CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 13

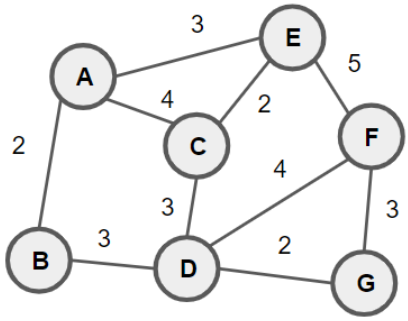
For: April 10–April 14

Goals:

- MST
- Dynamic Programming

Problem 1. MST Review

Problem 1.a.



Can you use the cycle and cut property of MST that we learnt in class to determine which edges must be in the MST?

Perform Prim's, Kruskal's, and Boruvka's (optional) MST algorithm on the graph above.

Solution:

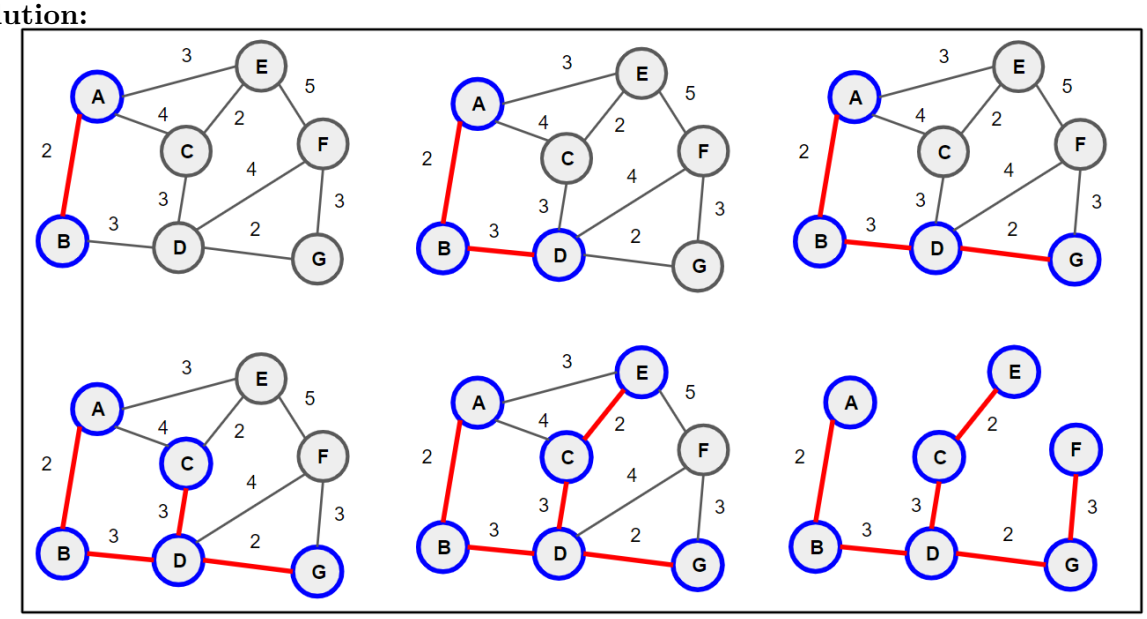


Figure 1: Prim's Algorithm

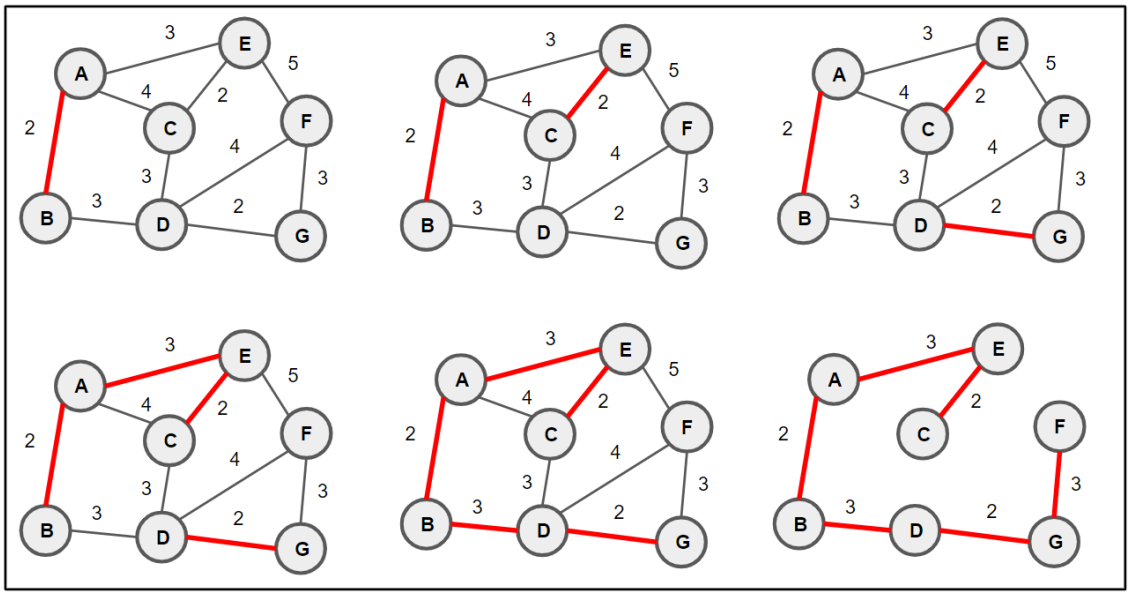


Figure 2: Kruskal's Algorithm

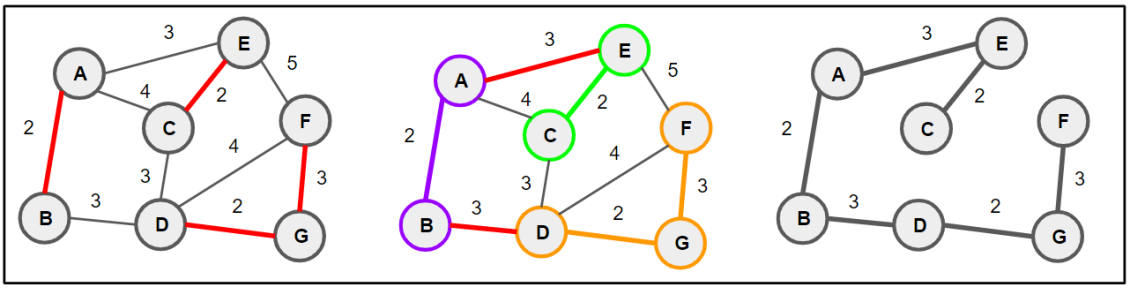


Figure 3: Boruvka's Algorithm

Problem 1.b. Henry The Hacker has some ideas for a faster MST algorithm! Recently, he read about *Fibonacci Heaps*. A Fibonacci heap is a priority queue that implements insert, delete, and decreaseKey in $O(1)$ amortized time, and implements extractMin in $O(\log n)$ amortized time, assuming n elements in the heap. If you run Prim's Algorithm on a graph of V nodes and E edges using a Fibonacci Heap, what is the running time?

Solution: In this case, each node is added into the priority queue at most once, which costs $O(V)$; each edge leads to one decreaseKey operation, which has cost $O(E)$; and each node is extracted once from the priority queue, which results in a cost of $O(V \log V)$. Hence, the total cost is $O(E + V \log V)$.

