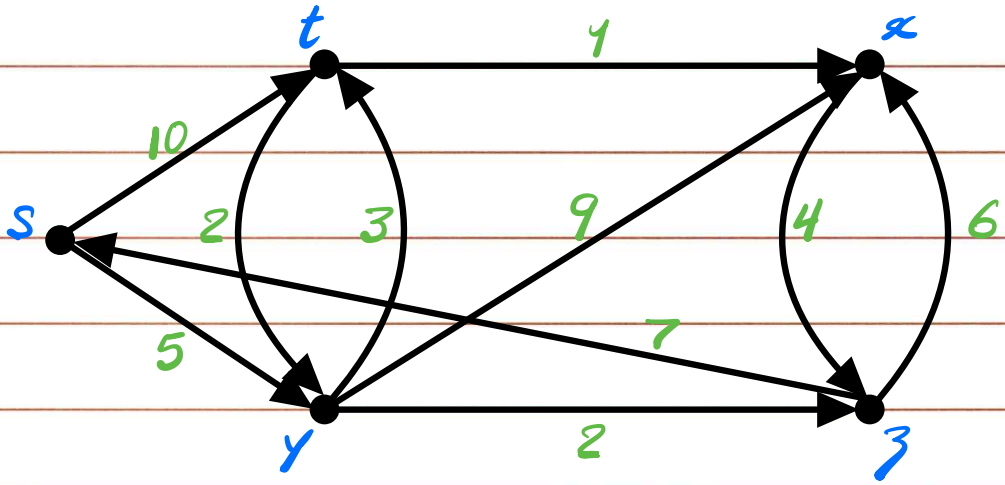
Shortest Path Problem

Problem Statement:

Given $G = (V, E)$ with $\omega(u,v) \geq 0$ for each edge $(u,v) \in E$, find the shortest path from $s \in V$ to $V - s$

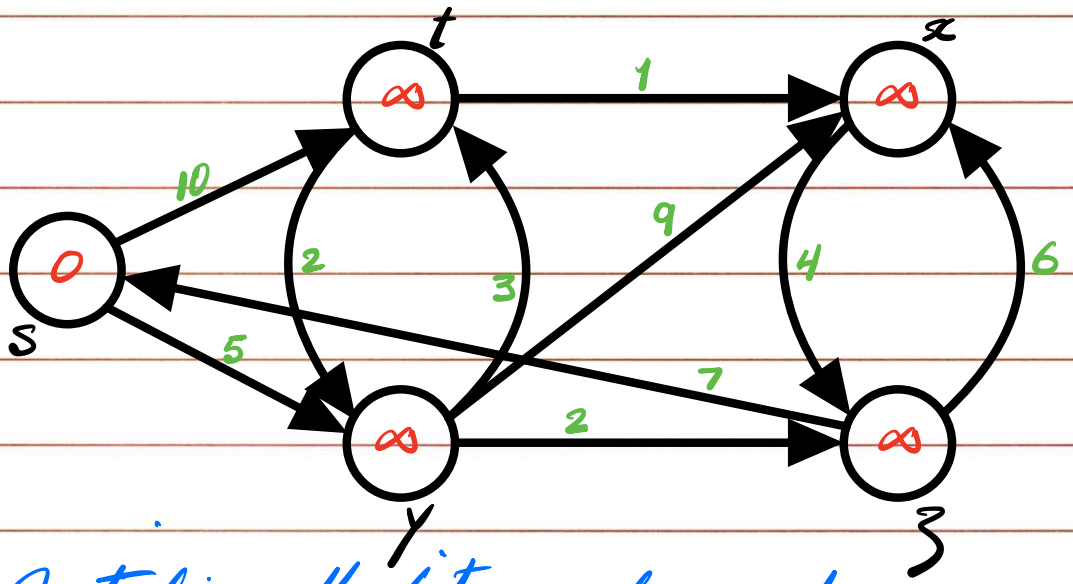


Lowest cost edge out of s will give us the shortest path to the nearest neighbor of s. How about shortest paths to other nodes?

