SRIRAJ
CS24BTECH11066

(1) To check whether a negative - weight cycle exists, an extension of Bellmann-Ford algorithm can be used.
After relaxing all edges V-1, relax the edges one more time, and if the shortest-path value of any vertex decreases, then that means a negative-weight cycle exists.

Psuedo code

[1] Initialise parent[V] array, e.g. if (u, v) is an edge in the graph then parent[v] = u.

[2] Initialise dist[V] to ∞, except the selected source vertex 's'. dist[s] = 0.

[3] Relax all the edges V-1 times.

[4] Relax the edges for the V^{th} time, and store the vertex, say k, for which dist[k] changed after the V^{th} relaxation.

[5] for i ← 1 to V :

K = parent [K].

This is to ensure that we get a vertex which a part of the negative-weight cycle.

[6] print(K)

```
temp = parent[K]
while (temp != K)
    print (temp)
    temp = parent[temp]
```

K now traverse along the cycle and the vertices are printed, until K is reached again.

---

② Let the vertices of the edge with decreased weight be u, v, and the new weight be w.

Find the path from u to v (using Dijkstra's algo./Bellman-Ford algo.) in the given MST 'T', and find the edge with max weight, say W, among the edges along the path.

If $W < w$, then return T

If $W > w$, then remove vv from the MST, add w to the MST and return the new MST.

---

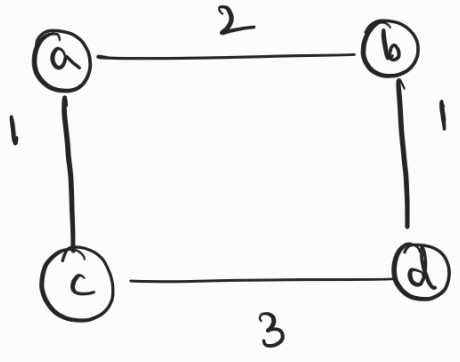
(3) MST is built by adding the lightest edge across a cut, where the MSTedges[] array starts from the lightest edge in the graph.

For each cut, if $\exists$ two different edges across the cut with the lightest weight then two different MSTs are possible, which are by choosing either one of the light edges.

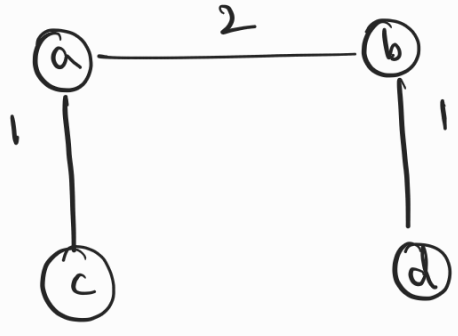
$\Rightarrow$ If for every cut, if there a unique light edge crossing the cut, then there is no dilemma of choosing the edge to include in the MST construction. So, for every cut there only a single choice of edge to include in the MST construction, thus the resulting MST is unique.

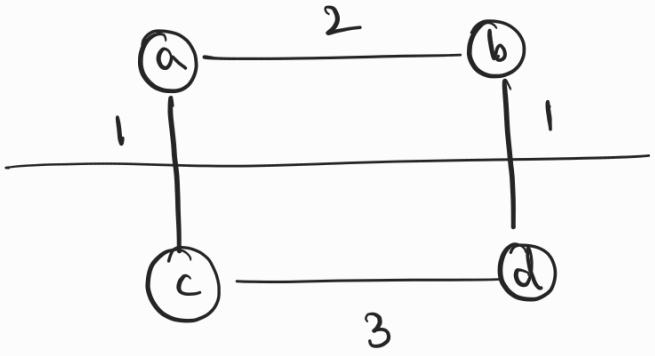
Contradicting example for the converse statement:

If a graph has a unique MST, then there should be a unique light edge across a cut. for every cut.



This graph has a unique MST.





(a, c) and (b, d) are the light edges crossing the cut i.e. the light edge is not unique.

