CSC 317: Data Structures and Algorithm Analysis

Dilip Sarkar

Department of Computer Science
University of Miami

UNIVERSITY
OF MIAMI

Outline I

Minimum Spanning Trees: Introduction

- Recall, we wanted to find minimum-cost optical network.

  What we wanted to find was a *minimum (cost) spanning tree* (MST).
- There are many problems that can be modeled as a weighted graph and then solutions to these problems are MSTs for corresponding graph.
  - Wiring an components on printed circuit board (PCB).
  - Building an electric grid.
  - Scheduling a robotic arm to drill holes on a PCB. etc.

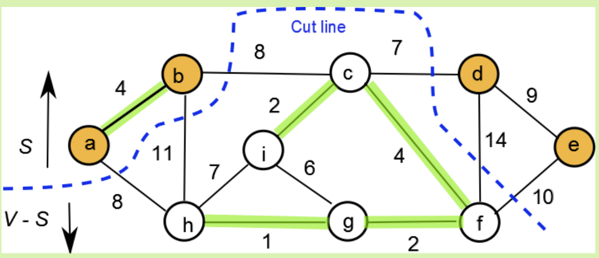
- **Def:** Let $G(V, E)$ be a graph, where $w(u, v)$ be weight of an edge $(u, v) \in E$. Let T be a tree on G. Weight $w(T)$ of the tree T is given as, $w(T) = \sum_{(u,v)\in T} w(u, v)$
- A **minimum spanning tree** has the minimum weight among all possible trees. Multiple MSTs are possible for a given $G(V, E)$.
- The problem of finding a MST is known as **MST problem.**
- We will learn two **greedy** algorithms for solving MST problems.
  1. Minimum-weight edge first
  2. Nearest-node first (among the remaining nodes)

Growing a Minimum Spanning Tree I

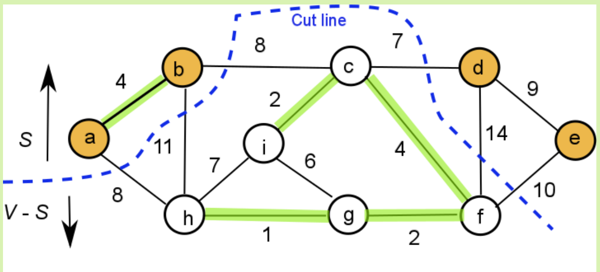
```
GENERIC-MST(G, w)
A = ∅
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    A = A ∪ {(u, v)}
return A
```

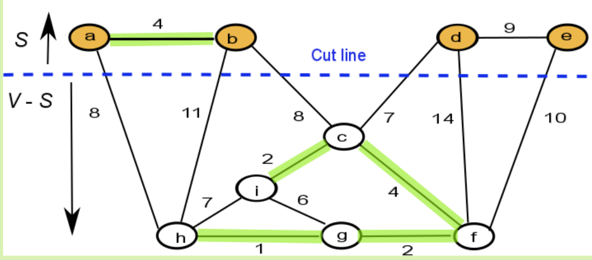
Let $S \subseteq V$ and $V - S$ be the subset of vertices that remains after the vertices in S are removed from V.

A **cut**



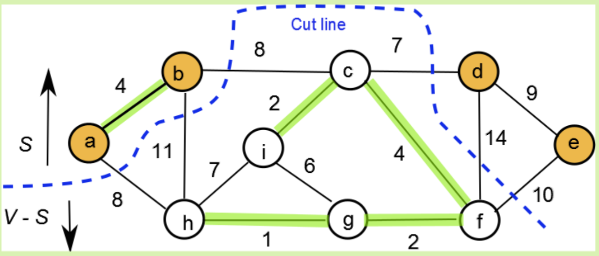
Redrawn graph for the cut





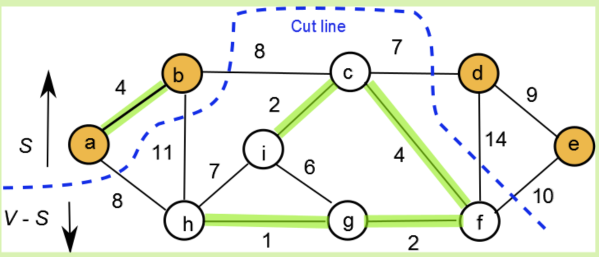
- A cut is used to develop an strategy for developing our **greedy** algorithms

Growing a Minimum Spanning Tree III



- An edge $(u, v) \in E$ **crosses** the cut $(S, V - S)$ of an undirected graph $G(V, E)$, it one of its endpoints is in S and the other is in $(V - S)$.
- A cut **respects** a set A of edges, if no edge in A crosses the cut.
- An example: $A = \{(a, b), (d, e)\}$ **respects** the cut $(\{a, b, d, e\}, \{c, f, g, h, i\})$.
- An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut.
- An Example: The edge (c, d) is a **light edge** for the cut $(\{a, b, d, e\}, \{c, f, g, h, i\})$.
- We will discuss two **greedy** algorithms for solving MST problems.
  1. Choose a minimum weight edge and **ensure** that it is a **safe** choice;
  2. A **safe** edge that connects to a nearest node among the remaining nodes

Growing a Minimum Spanning Tree III



Theorem (Theorem 23.1)

Let $G(V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, (u, v) is safe for A.

