CS218 Assignment 1

190050004 - Adarsh Raj
190050041 - Gudipaty Aniket
190050102 - Sahasra Ranjan
190050104 - Sambit Behera

May 11, 2021

Exercise 1

a) **Incorrect**. Counterexample: Consider intervals (1, 11), (2, 4), and (6, 8).
The proposed greedy strategy doesn't give the optimal solution in this case.
Solution according to this algorithm: $\{(1, 11)\}$
Optimal Solution:$\{(2, 4), (6, 8)\}$

b) **Incorrect**. The same counterexample given in 1.(a) will prove the proposed strategy wrong.
Solution according to this algorithm:$\{(1, 11)\}$
Optimal Solution:$\{(2, 4), (6, 8)\}$

c) **Correct**. Chose the course x that starts last, discard all classes that conflict with x, and recurse. This is similar to the `GreedySchedule` algorithm. One can reverse the starting and finishing time and the given approach will become same as the approach of `GreedySchedule` algorithm, which was proved in class. (Taken as a reference here)

d) **Incorrect**. Counter Example : (1,5), (6,10), (4,6)
Where the integers denote time and for each pair, first integer gives start time of a job and second integer gives end time.
Solution according to this algorithm:$\{(4, 6)\}$
Optimal Solution:$\{(1, 5), (6, 10)\}$

e) **Incorrect**. Counterexample : $a = (1, 10)$, $b = (4, 15)$, $c = (5, 15)$, $d = (6, 15)$, $e = (11, 20)$, $f = (15, 25)$, $g = (21, 30)$, $h = (26, 35)$, $i = (27, 35)$, $j = (28, 35)$, $k = (31, 40)$
In this case, the proposed strategy gives us a non-optimal solution of 3 intervals (f, a, k), whereas the optimal solution consists of 4 intervals (a, e, g, k).

f) **Incorrect** Counterexample : Consider the intervals (1, 10), (9, 13), (11, 15)
In this case the proposed strategy gives the wrong non-optimal answer
Solution according to this algorithm:$\{(11, 15)\}$ or $\{(9, 13)\}$
Optimal Solution:$\{(1, 10), (11, 15)\}$

g) **Incorrect** The same counterexample as given in (e) gives a non-optimal answer here, with the proposed strategy.
Solution according to this algorithm: (b, f, h)
Optimal Solution:(a, e, g, k)

h) **Correct** Here, we can see that given a choice, we always chose the interval with the earliest finish time, which is a correct strategy (as proved in the class). Now, consider x, and y as defined in the strategy. Now, we can show that if there is an interval z that has earlier end time than x and y, then x and y completely contain z. Now, consider the three cases as shown in the proposed strategy, in the same order:

- If x and y are completely disjoint, then x has the earliest ending time among all

the intervals, and it is selected, and no other interval can intersect with x, so we recurse on the set of all intervals excluding x.

- If x completely contains y, then y has an earlier end time than x, and since x intersects with y, we discard x. Now, y is the new x. Note that as we discussed before, if an interval z has an earlier end time than y, then z is completely contained in y, and hence at some point of the recursion, z will be taken into consideration (z can immediately become y if no other interval with an earlier start time than z exists, or it will come into consideration by applying the process described in this case itself multiple times), and will be selected if it turns out to have the earliest end time.
- In this case, x and y only partially overlap. So, x has an earlier end time than that of y. So, we can discard y, as it overlaps with x and any other interval z which has an earlier end time than x (but a later start time than x and y). So, the next interval which is assigned to y has an earlier end time than y (and possibly x as well), and y, which overlaps with it and any other interval with an earlier end time, has been eliminated.

So, in all the three cases, given a set of intervals, we eventually choose the interval with the earliest end time, and in the process, discard exactly those intervals which overlap with it; and the process continues. Hence, this strategy is correct, as it is equivalent to the `GreedySchedule` algorithm, whose correctness was proved in class.

i) **Correct** Again, we can show that this strategy is equivalent to the `GreedySchedule` algorithm, whose correctness was proved in the class.
When an interval x completely contains another interval y, then y has an end time not later than that of x, and the number of intervals overlapping with y is not greater than the number of intervals overlapping with x. So, we can safely discard x, because even if x exists in some optimal solution $\mathcal{OPT}$, then $(\mathcal{OPT} - \{x\}) \cup \{y\}$ is also an optimal solution that doesn't contain x.
After doing the above process, we have intervals in which every pair of intervals is either disjoint or they overlap only partially. In that case, it is easy to see that an interval which has the latest end time also has the latest start time. Hence, this strategy is also equivalent to the `GreedySchedule` algorithm, as shown in part (c) of this exercise.

