Problème 1.

A regional government wants to improve their existing infrastructure between a collection of towns T. In specific, the government want to build a minimum number of roads such that there is a route from each town to each other town. The government has been advised by a dubious consultant that in the resulting road network, the number of users of a given road is independent of the presence of alternative routes. The regional government wants to minimise the number of roads it has to built to ensure that one can travel from one town to the other. Furthermore, the government wants to maximize the benefits of the road network by maximizing the number of users of the roads built. Hence, the government wants to only build roads that are expected to be used often. To help the construction plans, the government has asked the dubious consultant to estimate, for each pair of cities, the number of road users that would use the road between these two cities (if that road was built). Now the regional government is looking for a construction plan for a minimum number of roads connecting all towns that see the highest total usage among them.

P1.1

Model the above problem as a graph problem: What are the nodes and edges in your graph, do the edges have weights, and what problems are you trying got answer on your graph?

- **Nodes**: Each town in the set of town T is represented as a node in the graph. Denote the set of nodes as V
- **Edges**: Each potential road can be built between two pair of town. Denote it as E. Each edge weights represents the estimated number of users who would use the road should it is constructed. Denote the weight of an edge between nodes i and j as $w(i, j)$.

The problem can then be modelled as given a weighted undirected graph $\mathcal{G} = (V, E)$ representing towns and potential roads, find a maximum weight minimum spanning tree of G.

Explanation:

- We need to ensure there is a route from each town to every other town while minimising the number of roads being built, thus the minimum spanning tree of the graph.
- Among all possible MSTs, we want to find the one with the maximum total edge weight, thus maximum weight MSTs.

P1.2

Provide an algorithm ConstructionPlan to find the minimum number of roads to build. Explain why your algorithm is correct.

Let $w(x, y)$ be the weight of the edge between nodes x and y.

Algorithm ConstructionPlan(G, T)

Require: Graph $G = (T, E)$ with nodes T representing towns and weighted edges E representing potential roads with weights as estimated road usage
Ensure: Set of edges E' representing roads to build for maximum weight minimum spanning tree

```
Adj ← new list[|T|]//Adjacency list for graph
for (u, v, w) ∈ E do
    Adj[u].append((v, w))
    Adj[v].append((u, w))
end for
E' ← ∅
Pick an arbitrary node s ∈ T
V ← {s}
Q ← new min-priority queue//Priority queue of edges
for (v, w) ∈ Adj[s] do
    Q.insert((s, v, w))
end for
while |V| < |T| do
    (u, v, w) ← Q.extract_max()//Get max weight edge
    if v ∉ V then
        E' ← E' ∪ (u, v)
        V ← V ∪ v//Mark v as visited
        for (x, w) ∈ Adj[v] do
            if x ∉ V then
                Q.insert((v, x, w))
            end if
        end for
    end if
end while
return E'
```

This is the Prim's algorithm for finding MSTs. We instead replace the minimum weight with maximum weights to fit with problem description in P1.1

Correctness:

Invariant:
V contains all visited nodes, E' contains edges of a maximum weight spanning tree over nodes in V

- At L4, invariant holds since $V = \{s\}$ and $E' = \emptyset$

- per iteration, we add E' with the maximum weight edge (u, v) from any visited nodes $u \in V$ to any unvisited node $v \in T \setminus V$ (L6-14). Then add v to V (L15), maintaining the invariant because:
  - $V_{\text{new}} = V \cup \{v\}$
  - $E'_{\text{new}} = E' \cup \{(u, v)\}$ is a maximum weight spanning tree over V_{new} due to (u, v) is the maximum weight edge connecting V to $T \setminus V$.

The algorithm never create cycle in E' since we only add edge from visited nodes to unvisited nodes, which cannot create a cycle.

Bound function:

Let $w(E')$ be the total weight of edges in E'. Per iteration, $w(E')$ is the maximum possible weight of any spanning tree over the nodes in V.

This holds initially when $V = \{s\}$ and $w(E') = 0$. Per iteration, we add the maximum weight edge (u, v) to E' from V to $T \setminus V$. Bound function is maintained because:

- $w(E'_{\text{new}}) = w(E') + w(u, v) \geq w(E')$
- Any other spanning tree T' over $V \cup \{v\}$ must contain an edge (x, y) from V to $\{v\}$, and $w(x, y) \leq w(u, v)$. Therefore, $w(T') \leq w(E' \cup \{(u, v)\})$

The loop terminates when $|V| = |T|$, i.e all nodes are visited.

