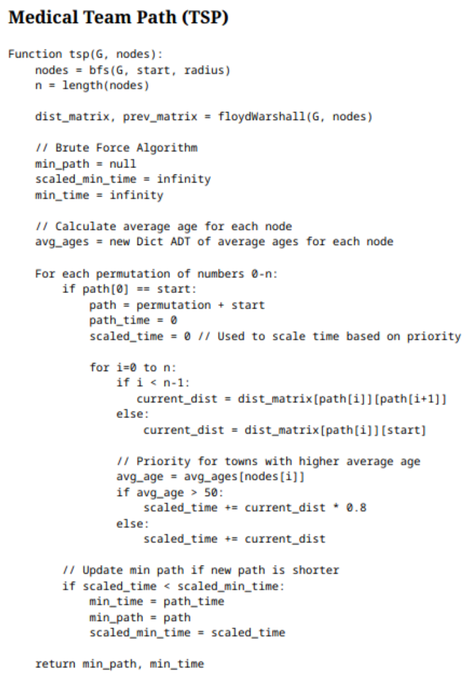
The deployment of these algorithms significantly enhances the Victorian government’s response capabilities to the Pangobat infestation and the spread of Itchy Nose Syndrome (INS). The analysis of the algorithms, specifically the single-source shortest path algorithm derived from Dijkstra's algorithm and the Travelling Salesman Problem (TSP) algorithm, reveals their respective strengths and limitations in real-world applications.

Pseudocode analysed:



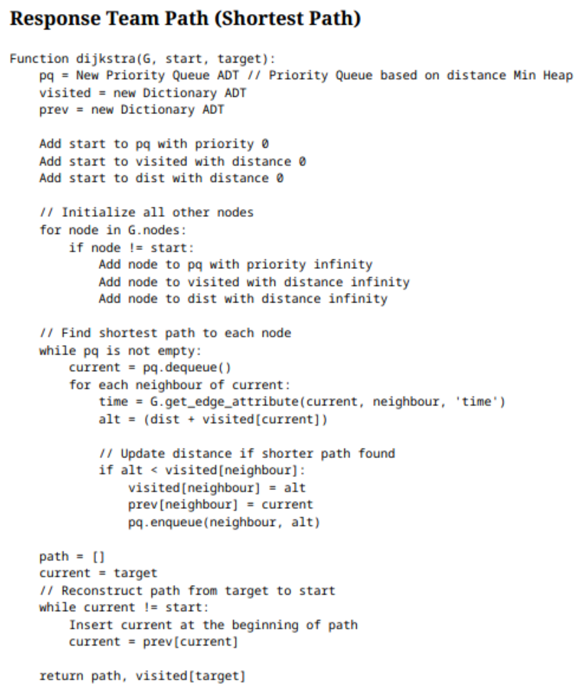
Floyd Warshall Algorithm

```
Function floydWarshall(G, nodes):
    n = length(nodes)

    dist_matrix = new 2D array of size n x n with default value infinity
    prev_matrix = new 2D array of size n x n with default value null

    // Fill the distance matrix with adjacency matrix values
    For i in range(n):
        For j in range(n):
            dist_matrix[i][j] = G.get_edge_attribute(nodes[i], nodes[j], 'time')
            if dist_matrix[i][j] != infinity:
                prev_matrix[i][j] = j

    // Floyd Warshall Algorithm
    For k in range(n):
        For i in range(n):
            For j in range(n):
                if dist_matrix[i][j] > dist_matrix[i][k] + dist_matrix[k][j]:
                    dist_matrix[i][j] = dist_matrix[i][k] + dist_matrix[k][j]
                    prev_matrix[i][j] = prev_matrix[i][k]

    return dist_matrix, prev_matrix
```



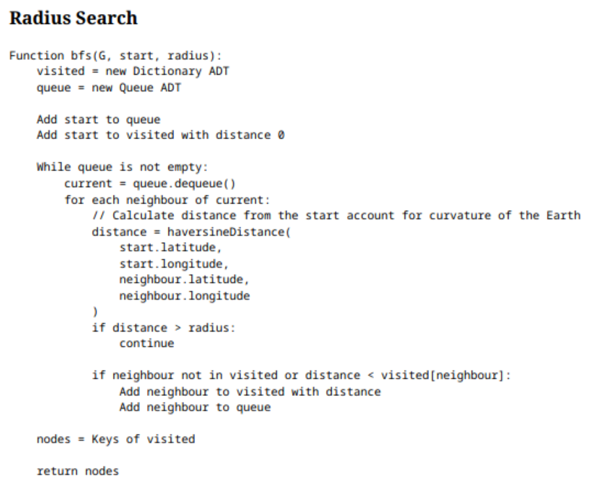
Medical Team Path (TSP)

```
Function tsp(G, nodes):
    nodes = bfs(G, start, radius)
    n = length(nodes)

    dist_matrix, prev_matrix = floydWarshall(G, nodes)

    // Brute Force Algorithm
    min_path = null
    scaled_min_time = infinity
    min_time = infinity

    // Calculate average age for each node
    avg_ages = new Dict ADT of average ages for each node

    For each permutation of numbers 0-n:
        if path[0] == start:
            path = permutation + start
            path_time = 0
            scaled_time = 0 // Used to scale time based on priority

            for i=0 to n:
                if i < n-1:
                   current_dist = dist_matrix[path[i]][path[i+1]]
                else:
                    current_dist = dist_matrix[path[i]][start]

                // Priority for towns with higher average age
                avg_age = avg_ages[nodes[i]]
                if avg_age > 50:
                    scaled_time += current_dist * 0.8
                else:
                    scaled_time += current_dist

        // Update min path if new path is shorter
        if scaled_time < scaled_min_time:
            min_time = path_time
            min_path = path
            scaled_min_time = scaled_time

    return min_path, min_time
```



Response Team Path (Shortest Path)

```
Function dijkstra(G, start, target):
    pq = New Priority Queue ADT // Priority Queue based on distance Min Heap
    visited = new Dictionary ADT
    prev = new Dictionary ADT

    Add start to pq with priority 0
    Add start to visited with distance 0
    Add start to dist with distance 0

    // Initialize all other nodes
    for node in G.nodes:
        if node != start:
            Add node to pq with priority infinity
            Add node to visited with distance infinity
            Add node to dist with distance infinity

    // Find shortest path to each node
    while pq is not empty:
        current = pq.dequeue()
        for each neighbour of current:
            time = G.get_edge_attribute(current, neighbour, 'time')
            alt = (dist + visited[current])

            // Update distance if shorter path found
            if alt < visited[neighbour]:
                visited[neighbour] = alt
                prev[neighbour] = current
                pq.enqueue(neighbour, alt)

    path = []
    current = target
    // Reconstruct path from target to start
    while current != start:
        Insert current at the beginning of path
        current = prev[current]

    return path, visited[target]
```



Radius Search

```
Function bfs(G, start, radius):
    visited = new Dictionary ADT
    queue = new Queue ADT

    Add start to queue
    Add start to visited with distance 0

    While queue is not empty:
        current = queue.dequeue()
        for each neighbour of current:
            // Calculate distance from the start account for curvature of the Earth
            distance = haversineDistance(
                start.latitude,
                start.longitude,
                neighbour.latitude,
                neighbour.longitude
            )
            if distance > radius:
                continue

            if neighbour not in visited or distance < visited[neighbour]:
                Add neighbour to visited with distance
                Add neighbour to queue

    nodes = Keys of visited

    return nodes
```