Exercise 2

a) **Incorrect**. Consider $t_{1,1} = 3$, $t_{1,2} = 2$; $t_{2,1} = 1$, $t_{2,2} = 4$. Then the proposed strategy doesn't give the optimal answer (which is (2, 1), i.e. second answer sheet corrected earlier than first).
Time taken by algorithm: $t_{1,1} + t_{2,1} + t_{2,2} = 8$
Optimal time taken: $t_{2,1} + t_{1,1} + t_{1,2} = 6$

b) **Correct**
Let OPT be the set of ordering of sheets given by optimal solution.
Let A be the set of ordering of sheets given by the following greedy approach.

Approach: Decreasing order of $t_{i,2}$ (i.e. Time taken by teacher for ith sheet.)

Suppose for two consecutive sheets to be checked i.e k and $k + 1$.

- If $t_{k,2} < t_{k+1,2}$ is in optimal solution OPT, then it can be proved that on swapping the above pair, the total time for completing all the tasks does not worsen.

  Before Swap of the above pair, the end times are (for some constant c denoting the starting time of k, which doesn't depend on the parameters of sheet number k or $k + 1$):
  $c + t_{k,1} + t_{k,2}$ and $c + t_{k,1} + t_{k+1,1} + t_{k+1,2}$

  After swap of the above pair, the end times are:
  $c + t_{k+1,1} + t_{k+1,2}$ and $c + t_{k+1,1} + t_{k,1} + t_{k,2}$

  As $t_{k,2} < t_{k+1,2}$, both above times are less than $c + t_{k,1} + t_{k+1,1} + t_{k+1,2}$. Hence it can be seen that swapping the above pair does not worsen the total end time of these two jobs. Note that the end times of the remaining jobs remain unaffected by this single swap. Hence, the total time required to complete all the jobs either decreases or remains the same. But since OPT is the optimal solution, either the total time remains the same, or the solution OPT is already sorted in non-increasing order with respect to $t_{i,2}$ s.

  Now, if we keep on finding such ks and repeat the above procedure of swapping, we will eventually end up with a sorted form of OPT, which is sorted in non-increasing order with respect to $t_{i,2}$ s. Let's call it A', which is also optimal. Now the elements are sorted in A as well. So, if the positions of some elements in A' don't match with those in A, we can simply keep on swapping adjacent elements with same values of $t_{i,2}$ to get A from A', and as shown in the above argument, we can easily show that the end time for any two tasks in consideration doesn't increase, and hence the total time also doesn't increase. Hence, **A is an optimal solution**.

So, the proposed strategy gives an optimal solution.

c) **Incorrect** Consider $t_{1,1} = 10$, $t_{1,2} = 1$; $t_{2,1} = 2$, $t_{2,2} = 5$. Then, the proposed strategy doesn't give an optimal solution (which is (2, 1)).
Time taken by algorithm: $t_{1,1} + t_{2,1} + t_{2,2} = 17$
Optimal time taken: $t_{2,1} + t_{1,1} + t_{1,2} = 13$

d) **Incorrect**. Consider $t_{1,1} = 1$, $t_{1,2} = 1$; $t_{2,1} = 2$, $t_{2,2} = 2$. The the proposed strategy doesn't give the optimal answer (which is (2, 1)).

Time taken by algorithm: $t_{1,1} + t_{2,1} + t_{2,2} = 5$
Optimal time taken: $t_{2,1} + t_{1,1} + t_{1,2} = 4$

e) **Incorrect**. If we take the same example as in (d), we get a non-optimal answer (while optimal is $(2, 1)$).
Time taken by algorithm: $t_{1,1} + t_{2,1} + t_{2,2} = 5$
Optimal time taken: $t_{2,1} + t_{1,1} + t_{1,2} = 4$

Exercise 3

We have n tasks where i-th task has an execution time t_i and deadline d_i. Also there is a precedence constraints amongst the tasks, specified by a directed acyclic graph having the tasks as vertices.

