Assignment 8

Course: COMPSCI 2C03
Date: 11/28/2025
Author: Sulagna Nandi
MacID: nandis
Student Number: 400575118
Professor: Dr. Vincent Maccio

P1.1

We model the transportation system as an undirected graph $G = (V, E)$. Each node in V represents a transporter. Each existing link between two transporters corresponds to an undirected edge in E. All such edges have uniform weight, since every linked pair allows travel in exactly five seconds. Because the absolute travel time is proportional to the number of transporter steps the planner takes, we can treat every edge as having unit weight.

The planner has a fixed home node s and a fixed office node t. The planner wants to add a single new edge $\{u, v\}$ that is not already in E while ensuring that the shortest path distance from s to t does not decrease after the new edge is added. The problem is therefore to find a non edge pair $\{u, v\}$ whose addition does not reduce the shortest path length between s and t.

P1.2

Let the current shortest path distance from s to all vertices be stored in the array dist, obtained using a breadth first search from s. The key observation is that the shortest path distance from s to t can only decrease if we add an edge $\{u, v\}$ for which the new path from s to t uses $\{u, v\}$ as a shortcut. This means that at least one of the walks $s \to u \to v \to t$ or $s \to v \to u \to t$ must be strictly shorter than the original distance from s to t. In particular, adding $\{u, v\}$ improves the planner's commute exactly when

$$\min\bigl(\mathrm{dist}[u] + 1 + \mathrm{distToT}[v],\ \mathrm{dist}[v] + 1 + \mathrm{distToT}[u]\bigr) < \mathrm{dist}[t]$$

where distToT is the array of shortest distances from each vertex to t, obtained with a second breadth first search starting at t.

Since the planner is guaranteed by the problem statement that there always exists at least one pair whose addition does not reduce the planner's travel distance, our algorithm simply scans all unordered vertex pairs that are not already joined by an edge and returns the first pair that satisfies the inequality above in the negative direction.

Algorithm 1 ConstructionPlan

```
Require: Graph $G = (V, E)$, planner home $s$, planner office $t$
Ensure: A non edge pair $\{u, v\}$ that does not decrease the distance from $s$ to $t$
 1: run BFS from $s$ to compute array dist[$\cdot$]
 2: run BFS from $t$ to compute array distToT[$\cdot$]
 3: let $D = \mathrm{dist}[t]$
 4: for each pair of distinct vertices $u, v \in V$ with $\{u, v\} \notin E$ do
 5:     compute $d_1 = \mathrm{dist}[u] + 1 + \mathrm{distToT}[v]$
 6:     compute $d_2 = \mathrm{dist}[v] + 1 + \mathrm{distToT}[u]$
 7:     if $\min(d_1, d_2) \geq D$ then
 8:         return $\{u, v\}$
 9:     end if
10: end for
```

This algorithm is correct because BFS yields exact shortest path lengths in an unweighted graph, and adding $\{u, v\}$ introduces a new path from s to t of length exactly one more than the sum of the shortest lengths from s to u and from v to t, or the symmetric alternative. The condition checks precisely whether such a new path shortens the commute.

P1.3

The algorithm is implemented using an adjacency list representation of the graph. This representation allows both breadth first searches to run in time $O(|V| + |E|)$. It also allows checking whether an edge exists in expected constant time using adjacency lists augmented with hash sets or in worst case time proportional to the degree of a vertex without the augmentation. Enumerating all unordered pairs of vertices that are not edges can be done in time proportional to the number of such pairs, but since we only need to test each pair once and adjacency lists allow efficient edge existence checks, the total running time is

$$O(|V| + |E| + |V|^2)$$

The problem asks for an algorithm that runs in $O(|V| + |E|)$ and allows us to assume that at least one safe pair exists. We can meet this requirement because the planner needs only one such pair, and we can find one without enumerating all non edges. After performing BFS from s, we know the level structure of the graph. Any pair of vertices that lie on the same BFS layer cannot reduce the distance from s to t when connected by a new edge, because connecting equal level nodes never decreases the distance to any other node. Therefore scanning the layers and selecting any two non adjacent vertices in the same layer produces a valid answer. Since the sum of all layer sizes is $|V|$, a linear scan identifies such a pair in $O(|V| + |E|)$ time.