Thus the algorithm is correct. □

> **P1.3**
>
> Explain which graph representation you used for your algorithm and what the complexity of your algorithm is using this graph representation.

Uses adjacency list representation of the graph $G = (T, E)$. Since we store a list of edges for each nodes, or for each node $u \in T$, we maintain a list $\text{Adj}[u]$ containing the node v and weight $w(u, v)$ for every edge $(u, v) \in E$.

The time complexity of the algorithm is $\mathcal{O}(|T|^2 \log |T| + |E|)$:

1. Initialising adjacency list: $\mathcal{O}(|E|)$, where the adjacency list takes $O(|E|)$ to populate
2. L4: Run for $|T|$ iterations
3. Inside the loop, find maximum weight edge takes from Q takes $O(\log |T|)$. Therefore each iteration the while loop takes $O(|T| \log |T|)$
4. Adding extracted edge to E' takes $O(1)$

Thus, the overall time complexity is $\mathcal{O}(|T|^2 \log |T| + |E|)$. If the graph is complete, then $|E| = \frac{|T|(|T|-1)}{2} = O(|T|^2)$, and the time complexity is $\mathcal{O}(|T|^2 \log |T|)$

The space complexity is $O(|T| + |E|)$

P1.4

What is the worst-case complexity of your solution if you use the other graph representation? Explain your answer

Worst case complexity of the algorithm using adjacency matrix representation is $\mathcal{O}(|T|^2 \log |T|)$.

If adjacency matrix representation is used, or we use a $|T| \times |T|$ matrix Adj to represent the graph, where $\text{Adj}[u][v] = w$ for an edge (u, v) with weight w.

Initialising the matrix will take $O(|T|^2)$ time.

Adding edges from starting node s to priority queue Q will take $O(|T| \log |T|)$ time, as we need to scan row $\text{Adj}[s]$ which has $|T|$ elements.

Inside the while loop, for loop (L18-23) that iterates over the edges of the newly visited node v now takes $O(|T| \log |T|)$ time, as we need to scane the row that has $|T|$ entries, and for each unvisited node, we insert the edge into Q which takes $O(\log |T|)$ time. Therefore, the while loop will take $O(|T| \cdot (|T| \log |T|)) = O(|T|^2 \log |T|)$

Thus, the total time complexity would be $O(|T|^2 + |T|^2 \log |T|) = O(|T|^2 \log |T|)$

Whereas the space complexity is $O(|T|^2)$. Could be more efficient for sparse graph where $|E| \ll |T|^2$.

Problème 2.

A directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ with $s \in \mathcal{N}$ the *source* and $t \in \mathcal{N}$ the *target* is a series-parallel graph if it can be constructed inductively using the following rules:

1. An *elementary* series-parallel graph is a single edge from s to t
2. The *series*-construction. Let $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1, s_1, t_1)$ and $\mathcal{G}_2 = (\mathcal{N}_2, \mathcal{E}_2, s_2, t_2)$ be two series-parallel graphs with the node c in common ($\mathcal{N}_1 \cap \mathcal{N}_2 = \{c\}$). The graph $\mathcal{G} = (\mathcal{N}_1 \cup \mathcal{N}_2, \mathcal{E}_1 \cup \mathcal{E}_2)$ is a series-parallel graph

3. The *parallel*-construction. Let $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1, s_1, t_1)$ and $\mathcal{G}_2 = (\mathcal{N}_2, \mathcal{E}_2, s_2, t_2)$ be two series-parallel graphs without nodes in common ($\mathcal{N}_1 \cap \mathcal{N}_2 = \emptyset$) and let $s, t \notin (\mathcal{N}_1 \cup \mathcal{N}_2)$ be two fresh nodes. The graph
$\mathcal{G} = (\mathcal{N}_1 \cup \mathcal{N}_2 \cup \{s, t\}, \mathcal{E}_1 \cup \mathcal{E}_2 \cup \{(s, s_1), (s, s_2), (t_1, t), (t_2, t)\})$ is a series-parallel graph.

Now assume we have a series-parallel graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$ with an edge-weight function $weight : \mathcal{E} \to \mathbb{Z}$ (here, $\mathbb{Z}$ are the integers, which includes negative numbers). We note that series-parallel graphs are relatively simple structures.