Problem 2. Divide-and-Conquer MST

Henry The Hacker has invented a new divide-and-conquer algorithm for finding a minimum spanning tree! The algorithm is super simple! It only involves 4 steps:

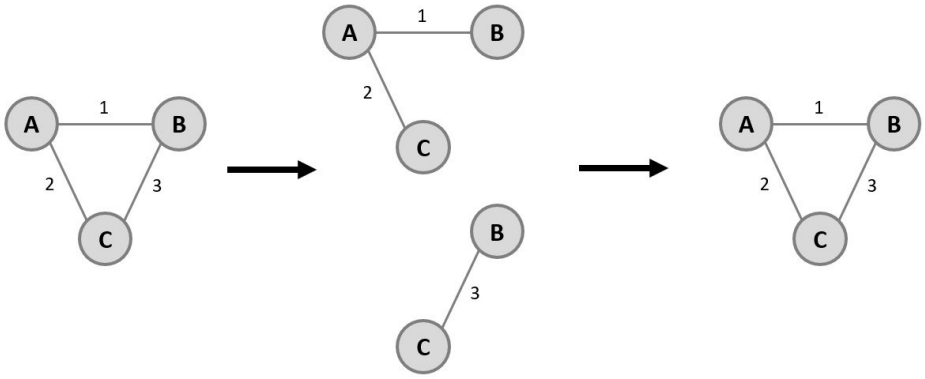
1. Find the median edge weight w_e.
2. Partition the edges using the edges into those $\leq w_e$ and those $> w_e$ into two graphs G_a and G_b.
3. Recursively find the MST for G_a and G_b, yielding trees T_a and T_b.
4. Combine the edges from T_a and T_b, and recursively find the MST for the resulting graph.

How well do you think this algorithm works? (What would happen if it continued to a base case of 1 edge?)

After a little bit of thought, Henry realizes that he can stop the recursion early (i.e., not going down to a base case of 1). Instead, if the number of edges in the graph $E < 4V$, then the recursion terminates and he uses Prim's algorithm to find the MST.

Why do you think this algorithm will return a valid MST (unlike the one we saw in lecture)?

Solution: You might have noticed that if the algorithm continues to the base case of 1, you might be stuck in an infinite loop. E.g. If the original graph have three edges forming a cycle, then by the algorithm you would divide it into two subgraphs of two edges and one edge respectively. Recursing on both subgraphs will yield the same subgraphs since we do not have enough edges to split the subgraphs again. Therefore, when we combine the two subgraphs, they will produce the original graph. By the algorithm, we would recurse again on this graph, leading to an infinite loop.



But introducing Prim's Algorithm to find the MST as the base case solves this issue!

The algorithm analysis itself goes as such

We first find the median in $O(E)$ time, e.g., using the QuickSelect algorithm. (Let's not worry about the issue of randomization.). We then partition the edges into 2 subgraphs, and recurse on each of them. After the two recursive calls on the subgraphs, there are at most $2V$ edges left in total, since each of the recursively constructed MSTs has at most V edges. We would run the algorithm upon merging the 2 subgraphs. Thus the basic recurrence here is:

$$T(V, E) = 2T(V, \frac{E}{2}) + O(E) + T(V, 2V).$$

And remember, the base case says that we use Prim's algorithm if $E < 4V$, so the last recursive call is just a direct call to Prim's. This gives a recurrence of:

$$T(V, E) = 2T(V, \frac{E}{2}) + O(E) + O(V \log V).$$

Since the base case happens before V edges, the recursion tree is going to have depth at most $\log \frac{E}{V}$, with at most $\frac{E}{V}$ leaves and $\frac{2E}{V}$ nodes. The cost of finding the medians is at most $O(E)$ per level of the recursion tree, so the cost in the entire tree is $O(E \log \frac{E}{V})$ in total. On the other hand, the cost of Prim's Algorithm is $O(V \log V)$ per node in the recursion tree, so the cost in the entire tree is $O(\frac{2E}{V} V \log V) = O(E \log V)$ in total. Hence, the total cost of the entire algorithm is $O(E \log \frac{E}{V} + E \log V) = O(E \log V)$, the same as Prim's and Kruskal's!