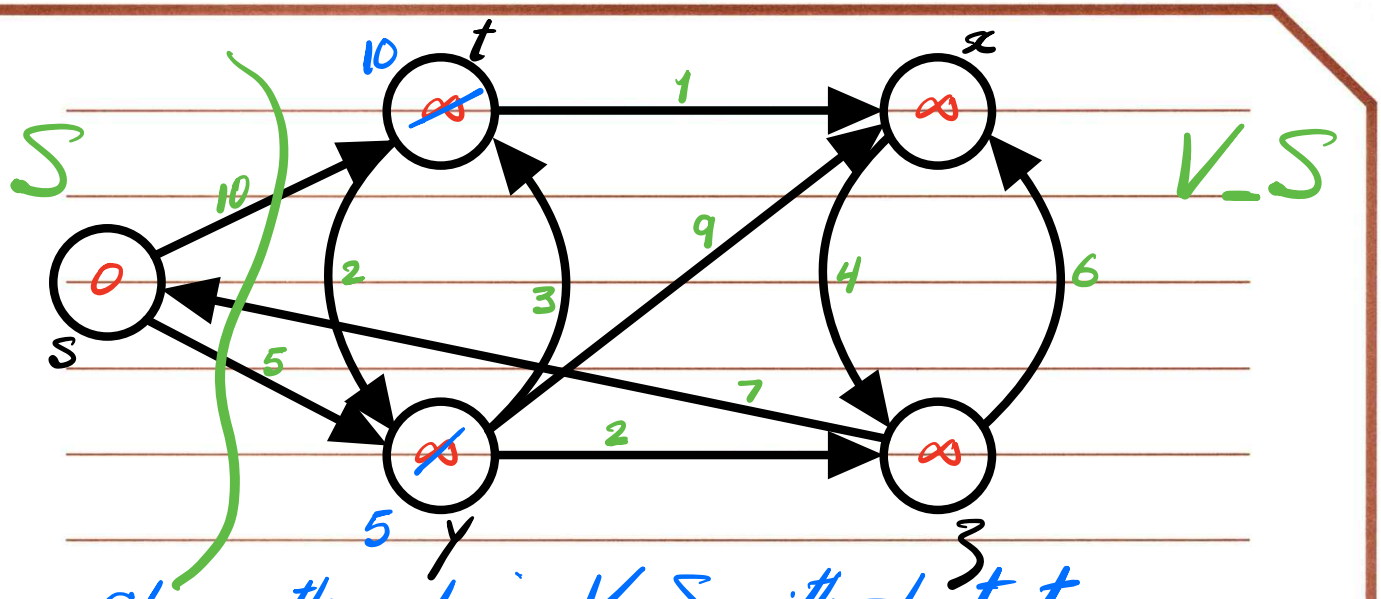
High Level Solution

1. Start with a set S of vertices whose final shortest path we already know.
2. At each step, find a vertex $v \in V - S$
3. Add v to S, and repeat.

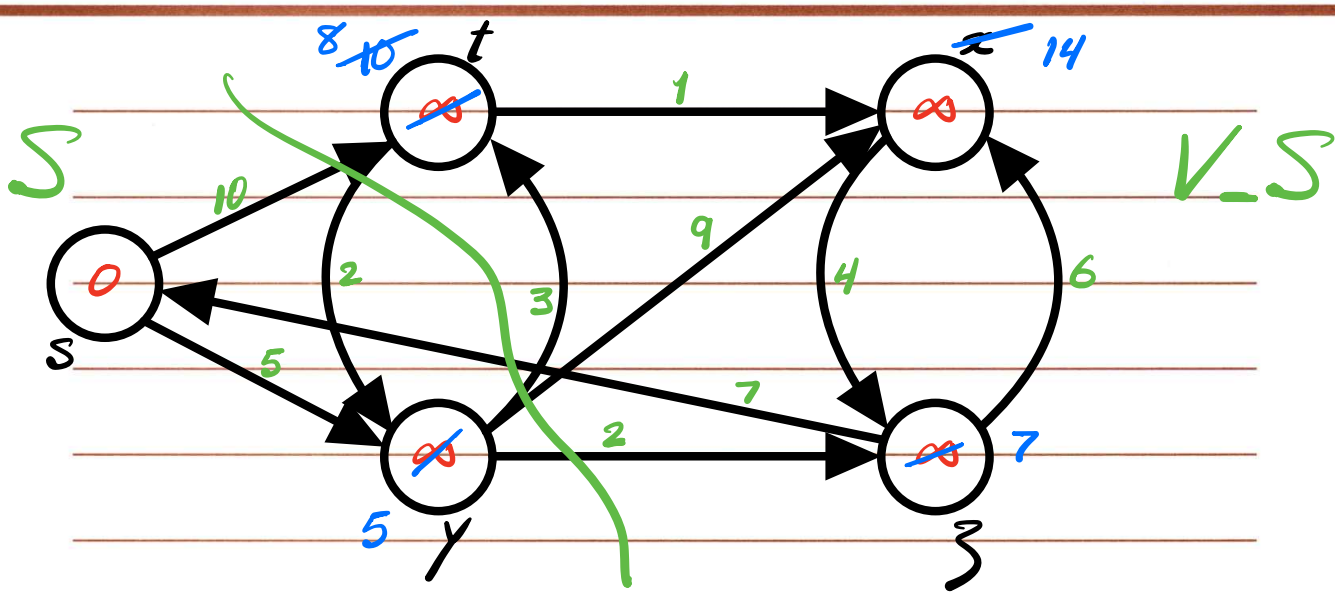
Initially all nodes are in $V - S$



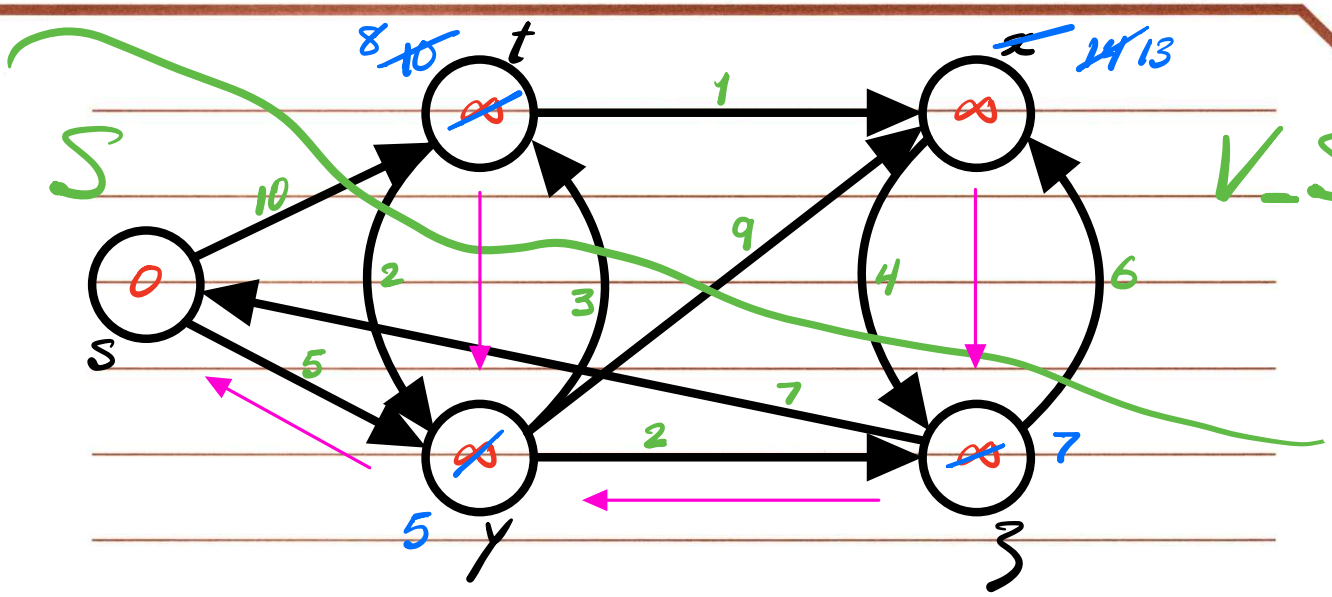
Initialize all distances from s to ∞, except for s itself where distance to itself is zero.