Proof.

Outline: The basic/intuitive idea is very simple. If we define two subgraphs S and $(V - S)$, and no MST-edge from S to $(V - S)$ exists, then a minimum weight (which is called a light) edge between these to subsets is a MST edge. Please read the details from the book. $\square$

Kruskal's Algorithm for MST I

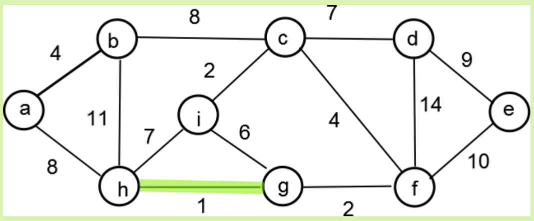
- Kruskal's algorithm is based on the **minimum-weight** edge first.
- As each step find the next minimum-weight edge and ensure that

  it is a **safe edge**.

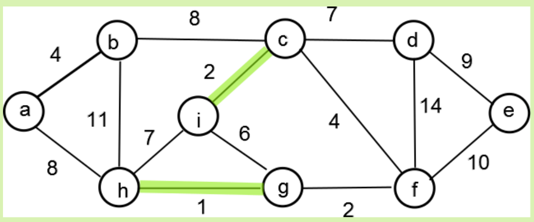
- Two data structures are used
- A sorted array of edges
- A set of set of vertices of the graph.
- An *edge* is **safe** if it does not form a cycle