The proposed algorithm for this problem is given below:

Algorithm 1: Determine if tasks can be completed before their respective deadlines

Result: True if possible to complete the jobs before deadlines, false otherwise
G ⟵ The given precedence graph
G' ⟵ Graph with reversed edge as the given precedence graph
A ⟵ Array with tasks sorted according to d_i in ascending order
t ⟵ 0 // Current time
for *incomplete task e in array A* **do**
    /*A is traversed from left to right Traverse the sub-tree in graph G' (containing only incomplete tasks as nodes) with e as the root node*/
    /* Iterate over all the incomplete tasks in it's sub-tree with the DFS method (order doesn't matter, but can do normal DFS traversal in linear time)*/
    for *each task i in e's subtree* **do**
        if $t + t_i > d_e$ **then**
            return false
        else
            Mark the task as completed;
            $t \longleftarrow t + t_i$
        end
    end
    // We have completed all the prerequisites before starting the task e
    if $t > d_e - t_e$ **then**
        // we cannot finish the task e
        return false
    else
        Continue
    end
end
return true

Outline:

1. We have to finish the task with earliest deadline first, trivially true.
2. Also for tasks with same deadline, there won't be any difference in the algorithm. We can take any of them first.
3. We will generate a new reversed graph and for any vertex, we must have to complete all the tasks in its sub-tree.
    - According to the original graph, we can say that they are the ancestors of the task e.
4. To complete the ancestors, there is no need to decide the order of DFS traversal when multiple choices are available, because all the durations are being added up, and if deadline of an ancestor is breached, then the deadline of the root-node of this DFS tree

(which in the actual graph is the descendant of these ancestors) is also breached, and detected by the algorithm.

Correctness of the algorithm:

- If the above algorithm returns **true**, then we can execute the tasks by first obtaining the **"min-element"** (the task e with the earliest deadline, found by the algorithm). Then, we find it's ancestors using the DFS tree and execute them in any valid topological ordering (according to the original dependency sub-graph corresponding to this DFS tree in G), and then execute the min-element. We mark these nodes as "visited" and tasks as "complete". Then we go to the next min-element found by our algorithm and repeat the same procedure.
  The above procedure is correct because here every task is being executed only after it's prerequisites have been satisfied and we can also show that every task has been completed before it's deadline (we prove this by contradiction). If the converse was true, a task x would have breached it's deadline, then let e be the min-element whose DFS tree x belonged to. Since $d_x \geq d_e$, the deadline of e would also have been breached (as we have been adding up the durations of all the tasks present in the DFS tree for calculating the current time t when we reach e, and this would be not before the current time at x during execution). But since our algorithm returned true, this is a contradiction.
  Hence, if our algorithm returns true, the tasks can be executed in an acceptable order, without breaching any deadlines.
- Now, if our algorithm returns **false**, then we claim that there is no acceptable order in which the given tasks can be performed without breaching a deadline.

  - There must be some min-element e, where the current time must have crossed the deadline of e in our algorithm, when our algorithm returned false.
  - Now, we assume that the tasks we executed using the procedure described above (using topological orderings). Now, note that using any other topological ordering (effectively changing the order of execution for the pre-requisite tasks) for the DFS tree of e or any other min-element won't help, because the total time taken before coming to e remains the same.
  - Executing e later (i.e. swapping e and if needed, some of it's pre-requisites with some later tasks which weren't visited by our algorithm before returning false) won't help, as the total time taken before executing e would have only increased because of the additional tasks in-between, and hence, the deadline of e would still have been breached. So, we want to see if it's possible to execute e earlier than when the algorithm visited it.
  - If e is the first min-element that the algorithm has visited, then there's nothing we can do to execute it earlier, as all the tasks executed before it are its pre-requisites.
  - Suppose e is not the first min-element found by our algorithm. Let e' be a min-element found before e by the algorithm.Then if we execute e (and hence its pre-requisites) before e', then let the latest min-element executed in the place of e be e'' (e'' is the min-element that our algorithm discovered just before e. Then since $d_{e''} \leq d_e$ and the total time taken to reach e'' is same as was taken before to reach e, now the deadline of e'' has been breached. This is true for any e'.