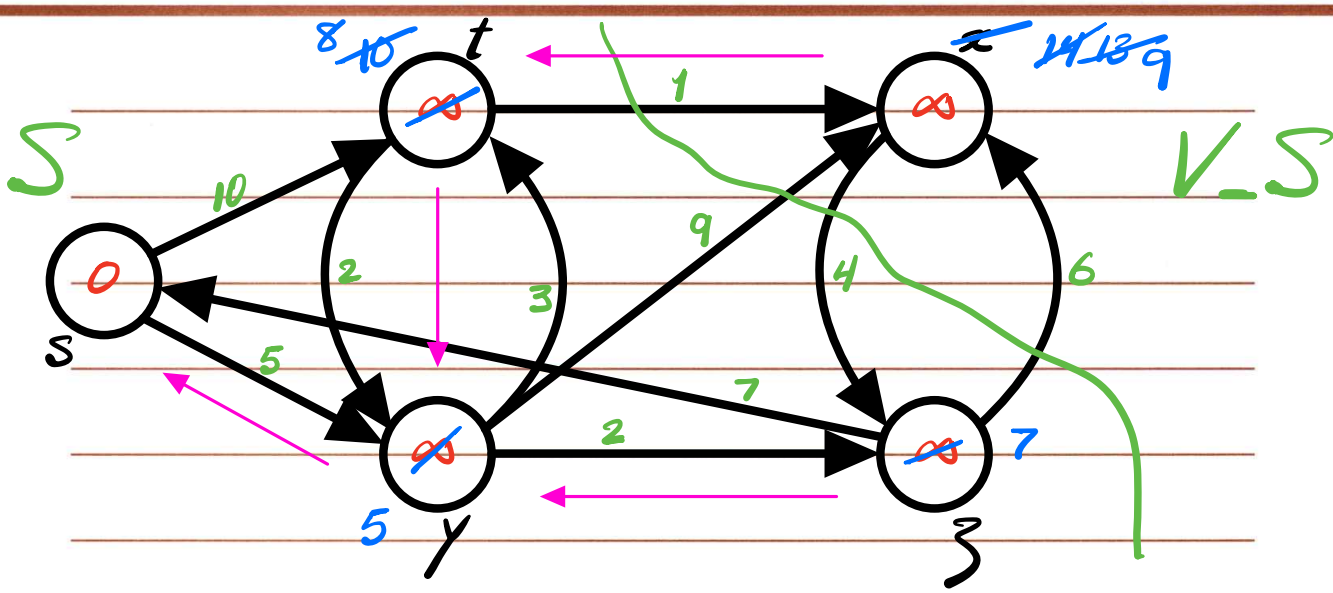
- Choose the node in V_S with shortest distance from s and move it to the set S
- Perform relaxation step: update shortest distances for neighbors of the node just moved to S.



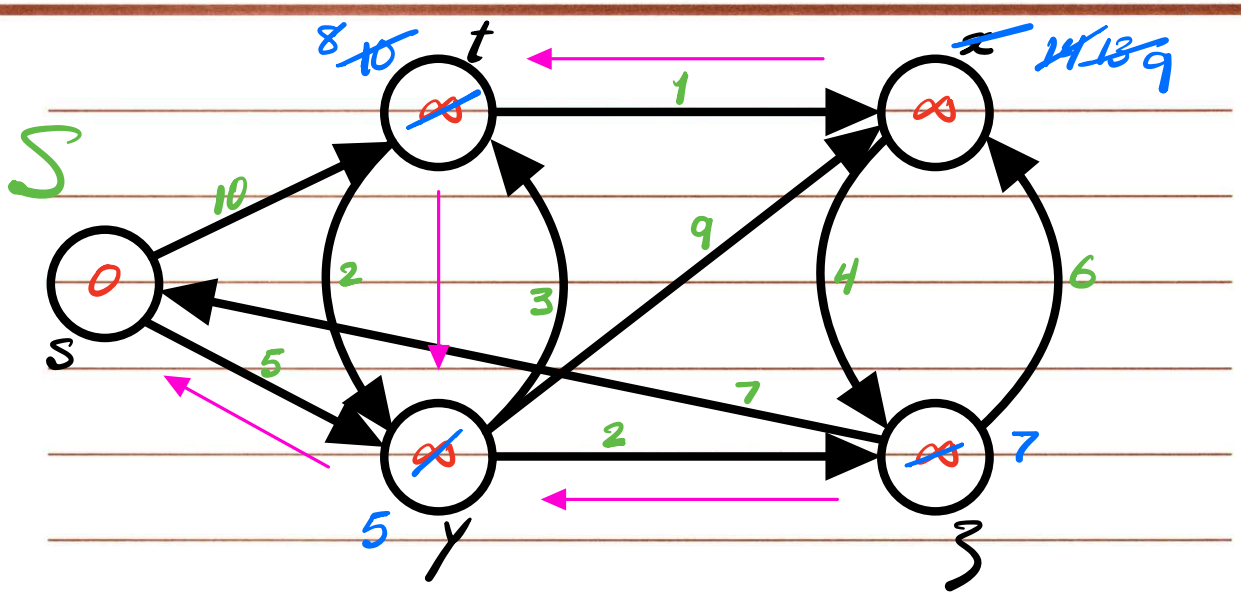
In the first iteration we placed s in the set S, in the second iteration, we have found the shortest distance to y and we will move y into the set S.