Kruskal's Algorithm for MST II

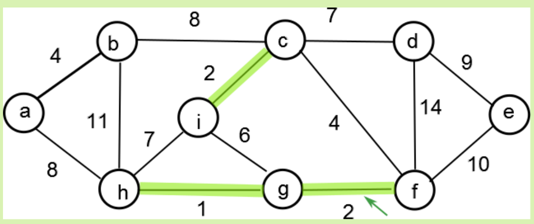
Step 1



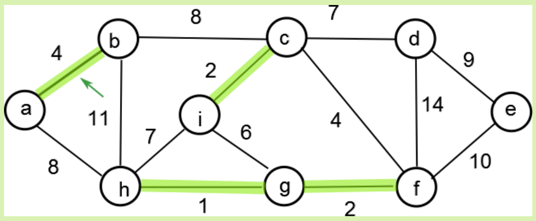
Step 2



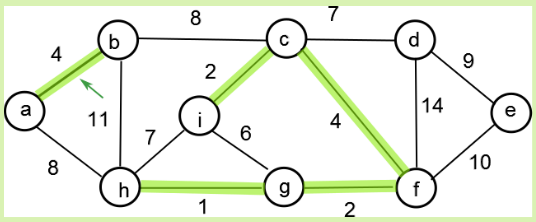
Step 3



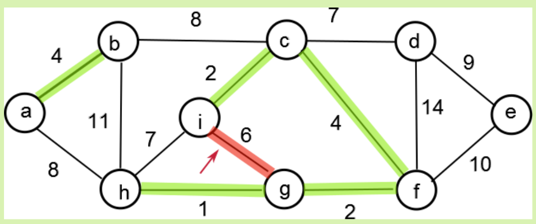
Step 4



Step 5

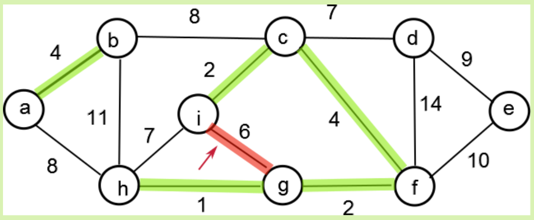


Step 6

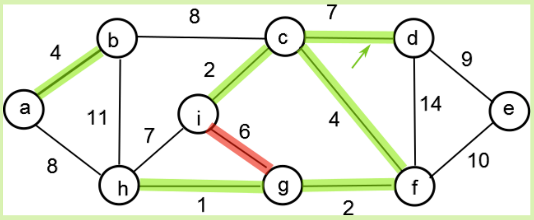


Kruskal's Algorithm for MST III

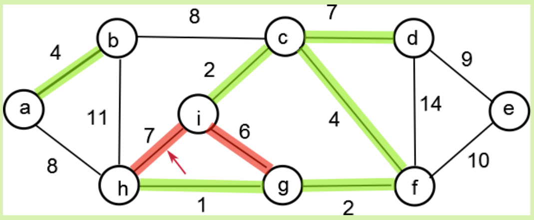
Step 6



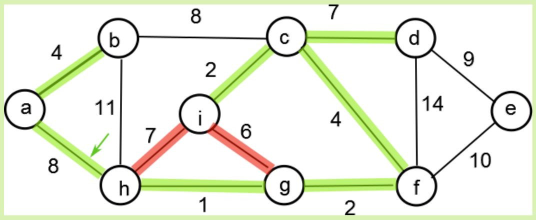
Step 7



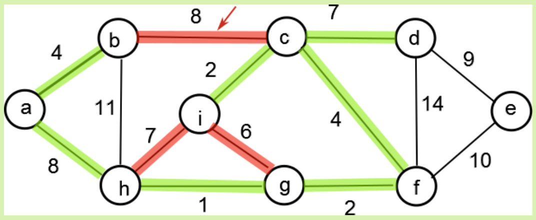
Step 8



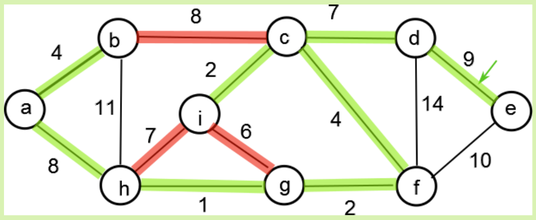
Step 9



Step 10

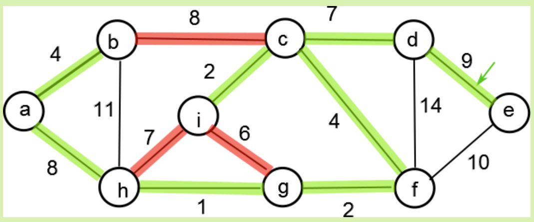


Step 11

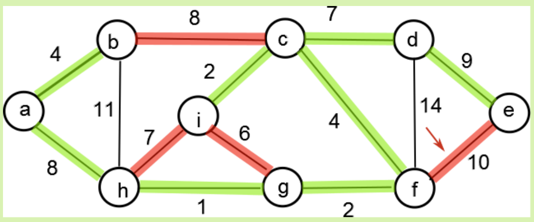


Kruskal's Algorithm for MST IV

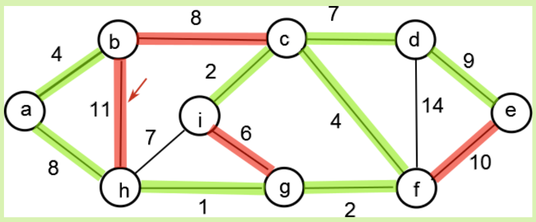
Step 11



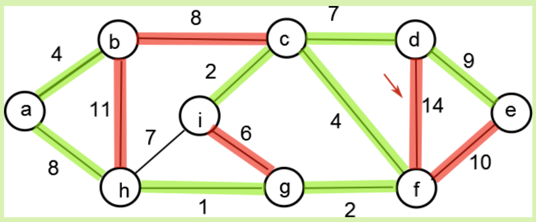
Step 12



Step 13



Step 14



Kruskal's Algorithm for MST V

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

- Data structures: Set of sets and an array of sorted edges
- Worst-case time complexity:

  Sorting of edges: $O(m \lg m)$ time
  MAKE-SET operations: $O(n)$ time
  FIND-SET operations: $2m$ FIND-SET operations take $O(m \lg n)$ time
  UNION operations: $(n-1)$ UNION operations take $O(n-1)$ time
- After adding everything, worst case time complexity is $O(m \lg m) = O(m \lg n)$

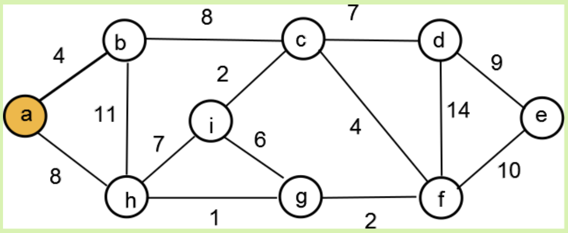
Prim's Algorithm for MST I

- The algorithm can start with any vertex

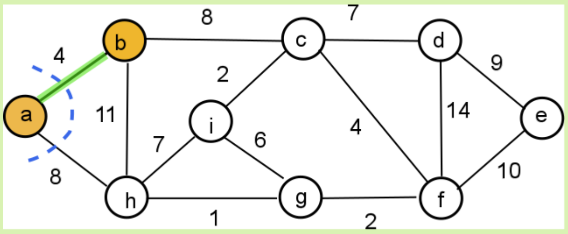
  For our example, starting vertex is a.
- The edges for a **selected** *subset of vertices*, S, have been selected
- If a cut is made to divide the *selected* vertices and the remaining vertices
- A safe edge from the cut edges is selected next.
- Let us see trace of an execution of the algorithm.