- Note that in the above point, we are only considering the tasks that we executed before e. If we would have considered bringing in the later tasks as well, the the time taken to reach e'' would have only increased, and hence, it's deadline would have been breached again.

  Hence, if our algorithm returns false, then there is no acceptable way in which the given tasks can be executed without breaching some deadline.

Hence, our algorithm can correctly determine whether a given set of tasks can be executed in an acceptable manner, without breaching any deadline.

Time-complexity analysis:

- Reversing the graph will take linear time
- Sorting the tasks in ascending according to their deadlines will take time $\mathcal{O}(n \log n)$
- Iterating through all the tasks will take linear time $\mathcal{O}(n + E)$, where E is the number of edges in the given DAG.
  - For each "incomplete" task we are doing a depth-first traversal
  - Each traversal will take linear time
- Since we are doing DFS for those task which are incomplete, we will visit all the only once.
- And since the given graph is DAG, total time complexity (polynomial) of our algorithm will be $\mathcal{O}(n + E + n \log n) = \mathcal{O}(n \log n)$

Exercise 4

We have an arbitrary undirected connected graph G, with positive weights assigned to the edges of of G.

First we give proof for **Prim's Algorithm** for finding a Minimum Spanning Tree in an arbitrary undirected connected graph. As the graph is arbitrary, the edge weights are not necessarily distinct.

Correctness Proof of Prim's Algorithm for Arbitrary Positive Weight Edges :

We will prove the correctness of Prim's algorithm by contradiction. The fact that the algorithm gives a spanning tree is obvious from the design of the algorithm itself. Now, suppose the algorithm gives a tree S which is not an MST. Let $ES = (e_0, e_1, \dots)$ be the sequence of edges computed by Prim's algorithm and let U be an MST which covers the longest possible prefix in the sequence ES among all the possible MSTs.

Let, $e_i = \{x, y\}$ be the first edge added to S that is not in U. Let W be the set of vertices immediately before e_i is selected. There must be a path x to y in U so let a, b be the first path with one endpoint (a) inside W and another (b) outside W.

Define the set of edges T = U + $\{\{x, y\}\}$ - $\{\{a, b\}\}$, and notice T is a spanning tree of graph G:

- **Case 1:** w(a,b) > w(x,y): In this case, in creating T we have added an edge that has smaller weight then the one we removed, and so $w(T) < w(U)$. However, this is impossible, since U is a minimum-weight spanning tree.
- **Case 2:** w(a, b) = w(x, y): In this case $w(T) = w(U)$, so T is also a minimum spanning tree. Furthermore, since Prim's algorithm hasn't selected edge a,b yet, that edge cannot be one of $e_1, e_2, \dots e_{i-1}$. This implies that T contains edges $e_1, e_2, \dots, e_i$, which is a longer prefix of ES than U contains. This contradicts the definition of tree U.
- **Case 3:** w(a, b) < w(x, y): In this case, since the weight of edge $\{a, b\}$ is smaller, Prim's algorithm will select $\{a, b\}$ at this step. This contradicts the definition of edge $\{x, y\}$.

Since all the possible cases lead to contradiction, our assumption (that S is not MST) is invalid. Hence, contradicted.

Subsections of the exercise

a) To find a minimum weight subset of edges to be removed, such that G becomes acyclic. For this, we can alter the weights of edges of the graph and create a new graph with same edges but different weights, by subtracting all the edge weights from (maximum weight + 1), we get a graph with the edge weights changed such that, maximum edge weight is now minimum and vice versa.

   Now, the required subset can be found by finding maximum weighted subset of edges which on removal make graph acyclic.

   A greedy approach can be the **Prim's Algorithm** to find the minimum spanning tree and then the result will be the set of edges not in tree, but originally present in graph.

   Let the weights of edges of the original graph be ($w(e_1)$, $w(e_2)$, $w(e_3)$..., $w(e_n)$), sorted in decreasing order, where $w(e_i)$ is edge weight function. Now, as the weights are sorted, $w(e_1)$ is maximum. According to algorithm specified, each edge weight is modified such

that, $\forall i$, $W_a(e_i) = w(e_1) + 1 - w(e_i)$, and hence by simple arithmetic, it can be seen that the new sequence, i.e. $(W_a(e_1), W_a(e_2), ..., W_a(e_n))$ is sorted in increasing order. Here, $W_a(e_i)$ is altered graph edge weight function.