∴ Unique MST $\not\Rightarrow$ Unique light edge across every cut.

④ Let the new vertex be 'v' and its no. of incident edges be 'k'.

Case-1: $k = 0$, i.e. v is isolated

→ New MST in $O(1)$ time

(The new MST is same as the old one)

Case-2: $k \neq 0$

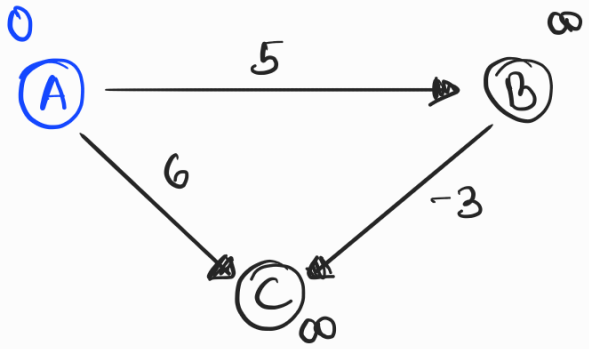
The original MST has all the original vertices connected, we just need to connect this new vertex v to the old MST and generate the new MST.

⇒ We need to find the incident edge of v with the least weight.

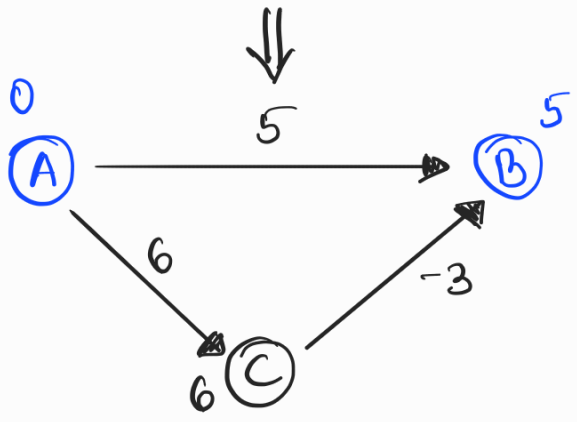
⇒ Time complexity = $O(k)$

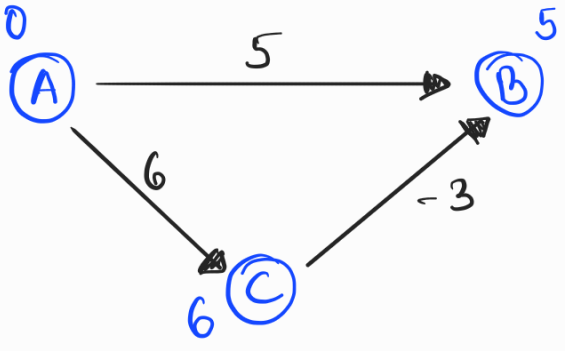
∴ Linear time in worst case.

(5)



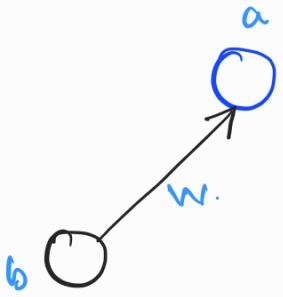
● ⇒ visited





The shortest path to (B) is (A) → (C) → (B),
but Dijkstra's algorithm gives (A) → (B).
This happened because, Dijkstra's algorithm doesn't check for an already visited vertex.
This works in case of non-negative weights, because the vertex with least value is marked visited, so any path from any other

unvisited edge will be greater than its value.



$a < b$ (∵ a is visited before b).

$a < b + w$ only if $\underline{w \geq 0}$.

So, for $w \geq 0$, there is no need to check this $(b+w)$ path, which is why Dijkstra doesn't check it.

But, in case $w < 0$, $a < b + w$ need not be true, so we need to check and then update the value which does not happen in Dijkstra's.

∴ Dijkstra's fails for negative weights.

---

⑥ $O(VW + E)$ time can be achieved by considering array of "buckets" instead of min-heap.