Node z is chosen in iteration #3.
Always keep a pointer back to the neighbor that gave it its shortest distance from s.



Node t is chosen at iteration 4.
Notice that x is now pointing to t as the neighbor that gives it the shortest path from s.



Finally, at iteration 5, we have found all shortest distances and shortest paths from s.

The algorithm described above is called

Dijkstra's Algorithm

Proof of Correctness

We will prove that at each step, Dijkstra's algorithm finds the shortest path to a new node in the graph.

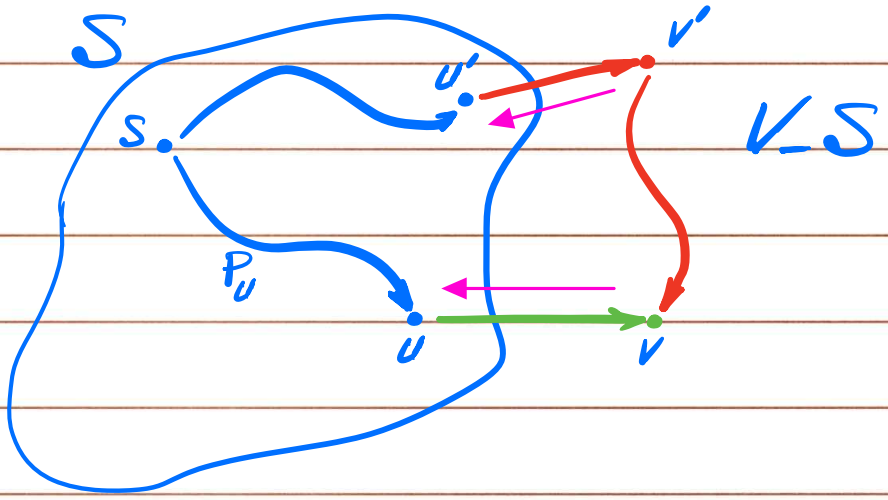
Proof using strong induction:

Base Case: $|S| = 1$, $S = \{s\}$ and $d(s) = 0$

Inductive hypothesis:
Suppose the claim holds when $|S| = k$ for some $k \geq 1$. In other words Dijkstra's has found the shortest path from s to the first k nodes in the graph

Inductive Step:

We now grow S to size k+1 and prove that we have found the shortest path to a new node.



Suppose Dijkstra's alg. chooses v at the k+1st iteration, and that u is the neighbor that gives v its shortest distance from s. We claim that $P_u + (uv)$ is the shortest path from s to v.

Let's assume there is a shorter path to v. This path must cross the boundry of S and V-S. Let (u'v') be the edge crossing the boundry on that path.

At the moment that Dijkstra's alg. picks node v, we must have:

Distance from s to v through u <
Distance from s to v' through u'.

Since there are no negative cost edges, the path from v' to v will only add to the length of that path.

Implementation of Dijkstra's

Initially S = Null, $d(s) = 0$, and
for all other nodes $d(u) = \infty$

While $S \neq V$

Select a node $v \notin S$ with at least one edge from S for which

$$d(v) = \min_{e(u,v): u \in S} (d(u) + l_e)$$

Add v to S

endwhile

What is the ideal data structure to store the set V?

A priority queue

More Detailed Implementation of Dijkstra's Alg.

$S = \text{Null}$

Initialize priority queue Q with all nodes V where $d(v)$ is the key value.
(All $d(v)$'s are set to ∞, except for s where $d(s)=0$)

While $S \neq V$

  $v = \text{Extract\_Min}(Q)$

  $S = S \cup \{v\}$

  for each vertex $u \in \text{Adj}(v)$

    if $d(u) > d(v) + l_e$

      Decrease_key$(Q, u, d(v)+l_e)$

  endfor

endwhile

(C = no of calls to Decrease_key)