Prim’s Algorithm for MST II

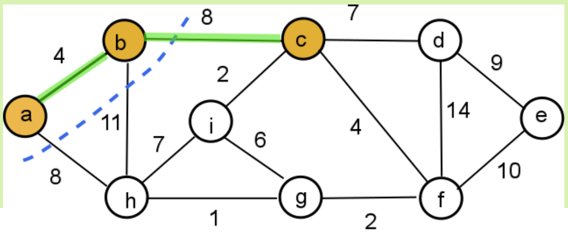
Step 1



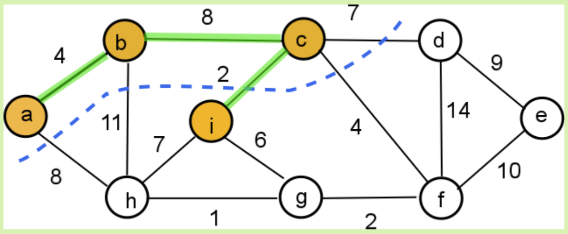
Step 2



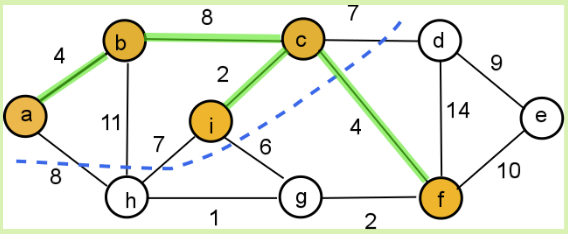
Step 3



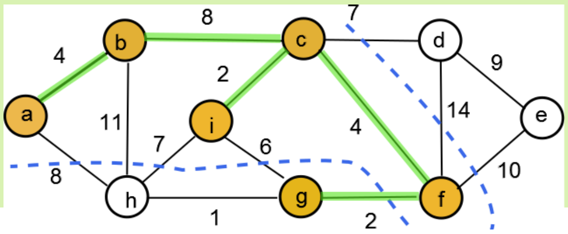
Step 4



Step 5



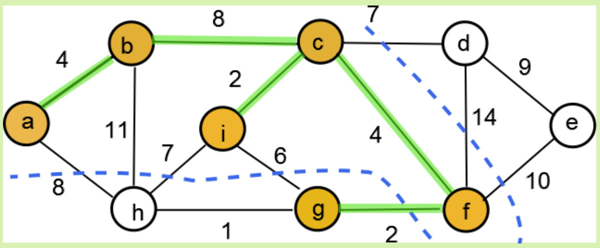
Step 6



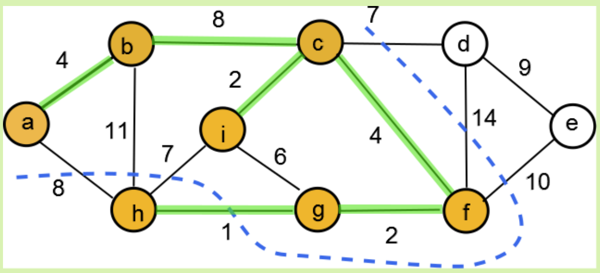
Growing a minimum spanning tree
Shortest Paths

Prim’s Algorithm for MST III

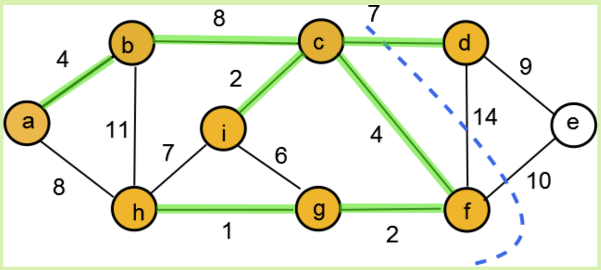
Step 6



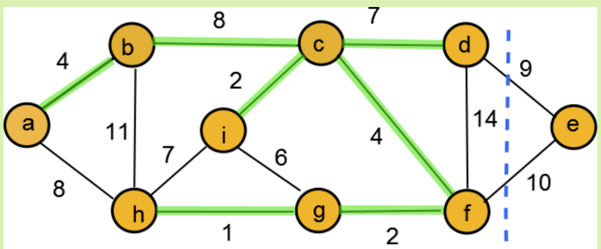
Step 7



Step 8



Step 9



Prim's Algorithm for MST IV

$A = \{(v, v.\pi) : v \in V - \{r\}\}$.

```
MST-PRIM(G, w, r)
1  for each u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = G.V
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      for each v ∈ G.Adj[u]
9          if v ∈ Q and w(u, v) < v.key
10             v.π = u
11             v.key = w(u, v)
```