Solution: (Continued from previous page)
This algorithm is different from the one we saw in class. So why is this variant correct? The simple answer is that when you find the recursive MST on a subset of the edges E', every edge eliminated is a "red edge", i.e. an edge that is the heaviest edge on a cycle among the edges in E'; therefore it is clearly also the heaviest edge on a cycle among all the edges in E (because E' is a subset of E). So it is safe to eliminate edges from the MST when looking only at a subset of edges.

There are in fact benefits to this type of approach. For example, we can adopt this to build cache-efficient MST algorithms (though you will want to do the merging more efficiently so you do not have to run Prim's every time).

Problem 3. Road Trip

Relevant Kattis Problems:

- https://open.kattis.com/problems/adventuremoving4
- https://open.kattis.com/problems/highwayhassle
- https://open.kattis.com/problems/roadtrip

You are going on a road trip. You get in your trusty car and drive to Panglossia, where you will spend a nice vacation by the beach.

You have already found the best route from your home to Panglossia (using Dijkstra's Algorithm). Next, you need to determine where you can buy petrol along the way. On the road between your home and Panglossia, there are a set of n petrol stations, the last of which is in Panglossia itself. Assume that your trip is complete when you reach the last station.

By searching on the internet, you find for each station, its location on the road and the cost of petrol. That is, the input to the problem is n stations, $s_0, s_1, \ldots, s_{n-1}$, along with $c(s_i)$, which specifies the cost of petrol at station s_i, and $d(s_i)$, which specifies the distance from your home to station s_i.

The tank of your car has a capacity of L liters. You begin your trip at home with a full tank of petrol. Your car uses exactly 1 liter per kilometer. Along the way, you must ensure that your car always has petrol (though you may arrive at a station just as you run out of petrol). Note that you do not need to fill your tank at a station. You can buy any amount of petrol, as long as it's within the capacity of your tank.

Your job is to determine **how much petrol to buy at each station** along the way so as to minimize the cost of your trip.

You may assume, for simplicity, that L and all the given distances are integers, i.e., all the quantities are integers.

Here are two examples, a simple one and a more complicated one.

Example 1: Your tank holds 6 liters, and there are 3 stations:

```
5km: $1
6km: $2
7km: $4
```

In this case, the best solution costs one dollar, where you purchase one liter of petrol at the first station at a cost of 1 per liter.

Example 2: Your tank holds 20 liters, and there are 10 stations:

```
7km:  $3
17km: $5
20km: $2
23km: $1
39km: $5
48km: $4
66km: $9
83km: $9
88km: $4
92km: $1
```

In this case, the best solution is to purchase petrol at each station as follows, for a total cost of 327 dollars:

```
0
0
3
20
5
20
15
5
4
0
```

Hint: To solve this problem, think about how to calculate $DP(s_j, k)$, the minimum cost to get from station s_j to the destination, assuming you have k liters of petrol left.

Solution: This is a typical dynamic programming style problem where the subproblems involve computing $DP(s_j, k)$ (the minimum cost to get from station j to station $n-1$) for each station s_j and each value of k. Let's use s_{-1} to denote your home.