P2.1

Write an algorithm to compute the single-source shortest paths from the source s to all nodes $n \in \mathcal{N}$ in $\mathcal{O}(|\mathcal{N}| + |\mathcal{E}|)$ time

---

Algorithm ShortestPathsSP($\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$, $weight$)

$dist \leftarrow$ //Initialize empty dictionary for distances
if $\mathcal{G}$ is an elementary graph (single edge (s, t)) **then**
  $dist[s] \leftarrow 0$
  $dist[t] \leftarrow weight(s, t)$
else if $\mathcal{G}$ is a series composition of $\mathcal{G}_1$ and $\mathcal{G}_2$ **then**
  $dist_1 \leftarrow ShortestPathsSP(\mathcal{G}_1, weight)$
  $dist_2 \leftarrow ShortestPathsSP(\mathcal{G}_2, weight)$
  $dist \leftarrow dist_1 \cup dist_2$//Combine distances
  for each node n in $\mathcal{G}_2$ **do**
    $dist[n] \leftarrow dist[n] + dist_1[t_1]$
  end for
else if $\mathcal{G}$ is a parallel composition of $\mathcal{G}_1$ and $\mathcal{G}_2$ **then**
  $dist_1 \leftarrow ShortestPathsSP(\mathcal{G}_1, weight)$
  $dist_2 \leftarrow ShortestPathsSP(\mathcal{G}_2, weight)$
  $dist \leftarrow dist_1 \cup dist_2$//Combine distances
  $dist[s] \leftarrow 0$
  $dist[t] \leftarrow \min(dist_1[t_1], dist_2[t_2])$
end if
return $dist$

P2.2

Explain why your algorithm is correct

Base case: For an elementary graph with one edge (s, t), it sets $dist[s] = 0$ and $dist[t] = weight(s, t)$, which is the shortest path from s to t.

If $\mathcal{G}$ is a series-composition: Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be the two series-parallel graphs with common node c.

- It recursively computes the shortest paths from s to all nodes in $\mathcal{G}_1$ and $\mathcal{G}_2$, store them in $dist_1$ and $dist_2$ respectively.
- For any node $n \in \mathcal{G}_2$, the shortest path from s \mu st go through target t_1 of $\mathcal{G}_1$. Thus, we update $dist[n] = dist[n] + dist_1[t_1]$.

If $\mathcal{G}$ is a parallel-composition: Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be the two series-parallel graphs with new source s and target t

- It recursively computes the shortest paths from s to all nodes in $\mathcal{G}_1$ and $\mathcal{G}_2$, store them in $dist_1$ and $dist_2$ respectively.
- The new source s has distance 0, correctly set by $dist[s] = 0$
- The new target t has distance $\min(dist_1[t_1], dist_2[t_2])$, which is the shortest path from s to t.

In both cases, it combines the distance using union operation, and the algorithm is correct. □

P2.3

Explain which graph representation you used for your algorithm and why your algorithm has the stated complexity

It uses adjacency list representation of the graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, s, t)$. Each node maintains a list of its outgoing edges. Since in series-parallel graphs, each node has at most 2 outgoing edges, the adjacency list representation is efficient. Additionally, the algorithm recursively traverse the graph, and finding outgoing edges takes constant time.

For base case, the algorithm takes $O(1)$ time to set the distance for the elementary graph.

In both cases, it calls itself on $\mathcal{G}_1$ and $\mathcal{G}_2$, taking $O(|\mathcal{N}_1| + |\mathcal{E}_1|)$ and $O(|\mathcal{N}_2| + |\mathcal{E}_2|)$ time respectively. Then it updates the distance for each node in $\mathcal{G}_2$ in $O(|\mathcal{N}_2|)$ time. Thus, the time complexity is $O(|\mathcal{N}_1| + |\mathcal{E}_1| + |\mathcal{N}_2| + |\mathcal{E}_2|)$

Since each node is visited exactly once during the traversal, the time complexity is $O(|\mathcal{N}| + |\mathcal{E}|)$ (since $|\mathcal{N}_1| + |\mathcal{N}_2| = |\mathcal{N}|$ and $|\mathcal{E}_1| + |\mathcal{E}_2| = |\mathcal{E}|$)

P2.4

What is the worst-case complexity of your solution if you use the other graph representation? Explain your answer

If using adjacency matrix representation, the worst-case complexity of the algorithm is $O(|\mathcal{N}|^2)$.

It will take $O(|\mathcal{N}|^2)$ time to initialise the adjacency matrix.

The analysis of the algorithm remains the same. The difference here is that finding outgoing edges for a node will take $O(|\mathcal{N}|)$ time in worst-case, as it will have to traverse the entire row of the matrix corresponding to that node. Since the ops is performed on each node during traversal, total complexity would be $O(|\mathcal{N}|^2)$.