- Data structures: An array of vertices,

  a vertex maintains $u.key$ for distance of the nearest vertex in the selected subtree

  a vertex also maintains a pointer $u.\pi$ to keep a link to its predecessor

  A priority queue Q to find a vertex that is connected to a **light** edge from the currently selected subtree
- Worst-case time complexity:
  - Initialization $O(n \lg n)$ for the priority queue and $O(n)$ for others
  - The loop is executed $O(m)$ time
    Line 7 takes $O(\lg n)$ time
  - Worst-case time for Lines 6 to 11 is $O(m + n \lg n)$
- Worst-case time complexity is $O(m + n \lg n)$

Shortest Paths

- Given a graph $G(V, E)$,

  a weight function $w : E \to \mathbb{R}$, and

  a path $p =< v_0, v_1, \cdots, v_k >$.
- The vertex v_0 is called source and the vertex v_k is called destination.

  Let the **weight** of path p be $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.
- **Shortest path** (SP) problem: given a weighted directed graph and two vertices u and v, find a path p from u to v such that $w(p)$ is the minimum among all paths from u to v.
- Let $p_1, p_2, \cdots, p_k$ be all the paths from u to v, then the shortest distance path from u to v,

  $\delta(u, v) = \min\{w(p_1), w(p_2), \cdots, w(p_k)\}$
- Four possible shortest path problems are:
  - **one-to-one** or single-pair shortest path problem
  - **one-to-all** or single-source shortest problem
  - **all-to-one** or single-destination shortest paths problem
  - all-to-all or all-pair shortest path problem

Optimal Substructure for a shortest path I

- Recall that optimal substructure is one of the key indicators that **dynamic programming** and **greedy algorithms** apply.
- Dijkstra's algorithm for solving SP problem is a greedy algorithm
- Floyd-Warshall for all-pair shortest path is a dynamic programming algorithm.
- Shortest-paths algorithms rely on the property that *a shortest path between two vertices contains other shortest paths within it.*

Lemma (Lemma 24.1 (subpaths of SPs are SPs))

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p =< v_0, v_1, \cdots, v_k >$ be a shortest path from vertex v_0 to v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} =< v_i, v_{i+1}, \cdots, v_j >$ be a subpath of p from vertex v_i to v_j. Then, p_{ij} is shortest path from v_i to v_j.

Proof.

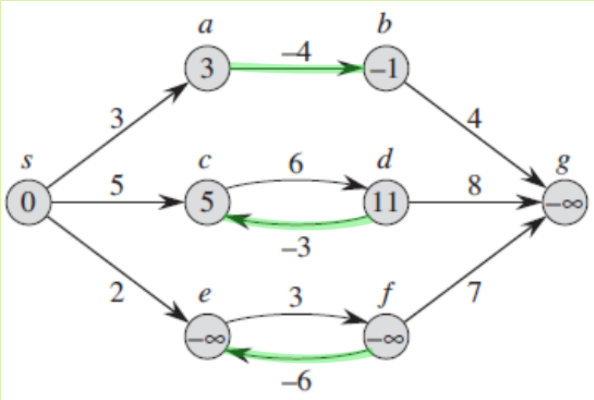
Outline: We use contradiction. Divide a path into three subpaths. Then assume that the middle subpath is not a shortest path. Find a shortest path for the middle section. Now the three subpaths together gives a shortest path that has smaller distance the original shortest path, which is a contradiction. □

Optimal Substructure for a shortest path II

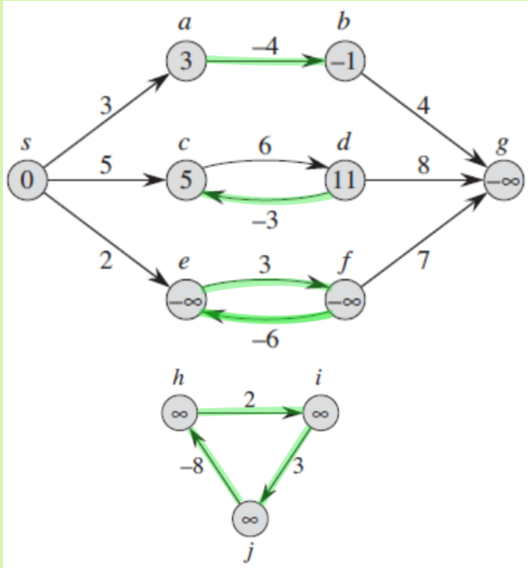
- Details for the proof.
- Let us decompose $p =< v_0, v_1, \cdots, v_k >$ into three subpaths:
  - p_{0i} from v_0 to v_i with distance $w(p_{0i})$
  - p_{ij} from v_i to v_j with distance $w(p_{ij})$
  - p_{0i} from v_i to v_k with distance $w(p_{jk})$
- Thus, $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$
- Assume that subpath p_{ij} from v_i to v_j is not shortest path from v_i to v_j
- Let the shortest subpath from v_i to v_j be p'_{ij}
- $\Rightarrow w(p'_{ij}) < w(p_{ij})$.
- $\Rightarrow w(p) < w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$.
- This is a contradiction to the fact that the assumption the path p is shortest.