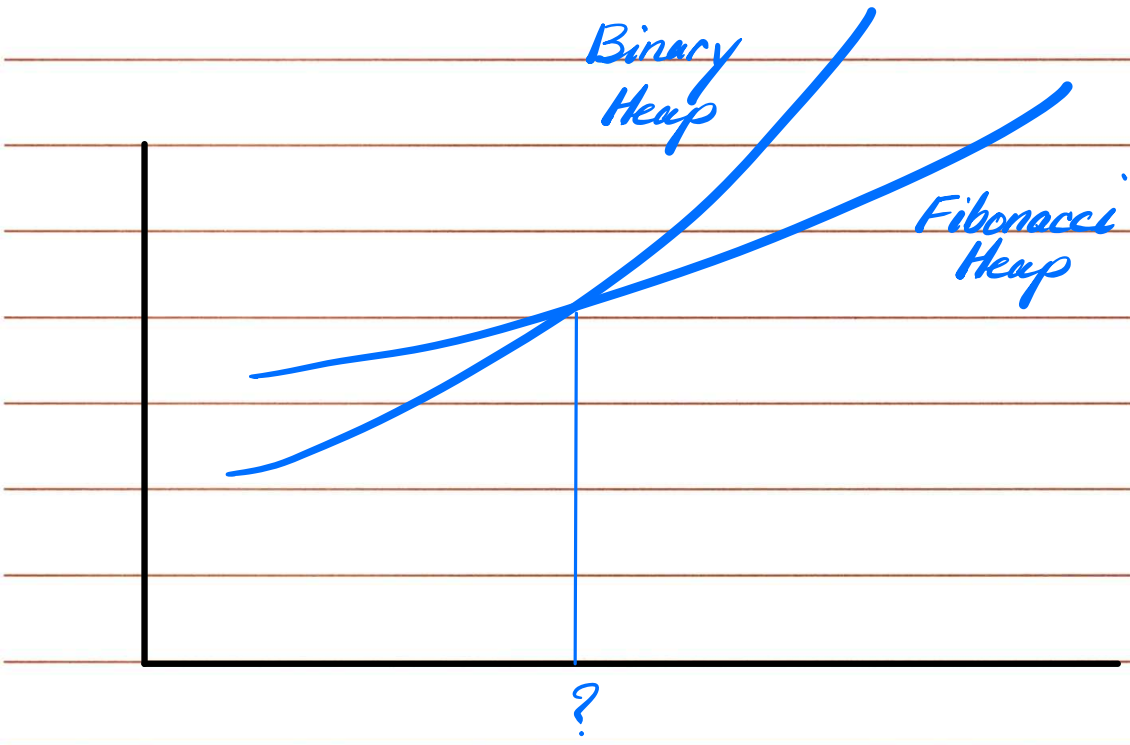
$C = \sum_{v \in V} d^+(v) = \Theta(m)$

$C = \Theta(d^+(v))$

Complexity Analysis

- Initialize / construct priority queue $O(n)$
- Max. no. of Extract-Min operations $O(n)$
- Max. no. of Decrease-key operations $O(m)$

	Binary Heap	Binomial Heap	Fibonacci Heap
n Extract-Min's	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$
m Decrease-key's	$O(m \lg n)$	$O(m \lg n)$	$O(m)$
Total	$O(m \lg n)$	$O(m \lg n)$	$O(m + n \lg n)$
Sparse graphs	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$
Dense graphs	$O(n^2 \lg n)$	$O(n^2 \lg n)$	$O(n^2)$

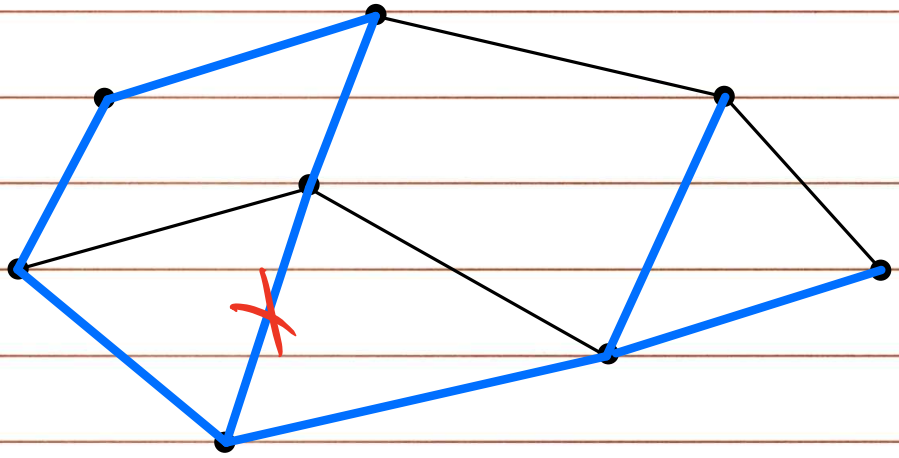


Even for dense graphs, the Fibonacci heap implementation will not outperform the binary heap implementation in practice due to:

- Higher memory requirements
- Higher constants involved

Problem Statment

Find a minimum cost network that connects all nodes in the weighted undirected graph G.



Def. Any tree that covers all nodes of a graph is called a spanning tree.

Def. A spanning tree with minimum total edge cost is a minimum spanning tree. (MST)

Problem Statement

Find a MST in an undirected graph.