The proposed algorithm for this problem is given below:

Algorithm 2: Finds the required subset of edges, by finding maximum spanning tree.

Result: Subset of set of edges of graph to be removed, such that G is acyclic.
$E \leftarrow$ Set of Edges of original graph
$V \leftarrow$ Set of Vertices of Graph
$T \leftarrow \phi$, $S \leftarrow \{v\}$ // v is an arbitrary vertex
while $|S| \leq n - 1$ **do**
  Let $E_s = \{(v, w) | v \in S \text{ and } w \in V \backslash S\}$
  Compute E_s
  Let $e' \leftarrow argmin_{e \in E_s}\{W_a(e)\}$ // Where $W_a(e)$ is altered graph edge weight function on e
  $S \leftarrow S \cup \{w\}$
  $T \leftarrow T \cup \{e'\}$
end
$R \leftarrow E - T$
Output R

Outline:

- Sort the edges of graph in decreasing order of their weights, and then alter the weights of all edges by subtracting them from (largest weight + 1).
- Start with any arbitrary node v.
- Loop on the size of S, and for node v, compute the set of edges such that one node is in S and other in $V \backslash S$. Compute the minimum weighted edge and then add the edge to T.
- After loop, Output the discarded edges set, i.e $R = E - T$, as the result for minimum weighted subset of edges to be removed to make G acyclic.

Correctness:
To prove the correctness of the above approach, we need to show that the new graph T does not have any cycles, T is connected and R is the minimum weighted subset of edges.

To prove these,

- By the design of the algorithm, T does not have cycles, as at each step we are adding an edge from the current S to $V \backslash S$.
- To show that R is the required subset, we first have to show that T is MST, which can be proved by the correctness of Prim's Algorithm, stated at the start of this exercise.

  Due to the design of algorithm, Minimum Spanning Tree is calculated for the graph with altered edge weight function $W_a(e_i)$.

  Now, on subtracting each edge weight again from $w(e_1) + 1$, we get, $\forall i, w(e_i) = w(e_1) + 1 - W_a(e_i)$, i.e, original edge weights, and thus the spanning tree obtained

will be maximum weighted spanning tree and the edges not included must be minimum weighted, since the sum total of the weights of all the edges is constant, irrespective of our choice of edges to exclude for the removal of all cycles.

Hence, R is the required subset of edges.

Time Complexity Analysis: $\mathcal{O}((m+n)\log n)$

First, the alteration of graphs by subtracting from largest edge weight + 1, can be done in linear time. Prim's Algorithm can be implemented using Priority Queue, similar to Dijkstra Algorithm. In Dijkstra, the key value is the length of the min-weight path from S. Here it is the cost of the smallest cost edge leading to a node. As the only change here is the update step, everything else can be done as in Dijkstra's algorithm, with time complexity $\mathcal{O}((m+n)\log n)$. Hence, The time complexity is $\mathcal{O}((m+n)\log n)$.

b) For a specified subset of vertices R in graph G, we have to find minimum weight edges connecting each node not in R to **some** node in R.
For this, a **Greedy Approach** can be **Prim's Algorithm** where the subset R is our initial subset instead of an empty subset.

The proposed algorithm for given problem is:

Algorithm 3: Finds the required subset of edges, by finding a minimum spanning tree.

Result: Subset of edges of graph, such that a path from each node in $V\backslash R$ to R
$E \longleftarrow$ Set of all edges in original graph{Implemented as Adjacency List}
$V \longleftarrow$ Set of all vertices
$R \longleftarrow$ Given subset of vertices
$S \longleftarrow \phi$
while $|R| \leq n-1$ **do**
  Let $E_R = \{(u,v) \mid u \in R \; and \; v \in V\backslash R\}$
  Compute E_R
  Let $e' \longleftarrow argmin_{e \in E_R}\{w(e)\}$
  $R \longleftarrow R \cup v$
  $S \longleftarrow S \cup e'$
end
Output S

Outline:

- Start with R as your initial subset of vertices. If R is empty, select a random vertex and insert in R.
- Loop on the size of R, and for all edges $\{v, w\}$ such that $v \in R$ and $w \in V\backslash R$, compute the minimum weighted such edge and add to S.
- Also add w to R.
- After all nodes have been added to R, output the set of edges S computed, as the result.