Negative-weight edges

- A graph may have negative edges.
- Do they affect SP algorithms



- The graph above has three negative edges.
- The negative weights are: $w(a, b) = -4$, $w(c, d) = -3$, and $w(e, f) = -6$.



- The graph above is disconnected.
- It has 2 negative cycles: $w(e\,f\,e) = -3$, $w(h\,i\,j\,h) = -3)$
- A graph with a negative cycle has no SP.

Initialization and Edge Relaxation for SP algorithms

Initialize-Single Source SP

INITIALIZE-SINGLE-SOURCE (G, s)

1 **for** each vertex $v \in G.V$

2 $\quad v.d = \text{inf}$

3 $\quad v.\pi = \text{NIL}$

4 $s.d = 0$

———————————-

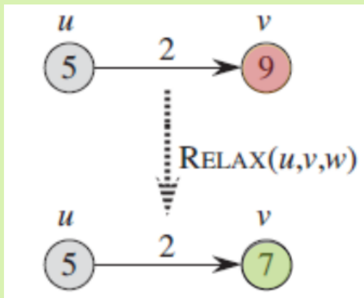
Relaxation

RELAX(u, v, w)

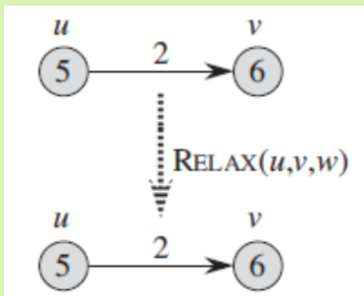
1 **if** $v.d > u.d + w(u, v)$

2 $\quad v.d = u.d + w(u, v)$

3 $\quad v.\pi = u$



- Relaxation when **if** clause is **true**



- Relaxation when **if** clause is **false**

Single-Source SPs in DAG I

DAG-SHORTEST-PATHS (G, w, s)

topologically sort the vertices of G

INITIALIZE-SINGLE-SOURCE(G, s)

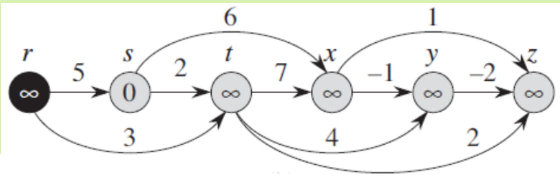
for each vertex u, taken in topologically sorted order

  for each vertex $v \in G.adj[u]$

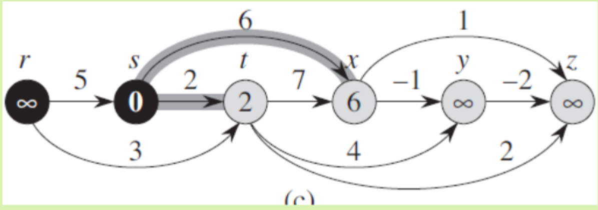
    RELAX(u, v, w)

- An example:

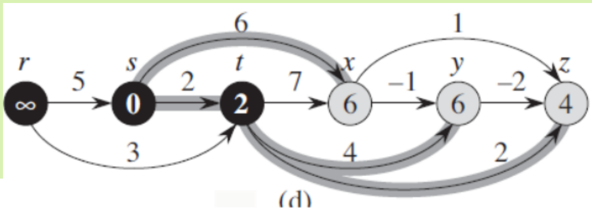
  Step 1: the vertices are topologically sorted from left to right. Starting node is s. Distance is shown in the circle.



Step 2: distances after relaxing edges (s, t) and (s, x). *An edge is highlighted if relaxation reduces distance.* Vertex s is final.

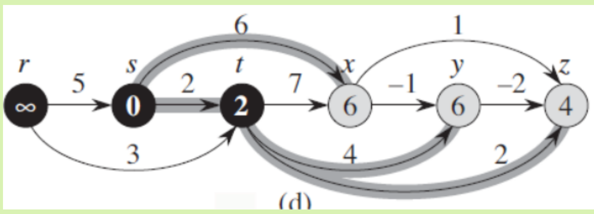


Step 3: distances after relaxing edges (t, x), (t, y) and (t, z). Vertex t is final.

