An *improved* union-find algorithm

- <u>makeset(e)</u>: e.parent $\leftarrow$ e, the canonical element is the root.
- <u>find(e)</u>: Follow the parent pointers from e to the root of the tree containing e. Return the root.
- **<u>union(a,b)</u>: a.parent $\leftarrow$ canonical element of *larger* of two sets**

Why is this better?

Improved Union-Find: Union by Rank

Idea: Store with each node v the size of the subtree (*rank*) rooted at y. In a union operation, make the tree of the smaller set/rank a subtree of the other tree, and update the size field of the root of the resulting tree.

Union by Rank

- With each node x we store a nonnegative integer x.rank that is an upper bound on the height of x.
- When carrying out makeSet(x), we set x.rank $\leftarrow$ 0.
- To carry out union(x,y), we compare x.rank and y.rank.
  - If x.rank < y.rank then x.parent $\leftarrow y$.
  - If y.rank < x .rank then y.parent $\leftarrow x$.
  - If x.rank = y.rank then x .parent $\leftarrow y$ and y.rank $\leftarrow$ y.rank + 1.

Amortized Time Complexity

- A series of n makeSet, (modified) union, and find operations starting from an initially empty partition take
  O(n log n) time.

- The *amortized* running time per operation is O(log n) time.

A series of n makeSet, (modified) union, and find operations starting from an initially empty partition take $O(n \log n)$ time

Lemma: Starting with sets of size 1, using the modified union algorithm, any tree of m nodes has height of at most $\log m$.

An *improved* union-find algorithm

- <u>makeset(e)</u>: e.parent ← e, the canonical element is the root. $O(1)$
- <u>find(e)</u>: Follow the parent pointers from e to the root of the tree containing e. $O(\log n)$ Return the root.
- **<u>union(a,b)</u>: a.parent ← the canonical element of the *larger* set** $O(1)$

Reminder: Kruskal's algorithm

Step 1. Create a forest where each vertex in V represents a tree. O($|V|$)

Step 2. Sort the edges of G in increasing order.

O($|E|$ log $|E|$) or store in heap O($|E|$)

Step 3. Pick the shortest (minimum weight) edge (x,y) and check whether it belongs to different trees in the forest. ***Union-Find Data Structure***

Graph

- tree
- vertex
- edge

Union-Find Structure

- set
- element
- ----------

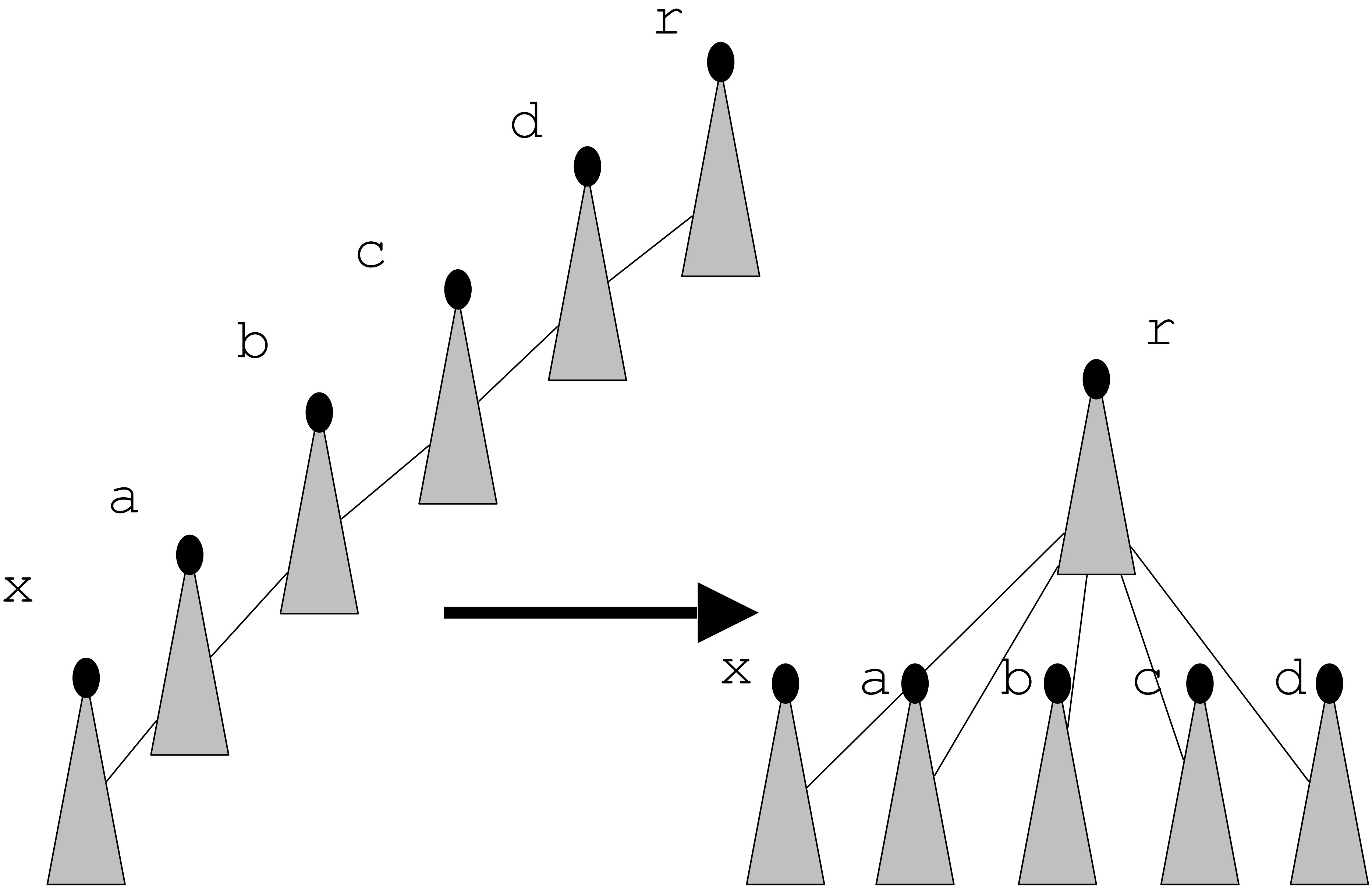
Union-Find:
Another Speed-Up Heuristic

- Union-by-size (also called Union-by rank) with
    - Path compression

Path compression

Idea: For each node v that the finds visits, reset the parent pointer from v to point to the root. In other words, after a find operation for an element x for every node v on the path from x to the root r is

$$v\text{.parent()} = r$$



r
d
c
b
a
x
r
x
a
b
c
d

Algorithm find(x):

```
if (x ≠ x.parent()) then
  x.parent ← find(x.parent())
return x.parent
```

Definitions and Claims (last class)

- *Amortized running time*

    overall running time/number of operations

- *Canonical element*

    root of tree

Definitions and Claims (last class)

- Claim 1: Once a node stops being a canonical element it can never become one again
- Claim 2: Once a node stopped being a canonical element its rank remains fixed
- Claim 3: Ranks are monotonically increasing in the trees as we travel from a node to the root of its tree
- Claim 4: When a node gets assigned rank k then there are at least 2^k elements in the subtree

Definitions and Claims

- Claim 5: The number of nodes that are assigned rank k throughout the execution of the union-find data structure is at most $n/2^k$
- Claim 6 (Corollary): The maximum rank of a tree in the union-find data structure with n elements in $\log(n)$
- Claim 7: The time to perform a single *find*-operation when we perform union-by-rank and path-compression is $O(\log(n))$

Amortized Time Complexity

- A series of m makeSet, union, and find operations on n elements using union-by-size and path compression starting from an initially single element sets take O(m ?) time.

- Then the amortized running time per operation is O(?) time.

Amortized Time Complexity

We conclude

- A series of $\boldsymbol{m}$ makeSet, union, and find operations for $\boldsymbol{n}$ elements, using union by rank, starting from an initially empty partition take $\mathbf{O}(\boldsymbol{n} + \boldsymbol{m} \log(\boldsymbol{n}))$ time.
- Then the amortized running time per operation is $\mathrm{O}(\log(n))$ time.

log*(n): The iterated logarithm

- log*(n) = min{i : $t(i) \geq n$} is the inverse of the *tower-of-twos function* $t(i)$ where

$$t(i) = \begin{cases} 1 & \text{if } i = 0 \\ 2^{t(i-1)} & \text{if } i \geq 1 \end{cases}$$

The tower-of-twos function

$$t(i) = \begin{cases} 1 & \text{if } i = 0 \\ 2^{t(i-1)} & \text{if } i \geq 1 \end{cases}$$

$\log^*(n)$: The iterated logarithm

- $\log^*(n) = \min\{i : t(i) \geq n\}$
- $\log^*(n) \leq 5$ for practical purposes

Even better: Amortized Time Complexity

- A series of n makeSet, union, and find operations for n elements using union-by-size and path compression starting from an initially single element sets take $O(n \log^*(n))$ time.
- Then the amortized running time per operation is $O(\log^*(n))$ time.