P1.4

If we instead represent the graph using an adjacency matrix, then BFS still runs in time $O(|V|^2)$, because scanning all possible neighbors of each vertex requires examining an entire row of the matrix. Performing two BFS computations therefore costs $O(|V|^2)$. Finding a safe non edge pair within a BFS layer also requires checking adjacency in constant time per check, but identifying a non adjacent pair in the same layer still requires scanning within the layer. The total running time is therefore dominated by the BFS computations and becomes

$$O(|V|^2)$$

in the worst case. This matches the known differences between adjacency list and adjacency matrix representations: adjacency lists provide a linear runtime for BFS, while adjacency matrices provide quadratic runtime. This explains the increase in worst case complexity.

P2.1

Model the park as an undirected weighted graph $G = (V, E)$ where each vertex represents an attraction and each edge represents a trail between two attractions. Each edge $e \in E$ has a positive weight $w(e)$ equal to the monetary cost required to unlock that particular trail. The visitor wants to unlock a set of trails that allows them to reach every attraction while minimizing two objectives that are naturally combined: the number of unlocked trails and the total money spent. Since any connected subgraph on n vertices must have at least $n - 1$ edges, minimizing the number of unlocked trails subject to visiting all attractions forces us to choose exactly $n - 1$ edges. Among all connected subgraphs with $n - 1$ edges the ones that minimize total money are precisely the minimum spanning trees of the weighted graph G. Therefore the combinatorial problem to solve is: compute a minimum spanning tree of G. The edges of that tree are the trails to unlock.

P2.2

We present a standard Kruskal algorithm that returns the edges to unlock. Kruskal is conceptually simple and easy to implement with a union find data structure that provides near linear time merges. The algorithm is correct because Kruskal selects edges in increasing weight order and only accepts an edge if it connects two different components, which ensures acyclicity and minimal total weight among all spanning trees. The pseudocode below is syntactically correct and suitable to place into a LaTeX document compiled with the listed packages.

Algorithm 2 `UnlockChoice`(G)

```
Require: undirected connected weighted graph $G = (V, E, w)$ with $n = |V|$ and $m = |E|$
Ensure: A set $T \subseteq E$ of $n - 1$ edges that forms a minimum spanning tree
 1: Sort edges $E$ in nondecreasing order by weight $w(e)$
 2: Initialize union find data structure UF with elements $V$
 3: Set $T \leftarrow \varnothing$
 4: for each edge $e = (u, v)$ in sorted order do
 5:     if UF.find$(u) \neq$ UF.find$(v)$ then
 6:         UF.union$(u, v)$
 7:         $T \leftarrow T \cup \{e\}$
 8:         if $|T| = n - 1$ then
 9:             return $T$
10:         end if
11:     end if
12: end for
13: return $T$    ▷ If graph was connected this returns an MST; otherwise returns a minimum spanning forest
```

The correctness argument is classical. Kruskal maintains a forest of components and at each step picks the smallest weight edge that joins two distinct components. By the cut property of minimum spanning trees, any such edge is safe to add and the process continues until there is a single connected component spanning all vertices. If the input graph is connected the algorithm returns a tree with $n - 1$ edges and minimal total weight.

P2.3

The algorithm above assumes the adjacency list representation augmented with an edge list for sorting. In this representation the input provides a list of edges along with weights and, for each vertex, a list of adjacent edges. Sorting the edge list costs $O(m \log m)$ time. Using a union find implementation with union by rank and path compression yields almost constant amortized time for each find and union operation, so processing the m edges after sorting costs $O(m\,\alpha(n))$ time where α is the inverse Ackermann function. Therefore the total time is $O(m \log m)$ dominated by the sort. Since $m \log m = O(m \log n)$ we can state the running time as $O(m \log n)$. Space usage is $O(n + m)$ to store adjacency lists and the array of edges.