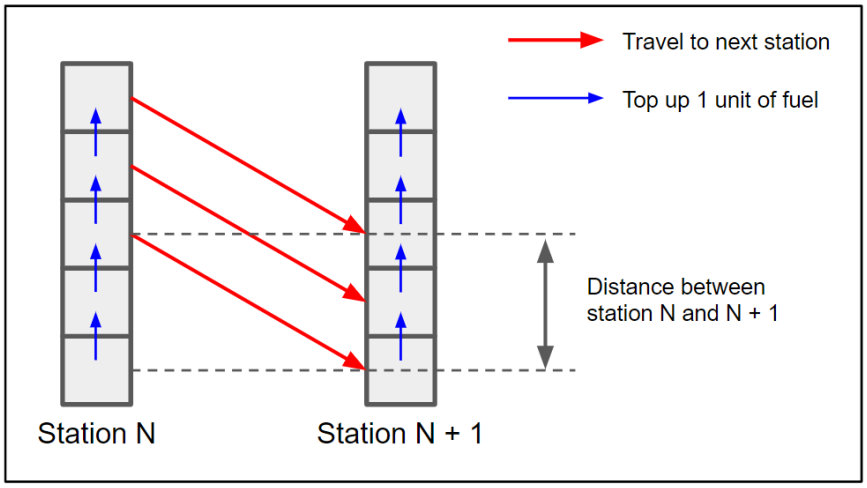
You can observe that if you have already solved the problem for all $j' > j$, then it is not hard to compute $DP(s_j, k)$. Since you have k liters and it is $h = d(s_{j+1}) - d(s_j)$ kilometers to the next station, for all $i \leq L$, lookup $DP(s_{j+1}, k-h+i)$, which is the minimum cost to go from s_{j+1} to s_{n-1} with $k-h+i$ liters of petrol, add $c(s_j) \times i$ to it, and keep track of the i that gives you the lowest cost. The value of i is the amount of petrol you should buy at station s_j. This gives the following state transfer equation:

$$DP(s_j, k) = \min_{max(0,h-k) \leq i \leq L-k} (DP(s_{j+1}, k-h+i) + c(s_j) \times i)$$

Run your DP backward from s_{n-1} until you get back to s_{-1}, you will discover the cheapest way to reach your destination.

The complexity of this solution is $O(nL^2)$, which is essentially exponential on the number of bits in L.

You may refer to `roadtrip_dp.java` for a Java implementation of this DP solution for Kattis problem Road Trip.



Alternatively, you may consider running your DP forward (from s_{-1} to s_{n-1}) and define $DP_1(s_j, k)$ to be the minimum cost to get from your home to station s_j. To reach s_j with k litres remaining, you can decide to either go from s_{j-1} with $k+h$ litres or pump 1 liter of petrol at s_j with $k-1$ litres remaining. In other words:

$$DP'(s_j, k) = min(DP'(s_{j-1}, k-h), DP'(s_j, k-1) + c(s_j))$$

The complexity of this solution is $O(nL)$, which still depends on L.

Solution:
Extension: In fact, there exists a greedy solution which is much more efficient than the DP solution presented above. The idea is to maintain a priority queue of petrol prices, which allows us to keep track of the price of cheapest petrol available nearby each station. As we traverse the n stations from s_0 to s_{n-1}, by greedily picking the cheapest petrol available and only making the purchase needed to reach the current location, we will end up with the cheapest total cost when we successfully reach the last station.

The complexity of this solution is $O(n \log n)$, which is no longer dependent on L.

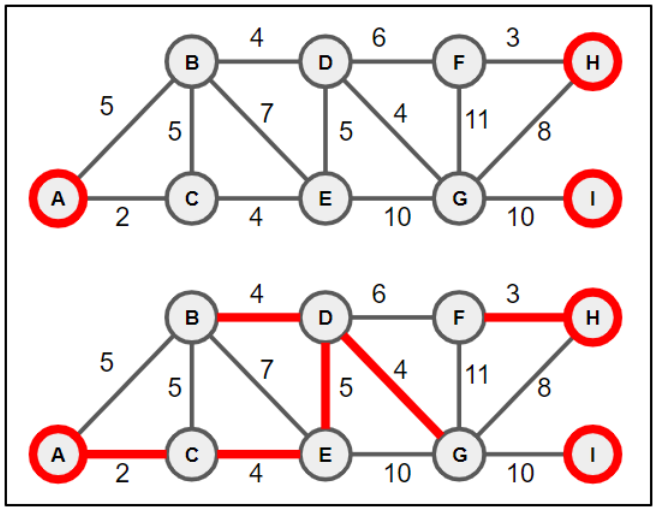
You can further optimize this idea by using a deque instead of a priority queue, which leads to an $O(n)$ solution.

This technique is called Sweep Line, which is commonly used in competitive programming.