$\alpha(n)$

$\alpha(n)$: inverse Ackerman function

- grows slower than $\log^*(n)$

$$t(3) = 2^{t(3-1)} = 2^{t(2)} = 2^{2^{1}}$$

$$= 2^{2^{2^{1}}} = 2^{2^{2}}$$

$$t(5) = 2^{t(4)} = 2^{2^{t(3)}} = 2^{2^{2^{t(2)}}}$$

$$= 2^{2^{2^{2^{t(1)}}}} = 2^{2^{2^{2^{2}}}}$$

$\alpha(n)$: inverse of Ackerman function $A(n)$

- $\alpha(n) = \min\{m : A(m) \geq n\}$ where $A(n) = A_n(n)$ and

$$A_i(n) = \begin{cases} 2n & \text{for } i = 0 \text{ and } n \geq 0 \\ A_{i-1}(2) & \text{for } i \geq 1 \text{ and } n = 1 \\ A_{i-1}(A_i(n-1)) & \text{for } i \geq 1 \text{ and } n \geq 2 \end{cases}$$

n, $n \log n$, $n \log^* n$, and $n\,\alpha(n)$

- What is the growth order of these four functions?

n, $n \log n$, $n \log^* n$, and $n\,\alpha(n)$

- What is the growth order of these four functions?

- $\log^* n$ grows slower than $\log n$.

n, $n \log n$, $n \log^* n$, and $n\,\alpha(n)$

- What is the growth order of these four functions?

- $\log^* n$ grows slower than $\log n$.
- $\alpha(n)$ grows slower than $\log^* n$.

n, $n \log n$, $n \log^* n$, and $n\,\alpha(n)$

- What is the growth order of these four functions?

- $\log^* n$ grows slower than $\log n$.
- $\alpha(n)$ grows slower than $\log^* n$.
- n grows slower than $n \log n$, $n \log^* n$, and $n\,\alpha(n)$

$\alpha(n)$: inverse Ackerman function

$$A(2) = A_2(2) = A_1(A_2(1)) = A_1(A_1(2))$$

$$= A_1(A_0(A_1(1))) = A_1(A_0(2^2))$$

$$= A_1(2 \cdot 2^2) = A_1(2^3) = A_0(A_1(2^3 - 1))$$

$$= A_0(A_0(A_1(2^3 - 2))) = A_0(A_0(A_0(A_1(2^3 - 3))))$$

$$= A_0(A_0(A_0(A_0(A_1(2^3 - 4)))))$$

$$= A_0(A_0(A_0(A_0(A_0(A_1(2^3 - 5))))))$$

$$= A_0(A_0(A_0(A_0(A_0(A_0(A_1(2^3 - 6)))))))$$

$$= A_0(A_0(A_0(A_0(A_0(A_0(A_0(A_1(2^3 - 7))))))))$$

$\alpha(n)$: inverse Ackerman function

$$= A_0(A_0(A_0(A_0(A_0(A_0(A_0(A_1(1))))))))$$

$$= A_0(A_0(A_0(A_0(A_0(A_0(A_0(2^2)))))))$$

$$= A_0(A_0(A_0(A_0(A_0(A_0(2^3))))))$$

$$= A_0(A_0(A_0(A_0(A_0(2^4)))))$$

$$= A_0(A_0(A_0(A_0(2^5))))$$

$$= A_0(A_0(A_0(2^6))) = A_0(A_0(2^7))$$

$$= A_0(2^8) = 2^9$$

And even better: Amortized Time Complexity

- A series of n makeSet, union, and find operations for n elements using union-by-size and path compression starting from an initially single element sets take
  $O(n\ \alpha(n))$ time.
- Then the amortized running time per operation is $O(\alpha(n))$ time.

Kruskal's Algorithm

Time complexity analysis

Algorithm Kruskal
Input: a weighted connected graph $G = (V, E)$
Output: an MST T for G
Data structure: Disjoint sets (lists or union-find) DS;
sorted weights A[]; and tree T

for each vertex v in V of G **do** C(v) ← DS.insert(v) **end** — $O(n)$
A[] ← sort all edges by edge weight; // sorted array priority queue — $O(m \log m)$ or O(m) with heap
$T \leftarrow \varnothing$; $k \leftarrow 0$;
while T has fewer than n-1 edges **do**
  (u, v) ← A[k]; $k \leftarrow k + 1$; // next edge with smallest weight; deleteMin()
  $C(v)$ ← DS.findCluster(v);
  $C(u)$ ← DS.findCluster(u);
  if $C(v) \neq C(u)$ **then**
    add edge (v, u) to T;
    DS.insert(DS.union($C(v)$, $C(u)$)); // merge two clusters — Total merging $O(m \log n)$
  end
end
return T

Total $O(m \log n)$

Prim's Algorithm

- Initialize tree with single chosen vertex

Cut property

- Grow tree by finding lightest edge not yet in tree and expanding the tree, and connect it to tree; repeat until all vertices are in the tree

- *Example of greedy algorithm*

Implementing Prim's algorithm

- Using heaps

Pseudocode: Prim's Algorithm

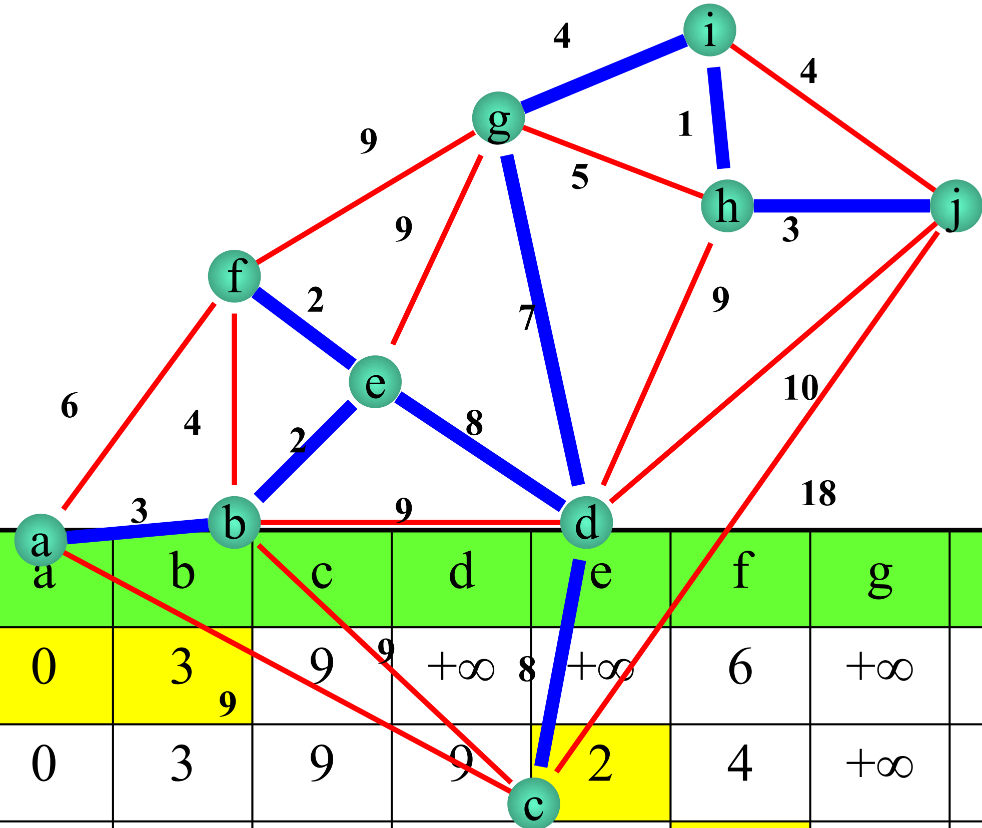
D :distance vector, maintains reachable vertices

PQ :a priority queue for the edges according to values in D

Algorithm Prim-Jarník-Dijkstra
Input: a weighted connected graph $G = (V, E)$
Output: an MST T for G
Data structure: array D; Priority Queue PQ; and tree T