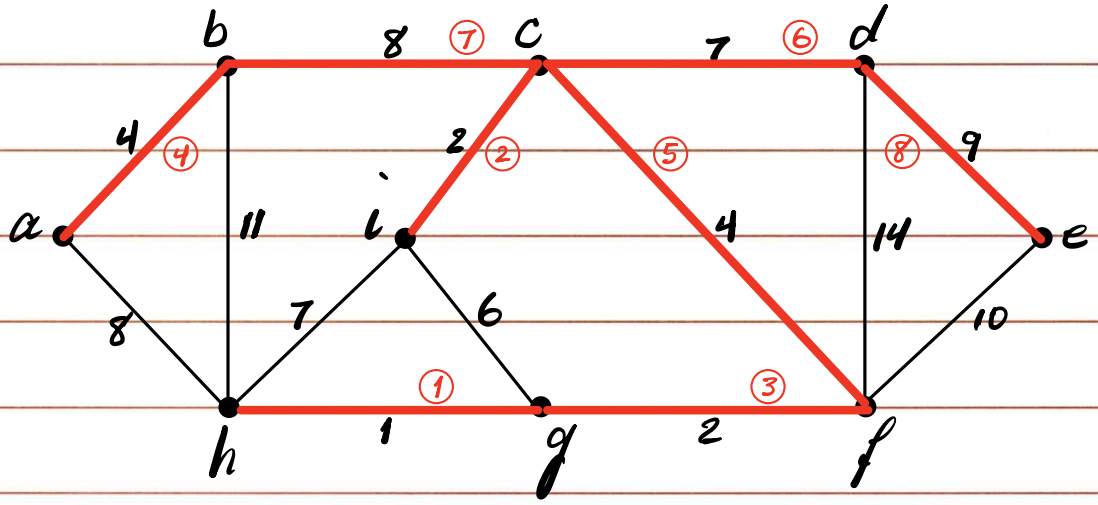
Sol. 1: Sort all edges in increasing order of cost. Add edges to T in this order as long as it does not create a cycle. If it does, discard the edge.

Kruskal's Alg.

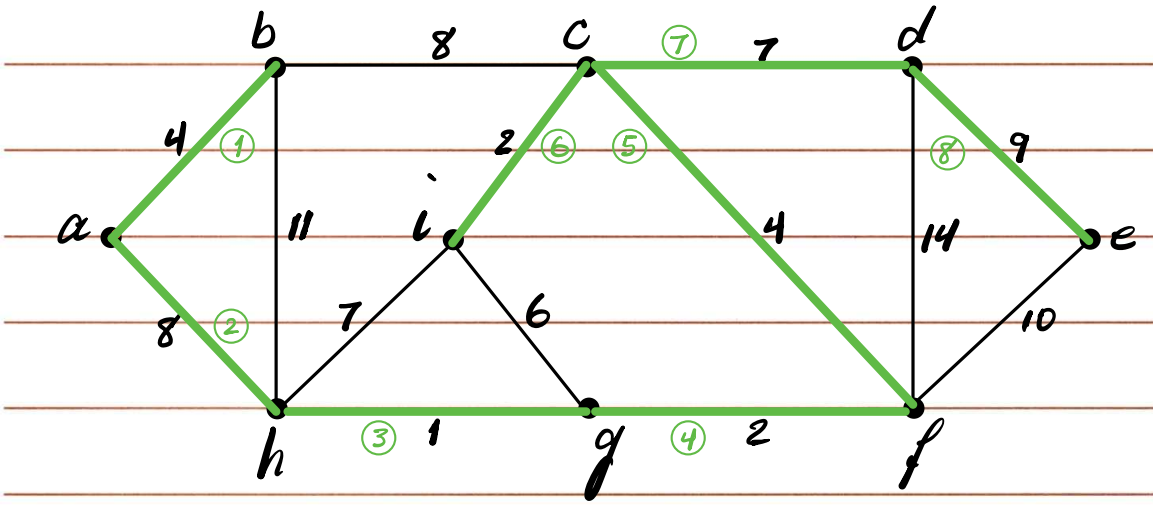
Sol. 2: Similar to Dijkstra's algorithm, start with a node set S (initially the root node) on which a minimum spanning tree has been constructed so far. At each step, grow S by one node, adding the node v that minimizes the attachment cost.

Prim's Alg.

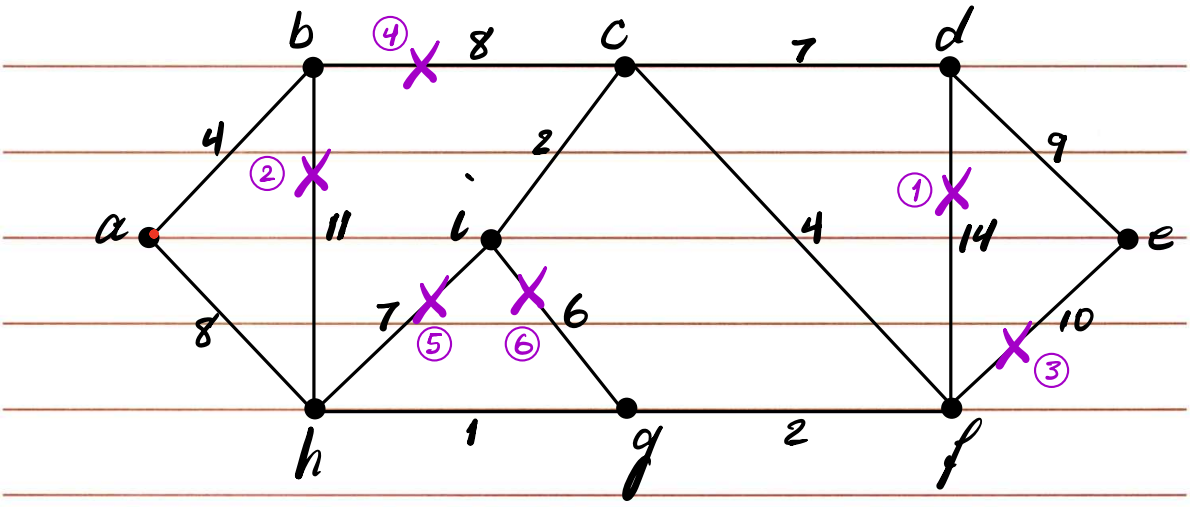
Sol. 3: Backward version of Kruskal's.
Start with a full graph.
Begin deleting edges in order of decreasing cost as long as it does not disconnect the graph.
Reverse Delete Alg.