You may refer to `roadtrip_sl.java` for a Java implementation of this greedy solution for Kattis problem Road Trip.

Problem 4. Power Plant

Given a network of power plants, houses and potential cables to be laid, along with its associated cost, find the cheapest way to connect every house to at least one power plant. The connections can be routed through other houses if required.



In the diagram above, the top graph shows the initial layout of the power plants (modelled as red nodes), houses (modelled as gray nodes) and cables (modelled as bidirectional edges with its associated cost). The highlighted edges shown in the bottom graph shows an optimal way to connect every house to at least one power plant. Come up with an algorithm to find the minimum required cost. You may assume that there exists at least one way to do so.

Solution: A way to solve this problem is to run Prim's algorithm with the priority queue initialized to contain all of the power plant nodes. This initial setup ensures that every node discovered later can be traced back to exactly one of the power plant nodes.

You can also run Kruskal's or Boruvka's algorithm with all the power plant nodes initialised to be in the same component.

Alternatively, you can insert an additional super node into the graph that is connected to each of the power plant nodes via edges of weight 0, then run any MST algorithm.

All of these approaches will run in $O(E \log V)$, where E is the number of potential cables and V is the sum of the number of power plants and houses.

Problem 5. Traffic Reduction

In order to reduce traffics, the Singapore government has passed a new law - cars are not allowed to go in circles. If you drive around in a circle, you will be charged a fine. Your job is to deploy a set of cameras to monitor the roads and catch drivers that are violating the new law.

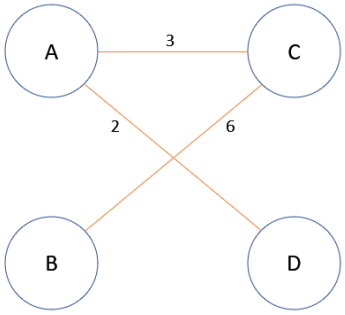
For this problem, you are given a road map of Singapore as a connected, undirected graph $G = (V, E)$. For each road (i.e., edge) e, you are given the associated cost (i.e., weight) $w(e)$ of building a camera station that can detect when the same car passes twice. Your job is to find the cheapest set of roads (i.e., edges) to deploy a camera on such that you can detect any car that makes a circle.

Solution: Find a maximal spanning tree (MaxST). All the edges that are *not* in the MaxST are designated for deploying cameras. Given a spanning tree T, any additional edge will form a cycle. So if you deploy a camera on each non-MaxST edge, you will catch anyone driving around in a circle. This is the cheapest such set of edges, since in a MaxST, for each cycle, the lightest edge is a *red* edge that is not included in the MaxST.

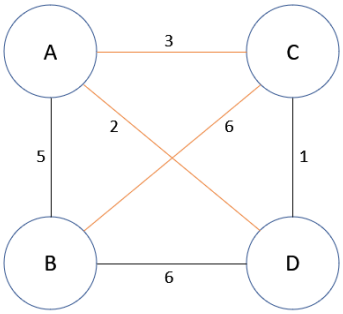
Problem 6. Let's play a game *(Just for fun)*

Now that we've reached the end of the semester, let's have some fun with an adaptation of the Stackelberg MST Game. You are now playing this game with the prof, and he is here to assess whether you have mastered CS2040S, here's how the game works:

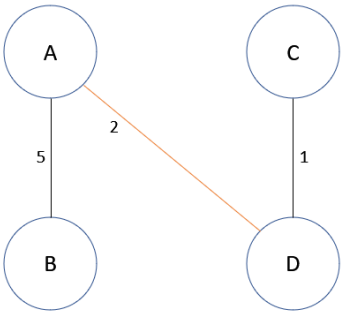
1. Prof draws an undirected weighted connected graph G_1 with $E = \theta(V)$. All edges that Player 1 draws will be **orange** edges.
2. You add any number of weighted edges to G_1 to create G_2. Adding duplicate edges is not allowed. All edges that you draw will be **black** edges.
3. Prof then selects an MST of graph G_2. This MST is M_1.
4. Your score is the sum of weights of the **black** edges in M_1.
5. Swap roles and repeat Step 2, 3, and 4 using the same graph G_1.
6. Whoever gets the higher score wins! (If it's a tie, you win!)