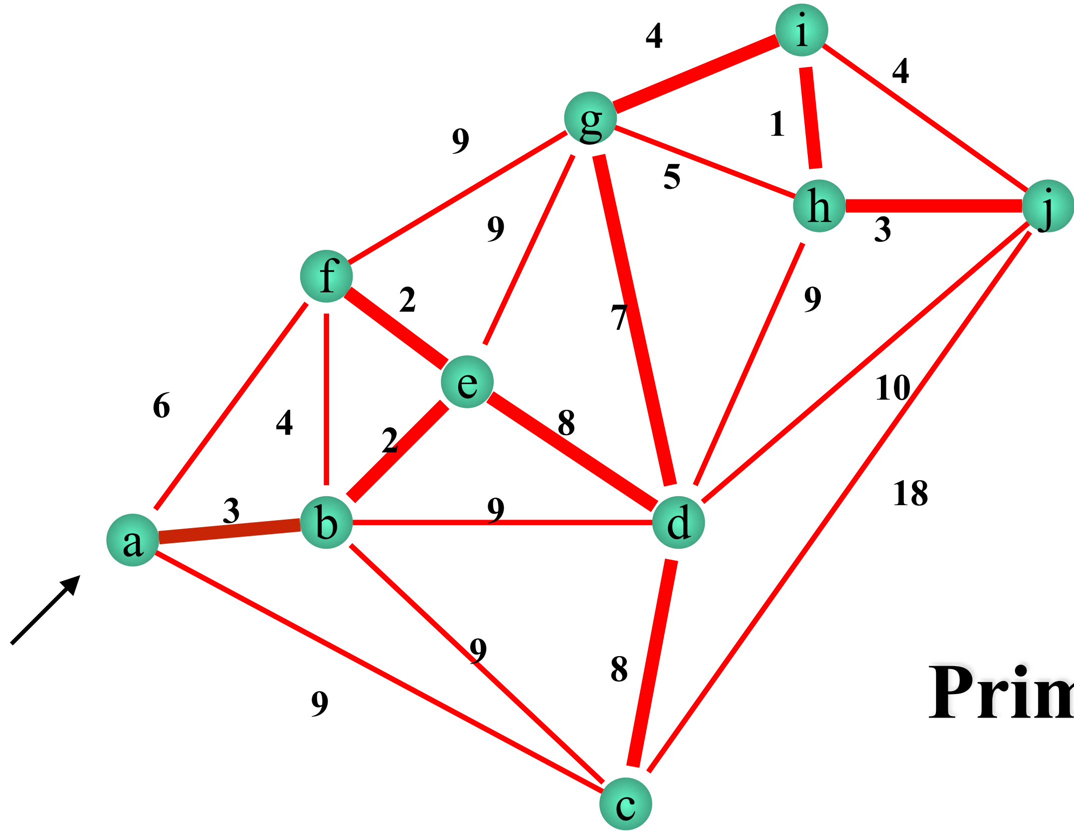
pick an arbitrary vertex v in G; $D[v] \leftarrow 0$
for each vertex $u \neq v$ **do** $D[u] \leftarrow +\infty$ **end**
$T \leftarrow \varnothing$
for each vertex u **do** PQ.insert({(u, (null), $D[u]$}) **end** // including v
// for each vertex u, (u, edge) is the element and $D[u]$ is the key in PQ
while not PQ.empty() **do**
  $(u, e) \leftarrow PQ$.deleteMin()
  add vertex u and edge e to T
  for each vertex z adjacent to u such that z is in PQ **do**
    if weight((u, z)) $< D[z]$ **then**
      $D[z] \leftarrow$ weight((u, z))
      in PQ, change element and key of z to $\{z, (u, z), D[z]\}$
      update PQ
    end
  end
end
return T

Prim-Jarník-Dijkstra



D

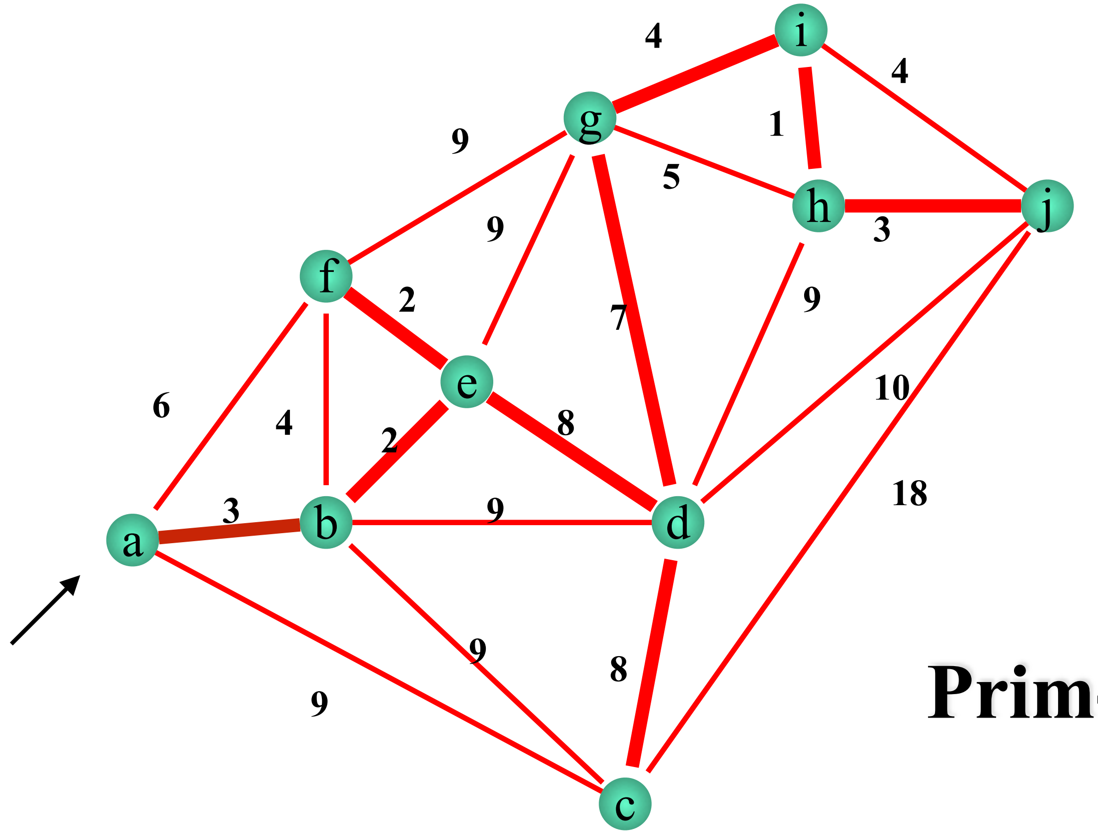
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	7	9	+∞	10
0	3	8	8	2	2	7	5	4	10
0	3	8	8	2	2	7	1	4	4
0	3	8	8	2	2	7	1	4	3
0	3	8	8	2	2	7	1	4	3
0	3	8	8	2	2	7	1	4	3



Prim-Jarník-Dijkstra

D

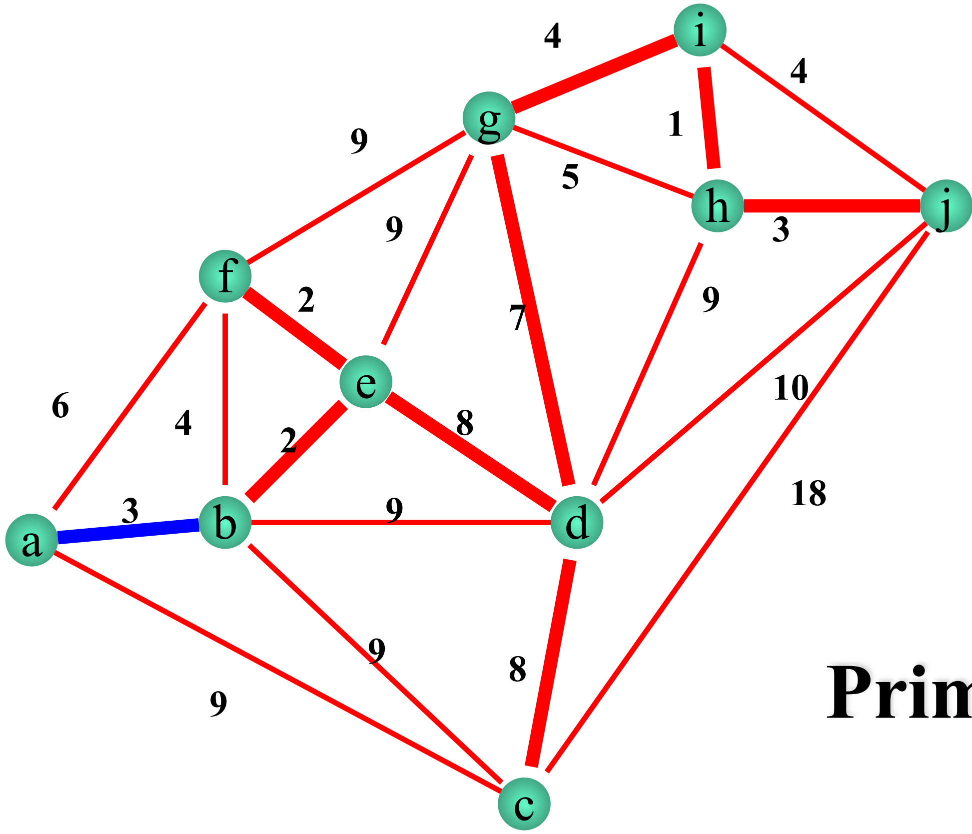
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞



Prim-Jarník-Dijkstra

D

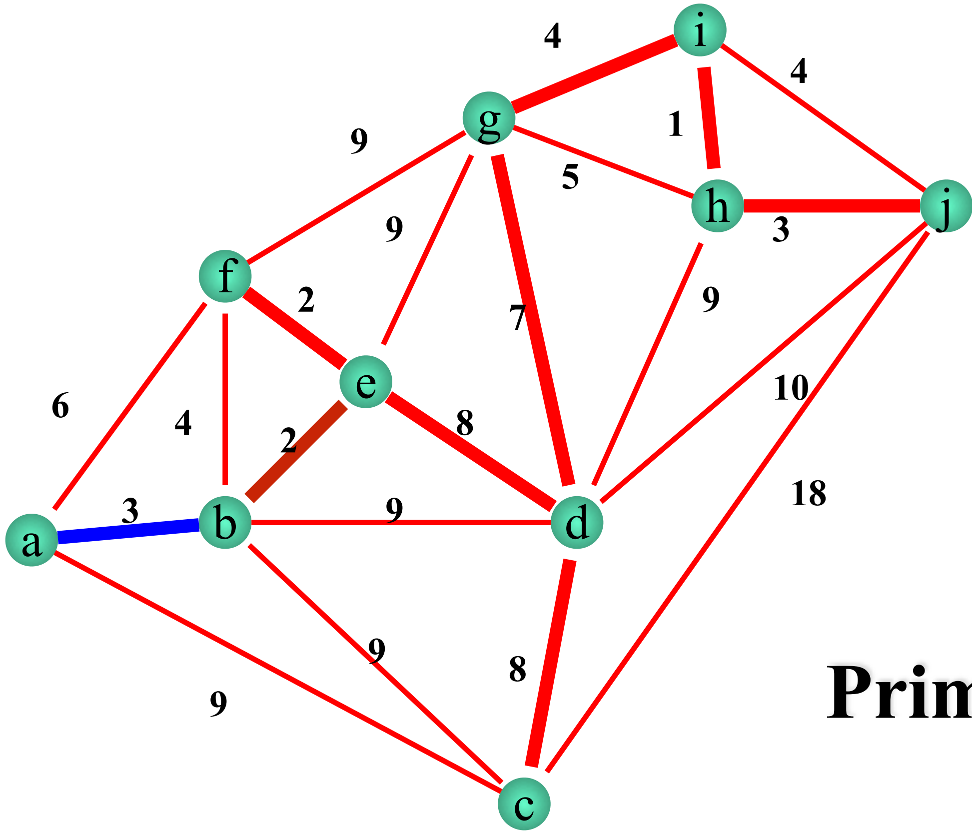
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞



Prim-Jarník-Dijkstra

D

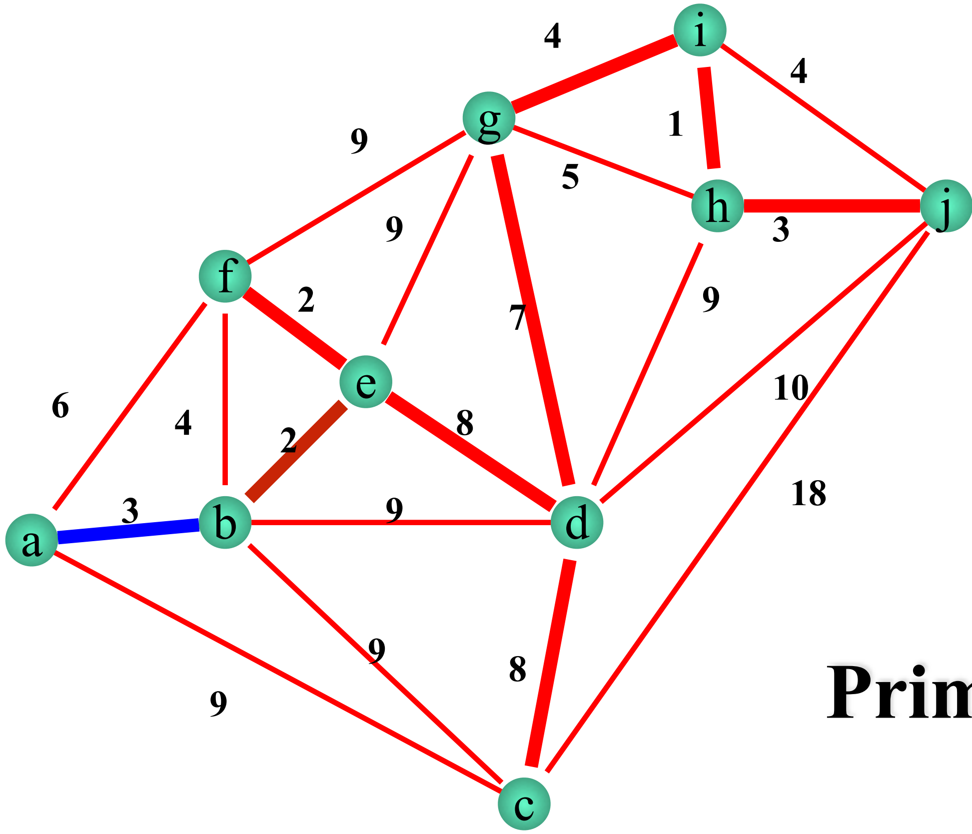
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞



Prim-Jarník-Dijkstra

D

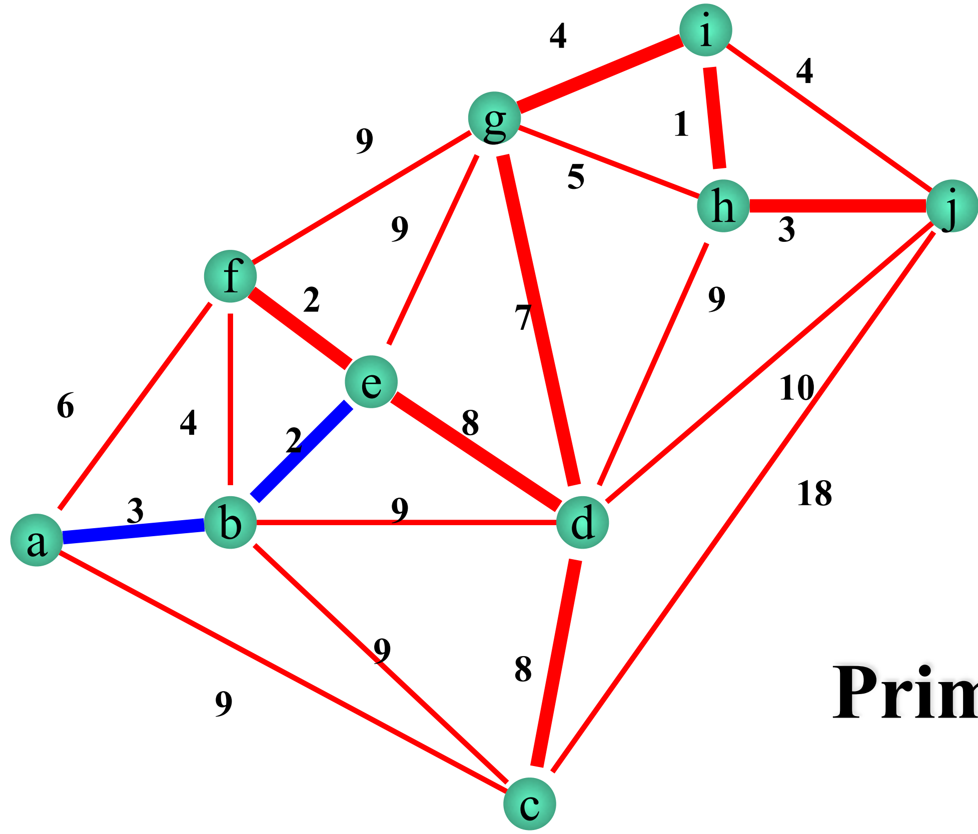
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞



Prim-Jarník-Dijkstra

D

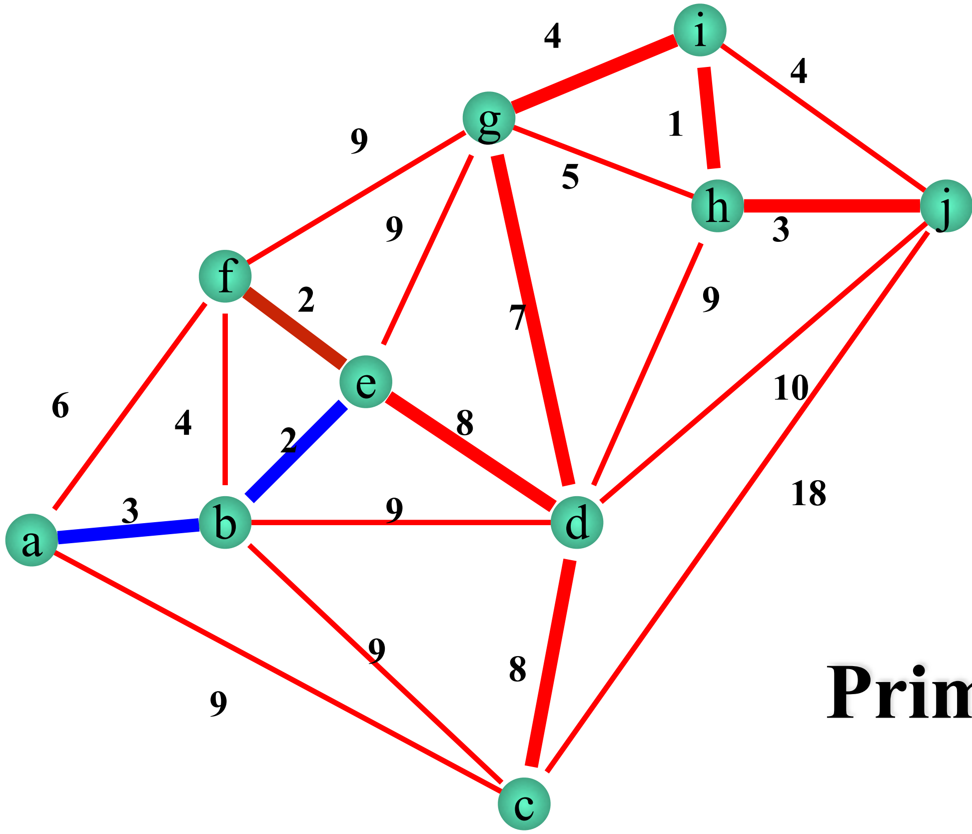
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞



Prim-Jarník-Dijkstra

D

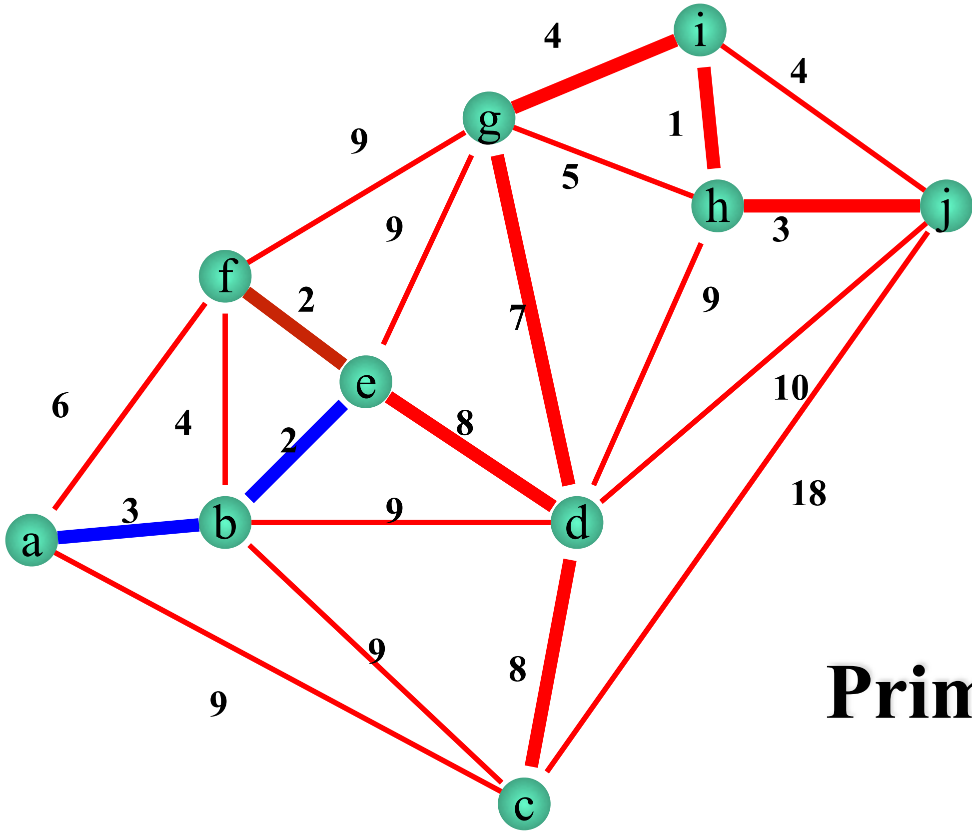
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞



Prim-Jarník-Dijkstra

D

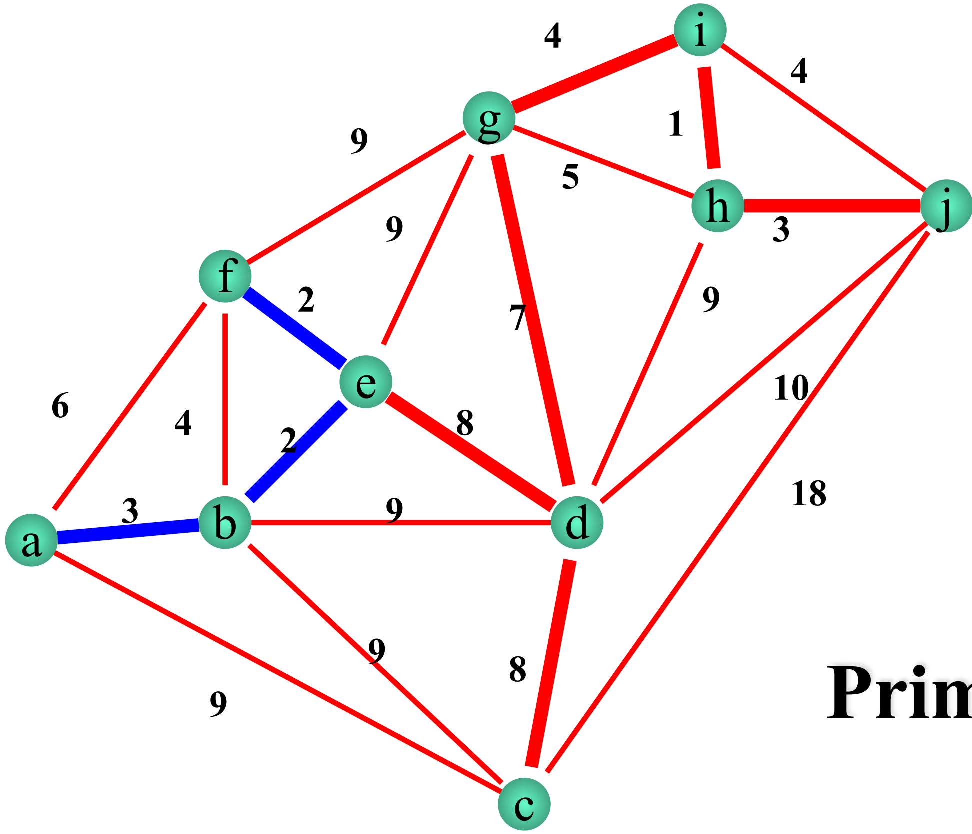
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞



Prim-Jarník-Dijkstra

D

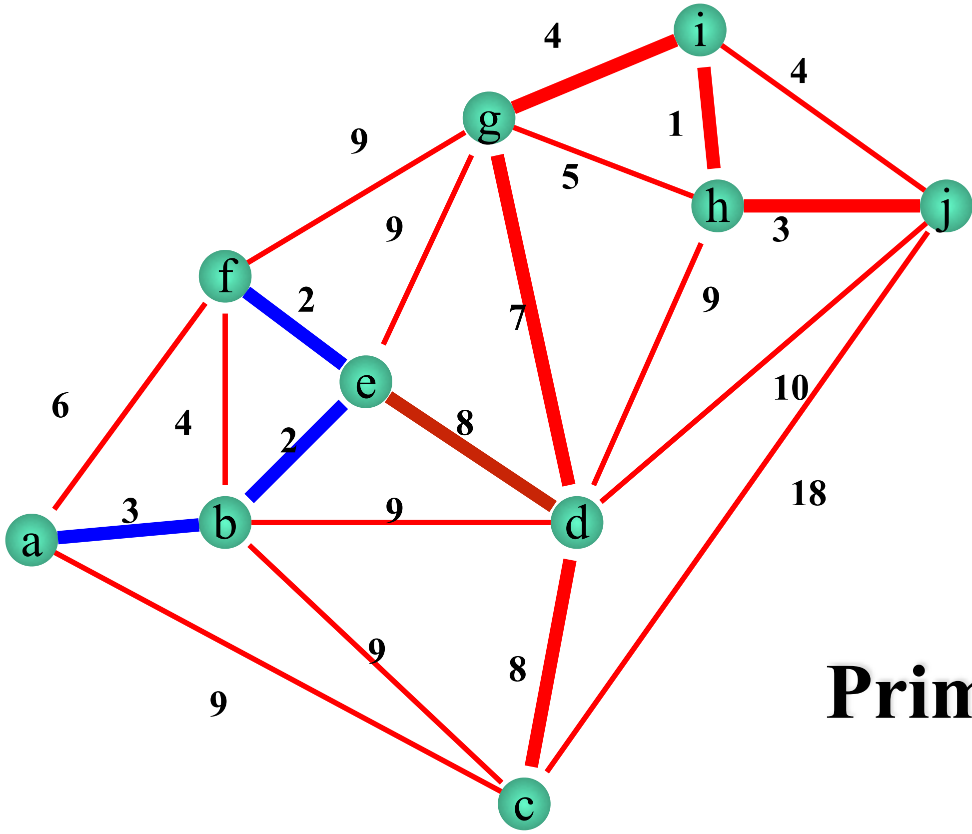
a	**b**	c	d	**e**	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞



Prim-Jarník-Dijkstra

D

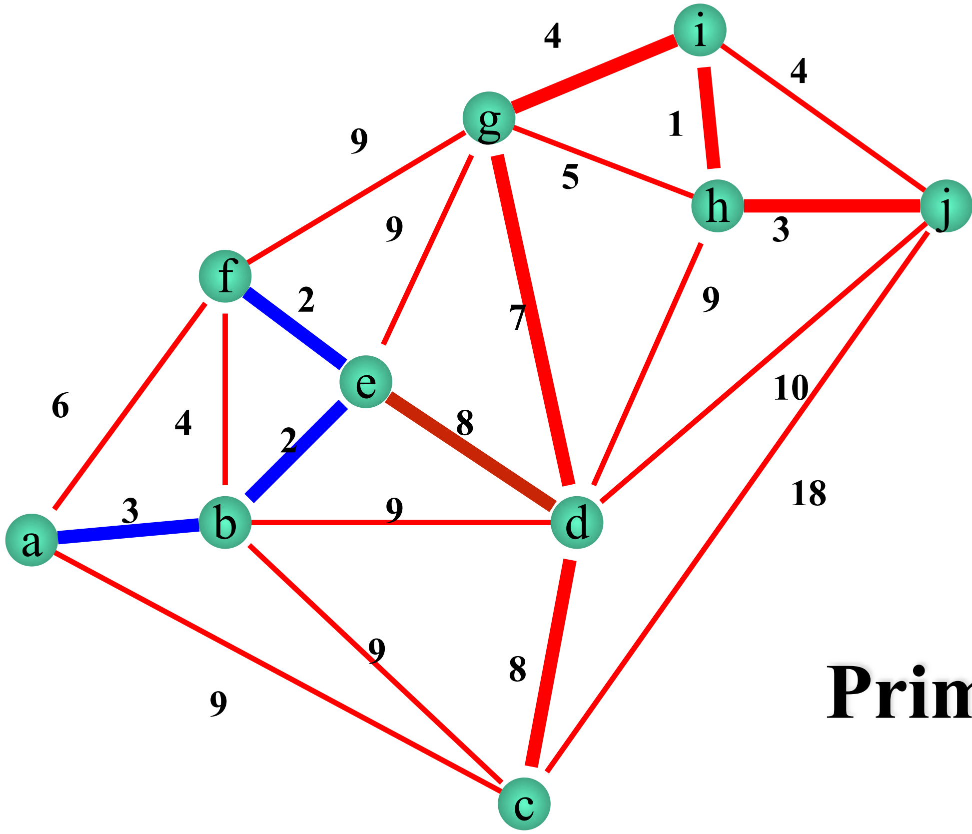
a	**b**	c	d	**e**	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞



Prim-Jarník-Dijkstra

D

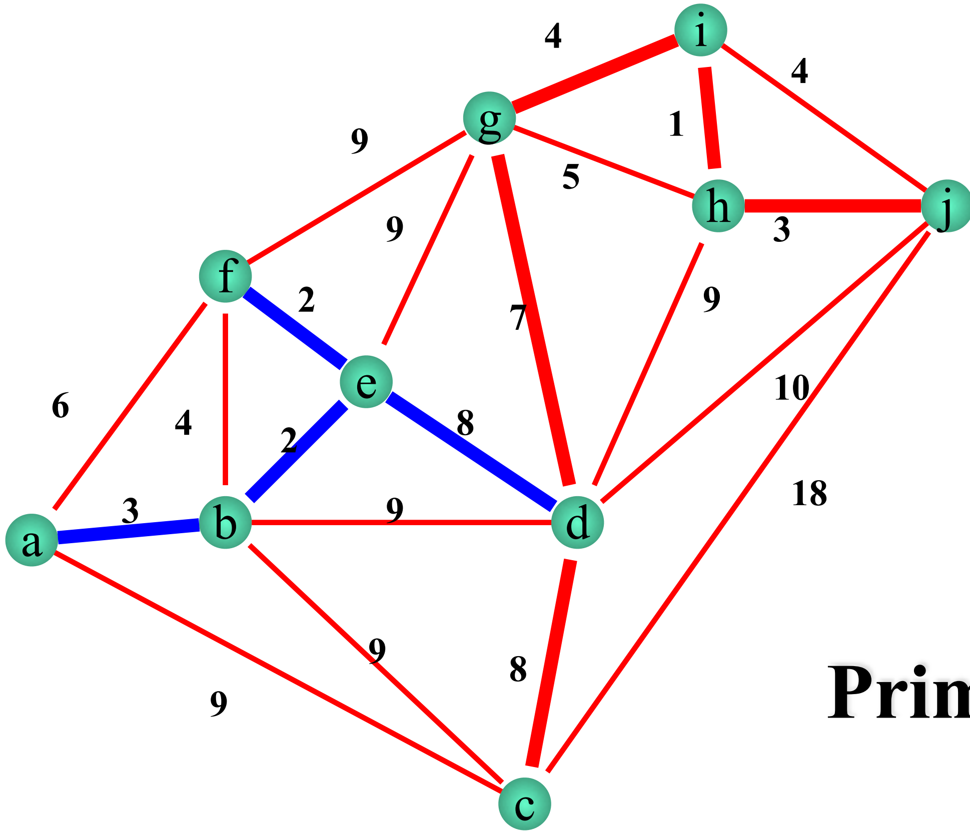
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	9	+∞	+∞	+∞



Prim-Jarník-Dijkstra

D

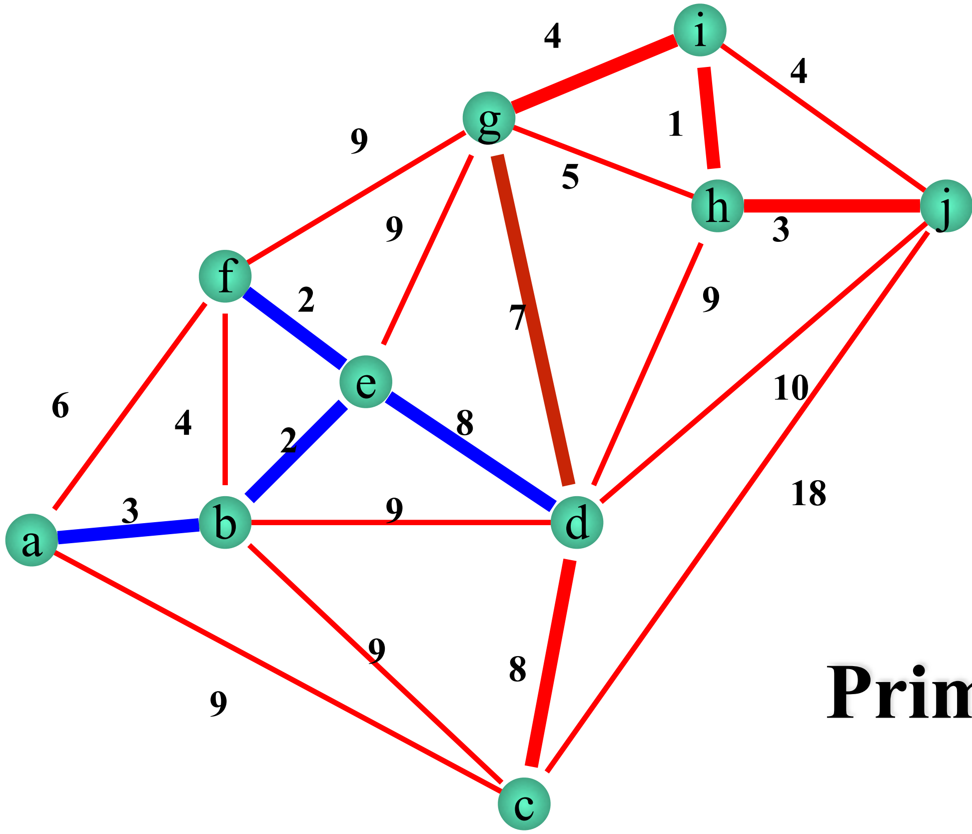
a	**b**	c	d	**e**	**f**	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	9	+∞	+∞	+∞