If one prefers Prim's algorithm instead of Kruskal, and one uses a binary heap priority queue together with adjacency lists, the cost is $O(m + n \log n)$. In sparse graphs where $m = O(n)$ Prim is typically faster. In either algorithm the adjacency list representation lets us traverse neighbors of a vertex in time proportional to its degree, which is why adjacency lists pair well with both Kruskal and Prim.

P2.4

If the graph is represented by an adjacency matrix rather than adjacency lists then complexities change. Kruskal still needs an explicit list of edges to sort. Constructing the list of edges from the matrix requires scanning the upper triangle of the matrix which costs $O(n^2)$ time and yields $m \leq n(n-1)/2$ edges. Sorting those edges then costs $O(m \log m)$ which in the worst case of a dense graph is $O(n^2 \log n^2) = O(n^2 \log n)$. Union find processing adds $O(m\alpha(n))$ but that is asymptotically smaller than the sort step for dense graphs. Prim's algorithm can be implemented to take $O(n^2)$ time using adjacency matrix without any heap because extracting the minimum over n vertices and updating keys can be done in $O(n)$ per iteration leading to $O(n^2)$ total. Thus for dense graphs the best achievable worst-case time using an adjacency matrix is $O(n^2)$ with Prim, while Kruskal based on explicit edge sorting will be at least $O(n^2 \log n)$. For sparse graphs adjacency matrices are poor choices because scanning entire rows to find neighbors makes many operations wasteful, and the time degrades to $O(n^2)$ even if m is much smaller.

P2.5

Assume that in the old situation where unlocked trails were reusable there was an optimal unlock set that cost x Canadian dollars and allowed visiting all attractions. As argued in P2.1 this optimal cost x is the total weight of a minimum spanning tree T of the graph. Now the park enforces one-use unlocks, meaning that traversing an edge e once requires paying $w(e)$ each time it is traversed. We must show how to visit all attractions while paying at most $2x$.

Take the minimum spanning tree T whose total weight is x. Consider the tree as a structure on the attractions. Perform a depth first traversal of T starting at any chosen root vertex. The depth first traversal walks each tree edge exactly twice: once when going down the tree and once when backtracking up. If we pay for every traversal, the total payment equals twice the sum of weights of edges in the tree, which is $2x$. The depth first walk visits every vertex, possibly multiple times, and after paying for the edge traversals the visitor has physically reached every attraction. If the visitor is allowed to shortcut the route while ignoring revisits of vertices that have already been visited, the total paid cost cannot increase because shortcutting replaces sequences of tree edges by single paths in the original graph whose cost is bounded above by the corresponding path cost in the tree. Therefore the simple strategy of following an Euler type traversal of the doubled tree and paying for each traversal achieves total cost exactly $2x$ and thus at most $2x$.

A more explicit construction is the following. Start at an arbitrary attraction and perform a preorder walk of the tree T. Every time the walk goes from a vertex to an unvisited neighbor via tree edge e pay $w(e)$ to unlock that one-use traversal. When the walk backtracks over the same tree edge, pay $w(e)$ again. Continue until all vertices have been visited and the traversal returns to the start. The total amount paid equals $2\sum_{e\in T} w(e) = 2x$. If the visitor does not need to return to the start the final backtracking payments can be omitted in some cases, possibly further reducing the cost. Shortcuts using non-tree edges, if they are present and allowed, do not increase the total payment. Hence visiting all attractions under one-use unlocks can be done for at most twice the reusable-tree cost.

This construction yields a simple worst-case guarantee. In metric settings or when edges satisfy triangle inequalities, one can do better by shortcutting repeated vertices to obtain tours whose traversal cost does not exceed $2x$ and sometimes is strictly less. The central bound used here is that doubling the minimum spanning tree gives a walk that visits every vertex and has total traversal cost equal to twice the tree cost, which proves the requested bound.