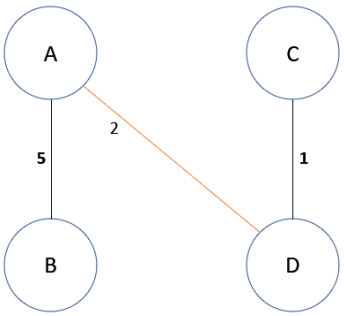
Player 1 constructs graph G_1



Player 2 draws additional edges to construct G_2



Player 1 selects an MST of G_2,which is M_1



Player 2 scores $5 + 1 = 6$

How can you win prof and show him that you have mastered CS2040S?

Solution: We want to maximise the total weight of all black edges in the MST. One way to find a good solution is to use the cut property of MSTs. For example, in the example shown, if we want to draw edge AB, we consider the cuts between $[A]$, $[B, C, D]$ and $[B]$, $[A, C, D]$. In the former cut, we see that to make our black edge favourable over the existing orange edges with weights 2 and 3, we know that our black edge's weight can be at most 1. For the latter cut, our black edge's weight can be at most 5. We then take the maximum of these 2 values and set AB to have a weight of 5.

We apply this similarly for CD and BD to draw the other weighted black edges.

However, do note that this solution is just a heuristic and the actual optimal solution is NP-hard!

Problem 7. Espionage (optional)

You are a spy, currently on a mission to obtain intel on the final examination that will be unleashed upon this world by the dreaded *Htes Treblig* within the next 25 days. At the moment, you have a message to send to HQ in the form of a **weighted tree** of N vertices, where all of the edge weights are positive integers.

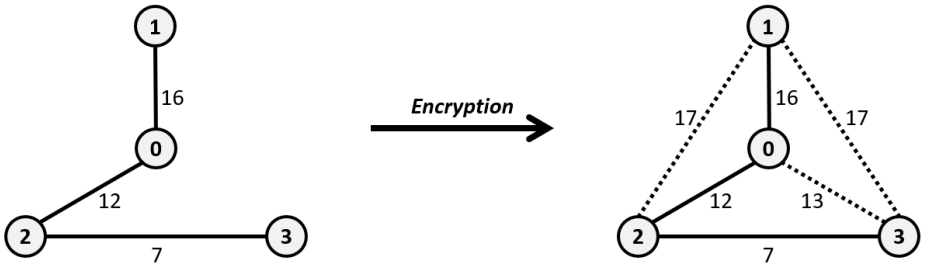
To ensure your message is not at risk of being intercepted, you need to encrypt your message. The way you decide to do this is by inserting additional edges (with integer weights) to your tree. The resulting graph, which represents the encrypted message, must satisfy the following properties:

- The resulting graph is a **complete graph** of N vertices (i.e. every pair of vertices in the graph must have exactly 1 edge between them).
- The initial tree is **the unique Minimum Spanning Tree** of the resulting graph.

Inserting edges with large weights will make the encryption process more complicated, so you would like to avoid them wherever possible. The cost of the encryption process is defined as the sum of the weights of the additional edges that are inserted into the graph during the encryption process.

Given the initial message, which consists of the number of vertices in the tree, N, and a list of $N-1$ edges of the tree, **output the minimum possible cost** of encrypting this message.

For example, let the initial message be the tree on the left, consisting of $N = 4$ vertices. The graph on the right represents the optimal way to encrypt the message with a total cost of $17+17+13 = 47$. Therefore, the expected output is 47.