Prim-Jarník-Dijkstra

D

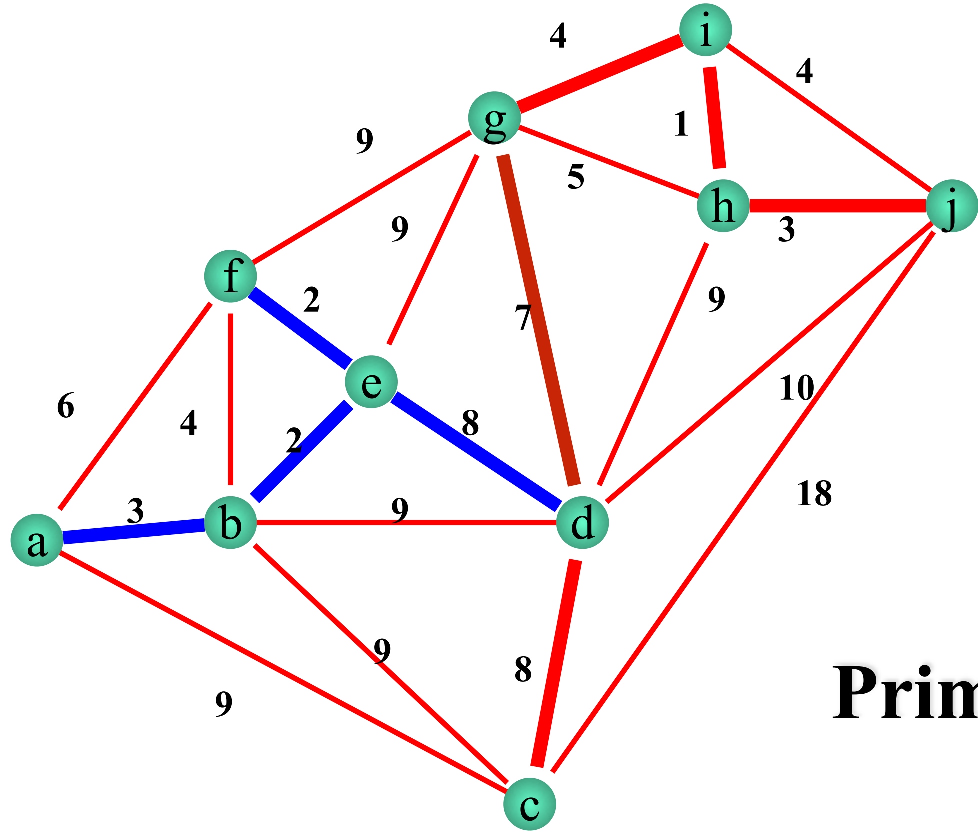
a	**b**	c	d	**e**	**f**	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	9	9	+∞	+∞



Prim-Jarník-Dijkstra

D

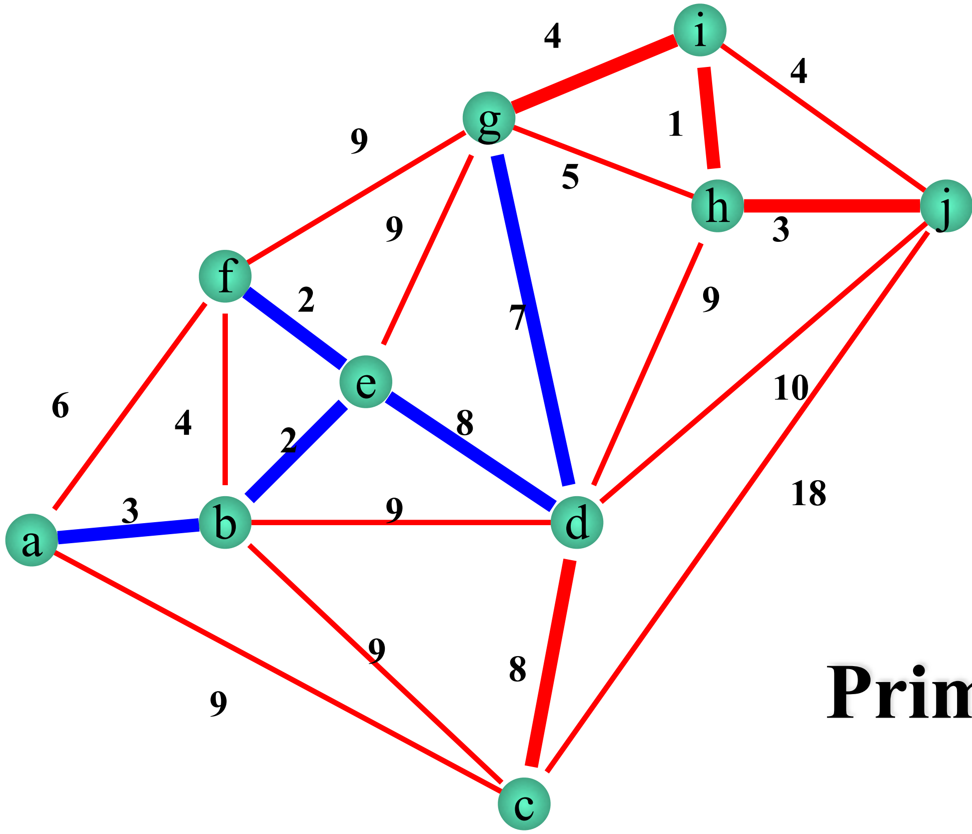
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	9	9	+∞	+∞
0	3	8	8	2	2	7	9	+∞	10



Prim-Jarník-Dijkstra

D

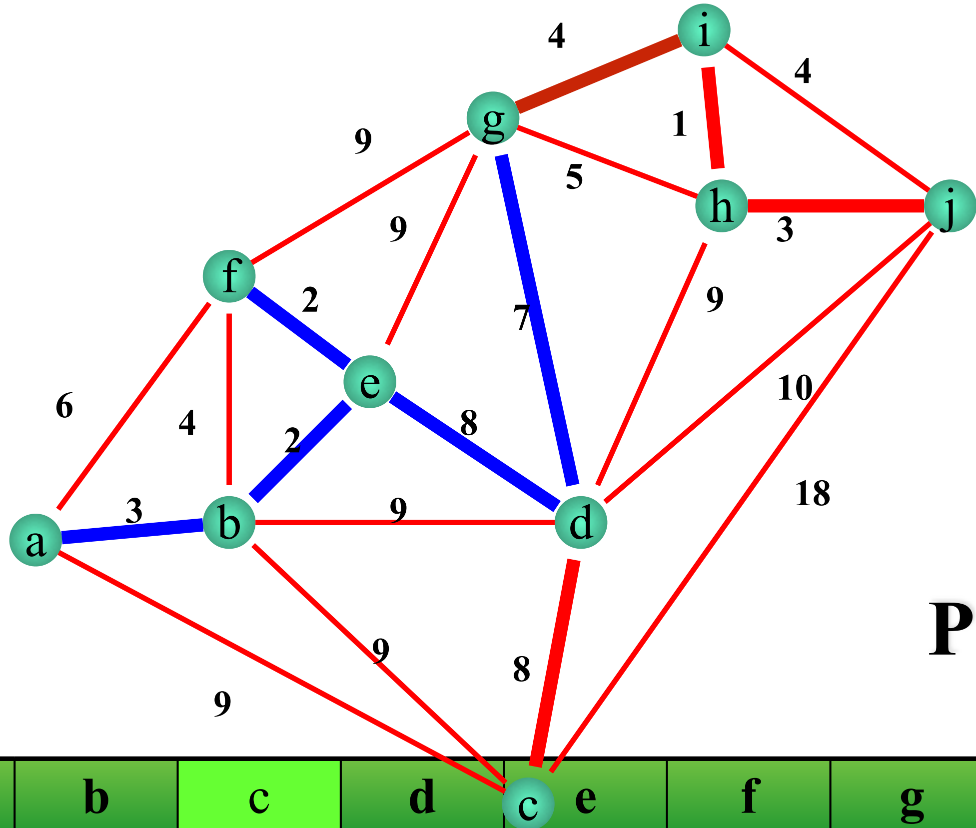
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	9	+∞	10