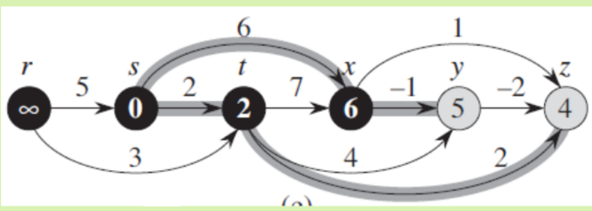


Single-Source SPs in DAG II

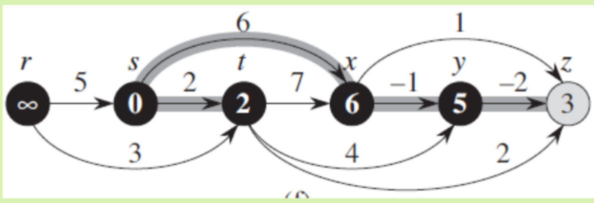
Step 3: distances after relaxing edges (t, x), (t, y) and (t, z). Vertex t is final.



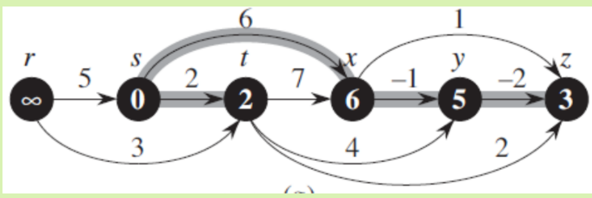
Step 4: distances after relaxing edges (x, y) and (x, z). Vertex x is final.



Step 5: distances after relaxing edge (y, z). Vertex y is final.



Step 6: No edge is relaxed. Vertex z is final.



Single-Source SPs in DAG III: Time complexity and Correctness

- Line 1: topological sorting takes $\Theta(n+m)$
- Line 2: Initialization takes $\Theta(n)$
- Lines 3-5: the **for** loop takes one iteration per vertex

  but the loop relaxes each edge only once, making complexity for each edge $\Theta(1)$ and $\Theta(m)$ for all edges.
- Thus, overall complexity is $\Theta(n+m)$

Theorem (Theorem 24.5)

If a weighted, directed graph $G(V, E)$ has sorve vertex s and no cycles, then at termination of the DAG-SHORTEST-PATHS *procedure, $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree.*

Proof.

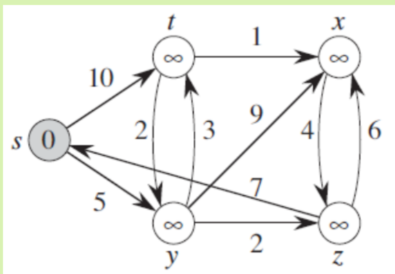
- Outline: Show that at termination $v.d = \delta(s, v)$ for all vertices.
- Because of predecessor subgraph property, G_π is a shortest paths tree.

□

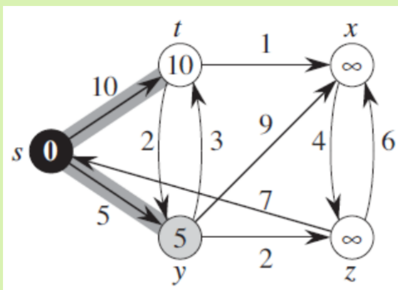
Dijkstra's Shortest Path Algorithm I

- **Requirement/Limitation:** Nonnegative edge weight
- Algorithm maintains a set S of vertices with already calculated shortest distances.
- At each iteration, it selects a vertex u from $V - S$ with minimum shortest-path estimate.
- Data structures:
  - S — a set of vertices
  - Q — a min-priority queue of vertices in $V - S$
  - $G.Adj$ — adjacency list representation of G
- Let us see trace of execution of the algorithm

After initialization: $S = \emptyset$ and top of the Q has $s.d = 0$

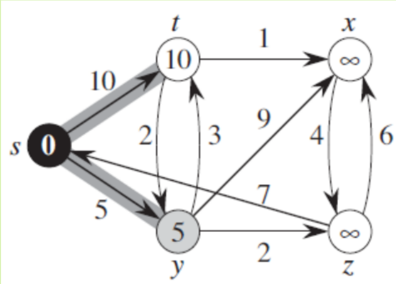


After 1st iteration: $S = \{s\}$ after initialization and top of the Q has $y.d = 5$