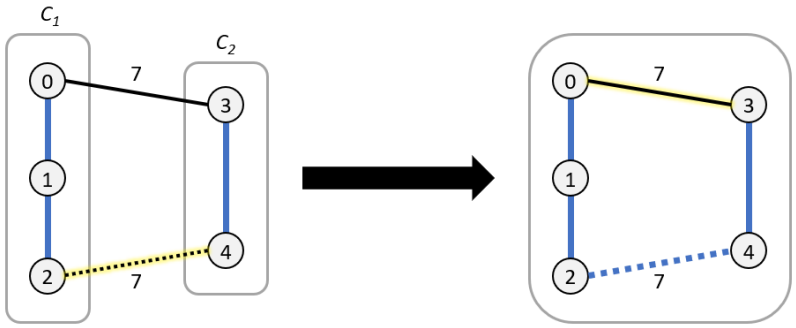
Initial Graph *Encrypted Graph*

Solution: The way to approach this problem is to take an MST algorithm and think about how to force it to select the edges from the initial tree to form the MST. Here, we will use Kruskal's algorithm. We need to ensure that Kruskal's will always select the edges from the initial tree and ignore all the new edges inserted during encryption.

Focus on a single edge e of weight w in the initial tree. When Kruskal's algorithm selects e to be in the MST, it will union two disjoint connected components. Let these two components be called C_1 and C_2, containing S_1 and S_2 vertices respectively.

We know that among all the edges in the initial tree, e is the only edge connecting C_1 and C_2 *(otherwise there would be a cycle in the initial tree)*. This means that there are $(S_1 \times S_2 - 1)$ edges to be inserted between C_1 and C_2.

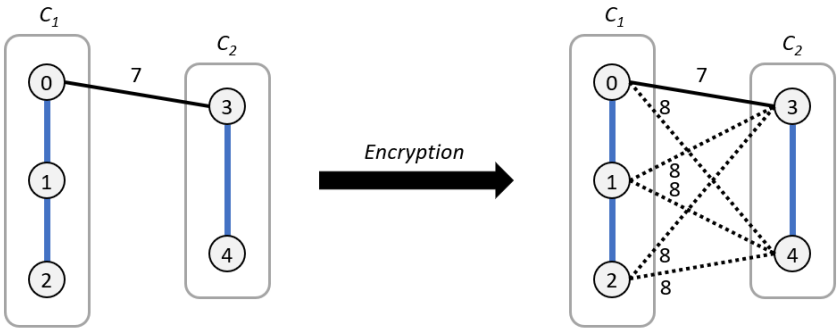
Notice that none of these new edges can have a weight $\leq w$. Otherwise, it is possible for Kruskal's algorithm to process one of the new edges before e. If that happens, then that edge will be selected as part of the MST instead of e.



Processing edge (2—4)
2 and 4 are in different components.
Add edge (2—4) to the MST.

Processing edge (0—3)
0 and 3 are in the same components.
Ignore edge (0—3)

On the other hand, as long as the new edges have a weight $> w$, then Kruskal's algorithm will definitely process these edges after processing e, where all the new edges will be ignored since they are already in the same connected component. Therefore, to minimize the total sum of edge weights, we should assign a weight of $(w + 1)$ to all $(S_1 \times S_2 - 1)$ additional edges.



Encryption cost of edges between C_1 and C_2 = $(3 \times 2 - 1) \times (7 + 1)$

Solution: (Continued from previous page)
This gives us the following algorithm:

1. Initialize a variable $cost = 0$.
2. Perform Kruskal's Algorithm using the edges of the initial tree as the edge list.
   (a) Let the current edge being processed be (u, v) with weight w.
   (b) Obtain the sizes of the sets containing u and v respectively in the Union-Find Disjoint Set. Let these sizes be S_u and S_v respectively.
   (c) Add $(S_u \times S_v - 1) \times (w + 1)$ to $cost$.
3. Output $cost$.

With N vertices and $(N - 1)$ edges in the initial tree, this algorithm runs in $O(E \log V) = O(N \log N)$ time.

- This is it for CS2040S this semester. All the best! -