ⓘ Marks the i^{th} edge chosen by Kruskal's alg. to be placed in the MST.

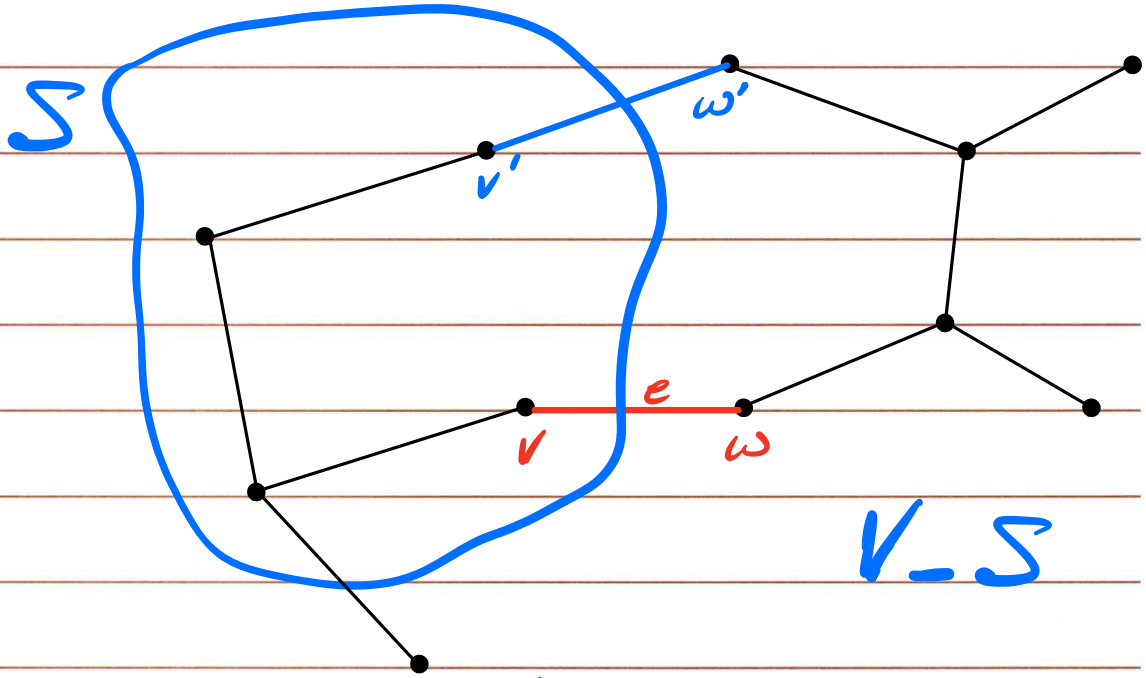


① Marks the i^{th} edge chosen by Prim's alg. to be placed in the MST (when starting from a)



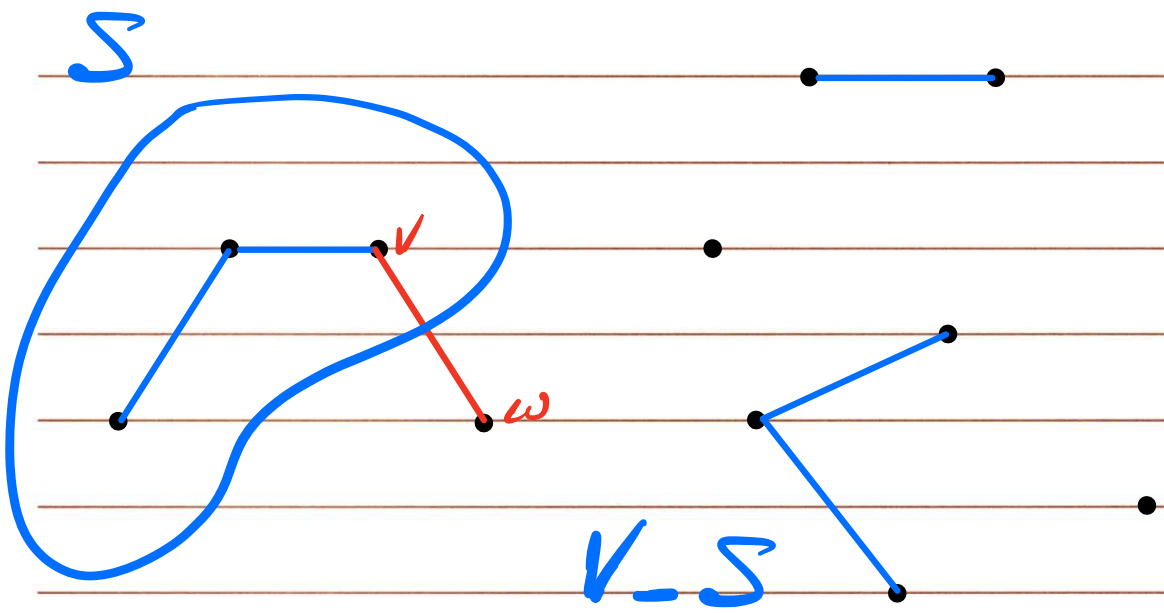
① Marks the i^{th} edge eliminated by the Reverse Delete alg. from the MST.

Fact: Let S be any subset of nodes that is neither empty nor equal to all of V, and let edge $e = (v, w)$ be the minimum cost edge with one end in S and the other end in $V - S$. Then every MST contains the edge e.