Bucket at index "i" stores the unvisited vertices whose current shortest-path estimate is "i".

⇒ $W(V-1)+1$ buckets are needs, as $W(V-1)$ is the maximum possible distance between two vertices.

Pseudocode:

[1] Initialise distance to source s as 0, and others as ∞.

[2] Place s in bucket 0, while the others are unbucketed.

[3] for i ← 0 to W(V-1):

```
while (bucket [i] is non-empty):
    u ← a vertex in bucket[i]
    for each outgoing edge (u,v) of wt. w:
        if (dist[v] > dist[u] + w):
            dist[v] = dist[u] + w
            place (bucket[i+w], v)
# Put v in bucket [i+w].
```

[4] Terminate when all buckets up to the largest non-empty bucket are processed.

Time complexity:

Relaxation : $O(E)$

Going through buckets: $O(W(V-1)+1) = O(VW)$

$\therefore$ Time complexity: $O(VW + E)$.

In this algorithm, instead of using a min-heap to choose the unvisited vertex with shortest-path like in Dijkstra's, we move from bucket [0] to bucket [W(V-1)] i.e. from lowest to highest, so the vertex with the shortest path-length is still the first one to be visited.

---

⑦ The best case for a heap sort is if the input array is in ascending array.

Time complexity:

① Building the heap (bottom-up) takes $\Omega(n)$ time, since each node is checked for heap property.

② Deleting the minimum each time and applying heapify takes $\Omega(n\log n)$ time (∵ heapify is called 'n' times, and each call takes 'log n' time).

$\Rightarrow T = \Omega(n) + \Omega(n\log n)$

$\therefore T = \Omega(n\log n)$ ✓

(8)

$$[BB^T]_{ij} = \sum_{e \in E} b_{ie}\, b_{je}$$

$\underline{i = j}$

$b_{ie} \cdot b_{je} = b_{ie} \cdot b_{ie}$ is either 0 or 1 ($1 \cdot 1$ or $(-1) \cdot (-1)$)

1 if e leaves/enters i

0 otherwise.

$$\Rightarrow [B \cdot B^T]_{ii} = \sum_{e \in E} b_{ie} \cdot b_{ie} = \text{degree-in} + \text{degree-out} = \text{degree of } i$$

$\underline{i \neq j}$

$$b_{ie} \cdot b_{je} = \begin{cases} (1) \cdot (-1), & \text{if } e \text{ enters } i \text{ and leaves } j \\ (-1) \cdot (1), & \text{if } e \text{ leaves } i \text{ and enters } j \\ 0, & \text{otherwise.} \end{cases}$$

$$\Rightarrow [B \cdot B^T]_{ij} = \sum_{e \in E} b_{ie} \cdot b_{je} = -(\text{no. of edges connecting } i \text{ and } j)$$

$$\therefore [B \cdot B^T]_{ij} = \begin{cases} \text{degree of } i, & \text{if } i = j \\ -(\#\text{ of edges connecting } i \text{ and } j), & \text{if } i \neq j \end{cases}$$

⑨

```
for each u in V:
    for each v ∈ adj[u]:
        addEdge(adj2[u], v)   # 1 edge away from u
        for each w ∈ adj2[v]:
            addEdge(adj2[u], w)   # 2 edges away from u
```

(adj2 ⇒ adj. list/matrix of G^2)

Adjacency list:

adj2 might have some duplicates.

Time complexity:

[1] Computing adj2:

Outer loop: V

First nested loop: E

Second nested loop: sum of out-degrees of neighbors = $O(\sum_v deg(v)^2)$

⇒ $O(V \cdot E)$ for sparse graphs, where the 3rd term is small.

[2] Ensuring no duplicates while addEdge():

We have to look through all the neigbors and then add

⇒ Time complexity = $O(E)$.

∴ Total time complexity = $O(VE) + O(E)$

$= O(VE)$ ✓

Adjacency matrix:

There can be no duplicates here.

∴ Total time complexity = $O(VE)$