Time Complexity Analysis:

- Inserting all elements of $V\backslash R$ into priority queue Q (with key values equal to the smallest such edge weight for nodes with direct edges with some node in R and ∞

for others) takes $\mathcal{O}(E + VlogV)$ time.

- The While loop's running is of the order of $\mathcal{O}(V)$ and each time size of R increases by 1.
- Extracting minimum weight edge from Q is $\mathcal{O}(logV)$ each time inside the while loop and hence takes a total of $\mathcal{O}(VlogV)$ time.
- Iterating over the adjacency list of output in previous step takes a total $\mathcal{O}(E)$ time and updating key value during this step is $\mathcal{O}(logV)$ and hence results in a total $\mathcal{O}(ElogV)$ time.
- Hence we can conclude that the total running time of the algorithm is $\mathcal{O}((E + V)(logV))$.

Correctness:
To prove the correctness of our algorithm, we need to show, all vertices not in R have a path to some node in R and the weight of edges picked is minimum.

- Consider the base step of algorithm with original R. When our algorithm picks a least cost edge $\{u, v\}$, $u \in R$, $v \in V \backslash R$, it satisfies the required condition of minimum weighted edge condition, and also the fact that v has now some path to R. Inducting on this fact at any arbitrary step with R' as our expanded subset, the newest node we pick will either be connected directly to some node in R or a node in $V \backslash R$ which has a path to R and hence we can ensure that our algorithm satisfies the condition of connected-ness to R.
- The initial subset R can be shrunk to a "super-node", and can be thought of as a single node, with all edges going from R to $V \backslash R$ being the edges connected to this node. We are applying Prim's Algorithm on this new formed Graph (Proof provided at start of ex 4) to find a MST and hence our algorithm produces a minimum value subset of edges only.
- Note that our algorithm picks only edges that go from R to $V \backslash R$. Suppose the optimal solution had an edge with both vertices in R. But, all vertices in $V \backslash R$ are connected to some node in R already (as proved in the above point) and removing this edge from the optimal solution won't affect connected-ness of all the nodes in $V \backslash R$ to R and will only reduce the total cost. Hence there is no edge belonging completely to vertices in R in the optimal solution.

c) To find the second minimum weight spanning tree in G:

Observations:

- Let T be the MST of the given graph.
- The second best MST differs from T by only one edge replacement. (Proof in Correctness Section of Same Solution)
- As there is only one edge, we need to find an edge e_n which is in not in T, and replace it with an edge in T (let it be e_o) such that the new graph $T' = (T \cup \{e_n\}) \backslash \{e_o\}$ is a spanning tree.
- Also, the weight difference $(e_n - e_o)$ must be minimum, as the required result is second best MST.

Greedy Approach (Algorithm):

- Find a Minimum Spanning Tree using Kruskal's Algorithm, after sorting the edges in increasing order of your weights.
- For each edge e not already in the MST, temporarily add it to the MST, creating a cycle.
- Remove k (edge with maximal weight in the cycle that is not equal to e) temporarily, creating a new spanning tree.
- Compute the weight difference $\delta_e = w(e) - w(k)$, and remember it together with the changed edge.
- Repeat second step for all other edges, and return the spanning tree with the smallest weight difference to the MST.

Time Complexity Analysis:

- Sorting of edges is done in $\mathcal{O}(E \log E)$, and as Kruskal's Algorithm is linear, time taken is $\mathcal{O}(E)$
- The time complexity of the algorithm depends on how we compute the k_s, which are the maximum weight edges in second step of this algorithm. One way to compute them efficiently in $\mathcal{O}(E \log V)$ is to transform the problem into a Lowest Common Ancestor (LCA) problem.
- We can find the largest edge in the cycle with LCA preprocessing using vectors as data structures and binary lifting, in which we have to find ancestors of each node in binary order, i.e first, second, fourth, etc and store them in a 2d vector or array. This allows us to jump to any ancestor in array in $\mathcal{O}(\log V)$ time. The array is computed by DFS traversal of a tree. Preprocessing and all LCA queries can be done in $\mathcal{O}(E \log V)$ time.
- Therefore, the final time complexity of this approach is $\mathcal{O}(E \log V)$.