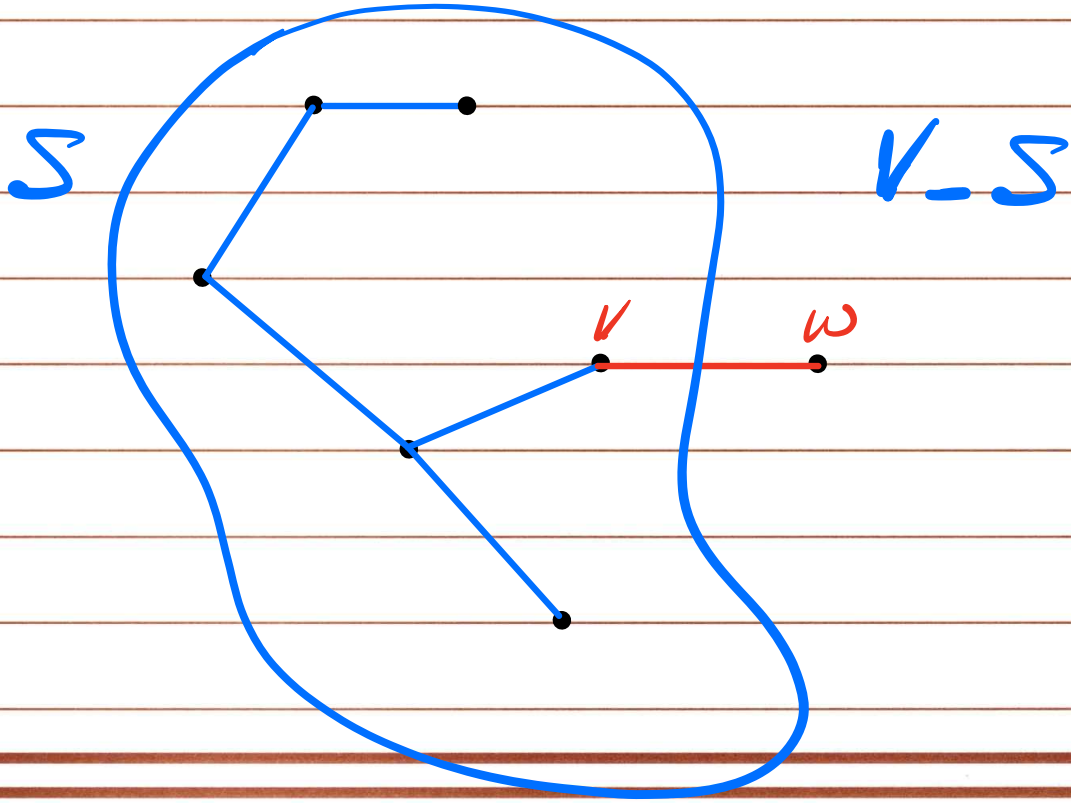
Proof by contradiction. Can replace v's' with e to create lower cost spanning tree.

Proof of correctness for Kruskal's Alg.



Each edge chosen by Kruskal's alg. is the lowest cost edge connecting S to V-S.

Proof of correctness for Prim's Alg.



Each edge chosen by Prim's alg. is the lowest cost edge connecting S to V-S.

Proof of correctness for Reverse-Delete.

FACT: The highest cost edge in a cycle cannot belong to a MST.

Each edge chosen by Reverse-Delete is a highest cost edge in a cycle.

More Detailed Implementation of ~~Dijkstra's~~ Prim's Alg.

S = Null

Initialize priority queue Q with all nodes V where $d(v)$ is the key value.
(All $d(v)$'s are set to ∞, except for s where $d(s)=0$)

While $S \neq V$

v = Extract_Min(Q)

$S = S \cup \{v\}$

for each vertex $u \in Adj(v)$

if $d(u) > d(v) + l_e$

Decrease_key($Q, u, d(v) + l_e$)

endfor

endwhile

Kruskal's Alg. – High level Implementation

Create an independent set for each node
A = Null
Sort edges in non-decreasing order of weight
For each edge $(u,v) \in E$ taken in this order
    if u and v are NOT in the same set, then
        $A = A \cup \{(u,v)\}$
        Merge the two sets
    endif

Endfor

Union-Find data structure

Operations:

– Make Set	O(1) for set of size 1
– Find Set	O(1) or O(lgn)
– Union	O(lgn) or O(1)
	Array Impl. / Pointer Impl.

More Detailed Implementation of Kruskal's Alg.

```
A = Null
For each vertex v ∈ V                    } O(n)
    Make_Set(v)
Endfor

Sort the edges of E into a non_decreasing order of cost    } O(mlgm)

For each edge (u,v) ∈ E in this order    } O(mlgn)
    if Find_Set(u) ≠ Find_Set(v) then
        A = A ∪ {(u,v)}
        Union(u,v)
    endif
Endfor
```

Overall complexity = $O(n) + O(m \lg m) + O(m \lg n)$

$= O(m \lg m)$

Prim's	Kruskal's
$O(m \lg n)$	$O(m \lg m)$

In the worst case where the graph is fully connected

$O(m \lg m) = O(m \lg n^2) = O(2m \lg n)$

$= O(m \lg n)$

Reverse-Delete – High level Implementation

Sort the edges of E into a non-increasing order of cost

For each edge $(u,v) \in E$ in this order
  if removing (u,v) does not disconnect the graph, then
    Remove (u,v)
  endif
Endfor

This will require a BFS or DFS search taking $O(m+n)$

Overall complexity is therefore $O(m^2)$