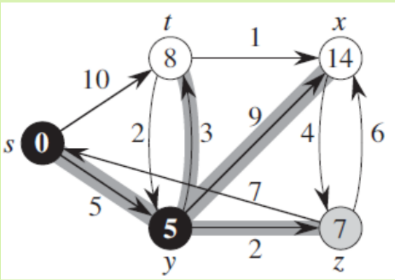


Dijkstra's Shortest Path Algorithm II

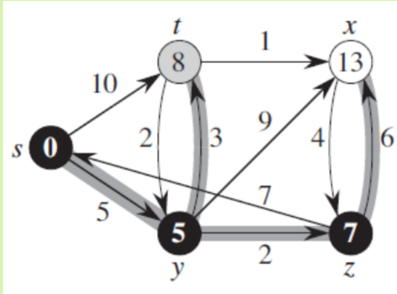
After 1st iteration: $S = \{s\}$ after initialization and top of the Q has $y.d = 5$



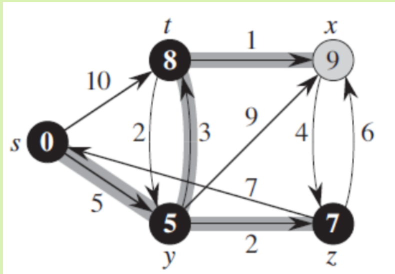
After 2nd iteration: $S = \{s, y\}$ and top of the Q has $z.d = 7$



After 3rd iteration: $S = \{s, y, z\}$ after initialization and top of the Q has $t.d = 8$

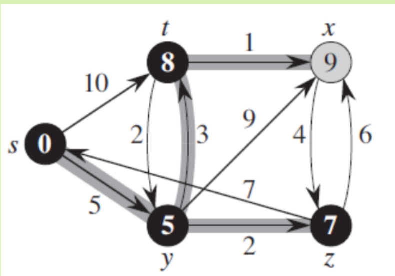


After 4th iteration: $S = \{s, y, z, t\}$ and top of the Q has $x.d = 9$

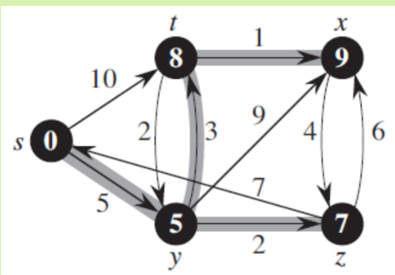


Dijkstra's Shortest Path Algorithm III

After 4th iteration: $S = \{s, y, z, t\}$ and top of the Q has $x.d = 9$



After 5th iteration: $S = \{s, y, z, t, x\}$ and top of the Q is empty.



DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Correctness and Complexity of Dijkstra's SP Algorithm

This is a greedy algorithm that selects the *closest* vertex from Q at every step.

Theorem (Theorem 24.6 — correctness of Dijkstra's algorithm)

Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and a source s terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Proof.

The proof use a loop invariant:

At the start of the each iteration of the **while** loop of lines 4-6, $v.d = \delta(s, v)$ for each vertex $v \in S$.

Initialization: Initially, $S = \emptyset$, so the invariant is trivially true.

Maintenance: we explain later.

Termination: At termination, $Q = \emptyset$, which implies that $S = V$. Thus, $u.d = \delta(s, u)$ for all vertices $u \in V$.

□

Correctness and Complexity of Dijkstra's SP Algorithm I

This is a greedy algorithm that selects the *closest* vertex from Q at every step.

Theorem (Theorem 24.6 — correctness of Dijkstra's algorithm)

Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and a source s terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Proof.

The proof use a loop invariant:

At the start of the each iteration of the **while** loop of lines 4-6, $v.d = \delta(s, v)$ for each vertex $v \in S$.

Initialization: Initially, $S = \emptyset$, so the invariant is trivially true.

Maintenance: we explain later.

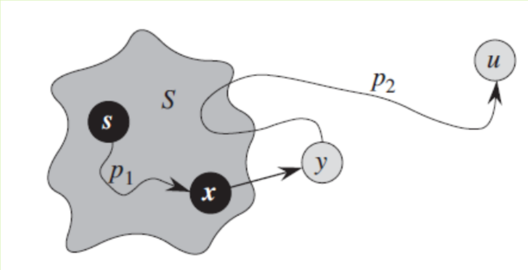
Termination: At termination, $Q = \emptyset$, which implies that $S = V$. Thus, $u.d = \delta(s, u)$ for all vertices $u \in V$.

□

Correctness and Complexity of Dijkstra's SP Algorithm II

Maintenance: Only an outline is provided. For details, read the book.

Since at initialization $s.d = 0$, at the first iteration s is added to S correctly. We only have to consider for values of $u \neq s$.



The idea of the proof is depicted in the figure above. At the end of the iteration vertex u is removed from Q and it is added to S.

Prior to adding u to S, a path p connects s to u.

Let us divide this path into three sections: a path from s to x, an edge from x to y (y is not in S), and a path from y to u.

vertex $x \in S$