Correctness Of Algorithm:

- Correctness of Kruskal's Algorithm can be verified from slides of the course. Hence, it results in minimum spanning tree.
- It can be easily seen using the basic concepts of graph theory (number of nodes, number of edges, and absence of cycles) that the final graph obtained is also a spanning tree.
- Now, when we add an edge e to the MST (let's call it T) and it creates a cycle, then it is always true that e is the largest weight edge in that cycle. Otherwise, if some other edge e' in that cycle had a larger weight, then we can simply use the exchange argument that the spanning tree $T' := T - \{e'\} \cup \{e\}$ has a smaller total weight than T, which is a contradiction, since T is a unique MST.

  Hence, $\delta_e = w(e) - w(k)$ is a positive quantity, which is the difference in the total weights of the spanning tree $T_e := T - \{k\} \cup \{e\}$ and T . So, T_e is a candidate for second minimum weight spanning tree. The candidate T_e with lowest such δ_e will

be the second minimum spanning tree, as we will prove in the subsequent points. (Note : The second minimum spanning tree doesn't have to be unique)

Note that removing an edge with a smaller weight than $w(k)$ from the cycle would have given us a spanning tree which would have had a higher total weight than that of T_e, and hence, it can't be the second minimum weight spanning tree.

- Now, we show that only those spanning trees which differ from the MST T (which is unique) by exactly one edge are the candidates for the second minimum weight spanning tree:

  Suppose a spanning tree T' differs from T by exactly $k(> 1)$ edges. Then there are k edges in $T : e_1, e_2, \dots, e_k$, which have been replaced by edges $e'_1, e'_2, \dots, e'_n$ respectively (explained below) to obtain T'. Now, let's analyze this phenomenon.

  Now, suppose that we are replacing the edges in T one by one. Suppose we have replaced $e_1, e_2, \dots e_i$ with $e'_1, e'_2, \dots, e'_i$ in T to create an intermediate spanning tree T'_i (So, $T' = T'_k$). Now, we remove an edge e_{i+1} from T'_i. One can easily conclude that e_{i+1} is also present in T (because removing an replacing previously added edges means the the eventually T and T' will differ by less than k edges, which is a contradiction). Now, suppose e_{i+1} was connecting the components S and $V\backslash S$, where V is the set of vertices in the original graph. Now, we replace e_{i+1} with e'_{i+1}, which also connects S and $V\backslash S$.

  Now, using the **cut property**, we can say that since the edge e_{i+1} belongs to the MST T (and e'_{i+1} doesn't), and e_{i+1} is the only edge in T which connects V and $V\backslash S$, and the minimum-weighted edge between V and $V\backslash S$ must be present in T by the virtue of the cut property, e_{i+1} is the minimum weighted edge between S and $V\backslash S$. Hence, $w(e'_{i+1}) > w(e_{i+1})$. Let's denote the tree created now as T'_{i+1}. So, the total weight of T'_{i+1} is higher than that of T'_i by $\delta_i = w(e'_{i+1}) - w(e_{i+1})$.
  So, let's denote the total weight of a tree T'' by $W(T'')$. Then, using the fact that $(k > 1)$, we get (Using $W(T'_{i+1}) - W(T'_i) = \delta_i > 0$) :

  $$W(T) < W(T'_1) \dots < W(T'_k) = W(T') \quad (\text{As } T'_k = T')$$

  So, T' can't be the second minimum weight spanning tree if it differs from the MST T by more than one edge.

  Hence, the second minimum weight spanning tree differs from the minimum spanning tree by exactly one edge, and is found by the algorithm described above, which looks at every excluded edge one by one in an optimal and greedy way.

  Note : If the given graph itself is a tree, then no such second minimum spanning tree exists, which is also reflected by our greedy algorithm, as it returns nothing in that case.

4.0.1 References

(a) Binary Lifting, Least Common Ancestor - CP Algorithms

(b) Prim's Algorithm - Prof. Stephen R. Tate

(c) Depth First Search - Wikipedia