Prim-Jarník-Dijkstra

D

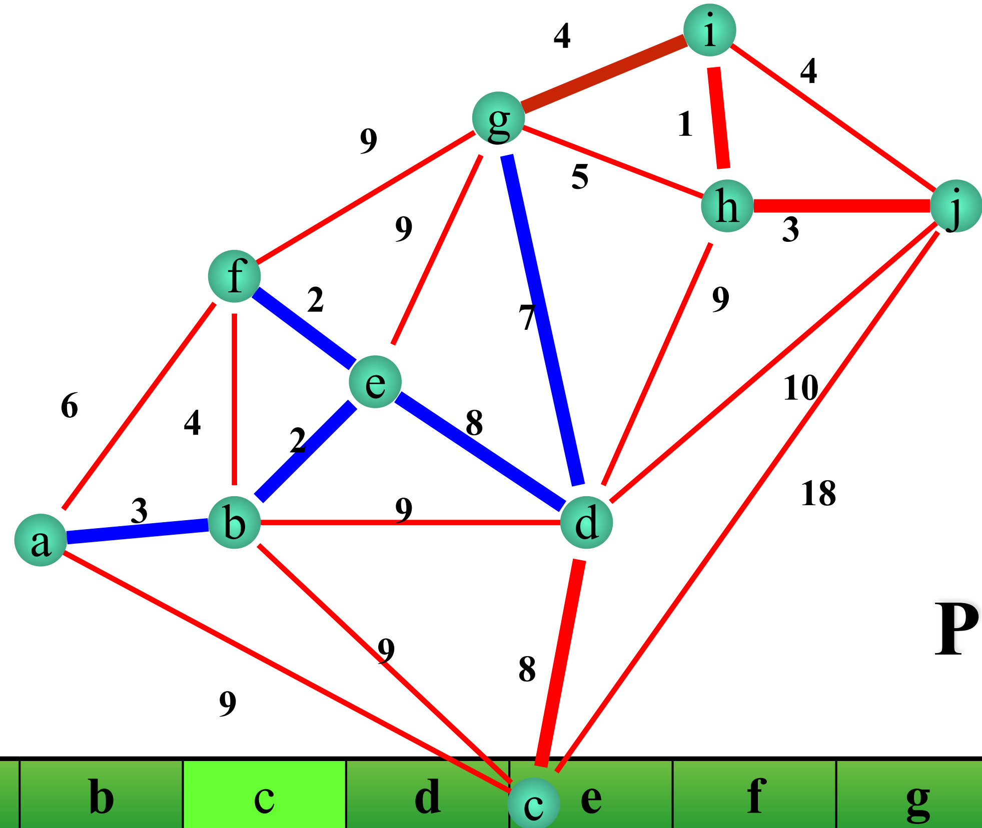
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	9	+∞	10



Prim-Jarník-Dijkstra

D

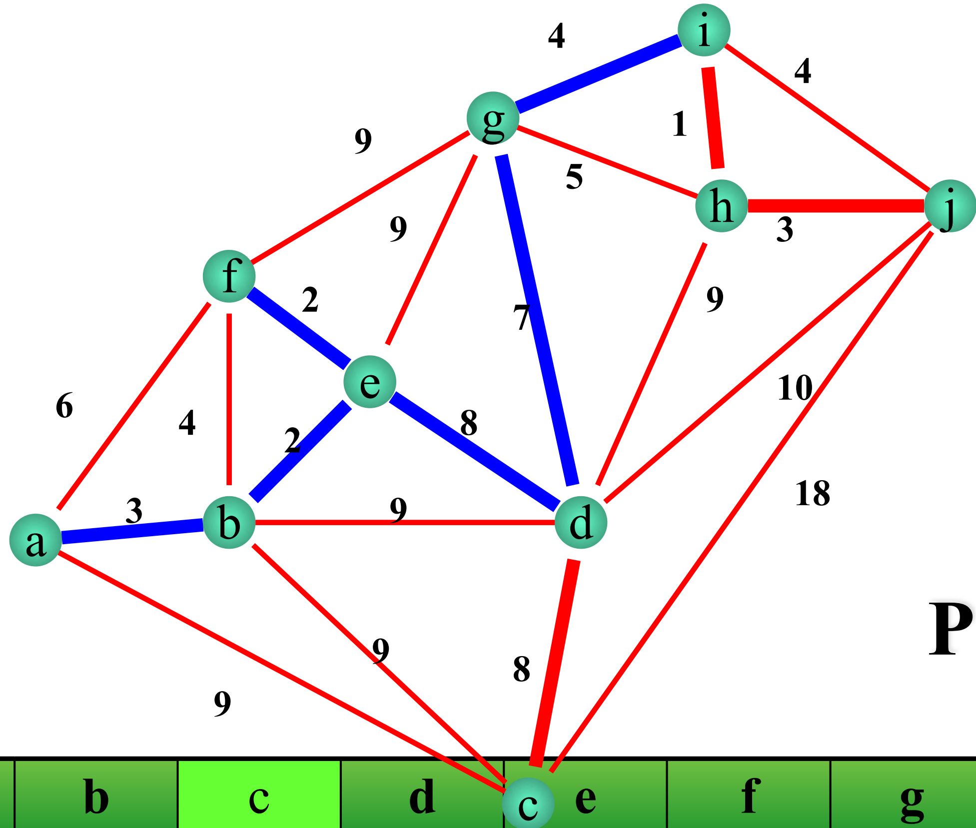
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	9	+∞	10
0	3	8	8	2	2	7	1	4	4



Prim-Jarník-Dijkstra

D

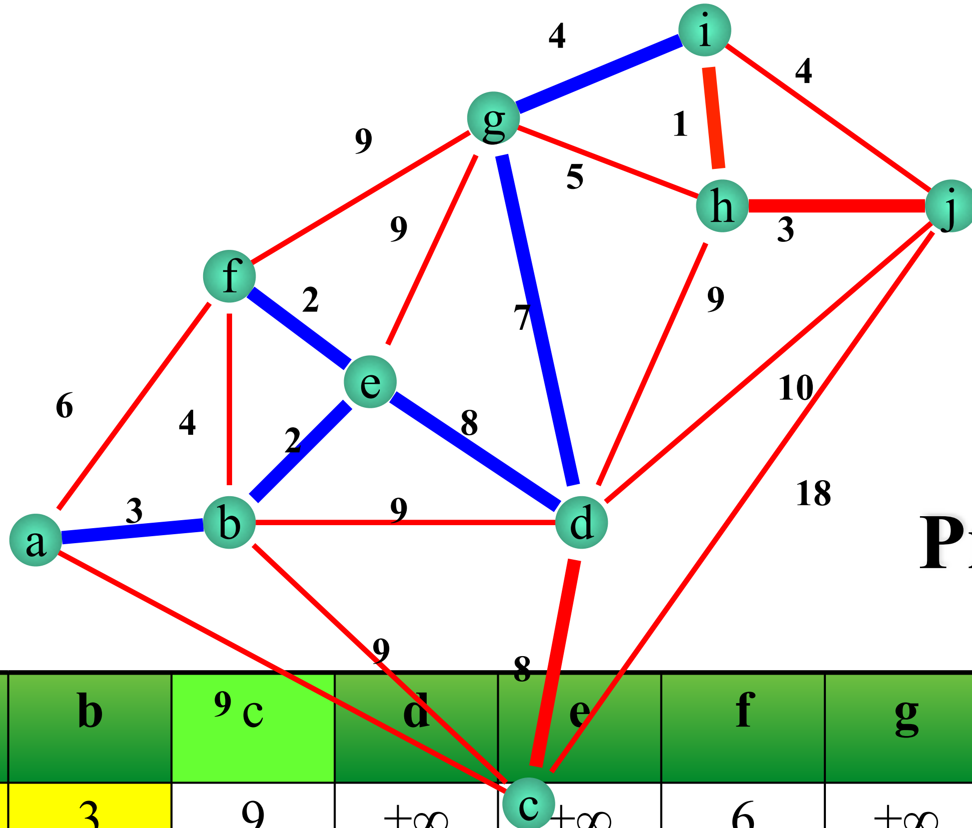
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	5	4	10
0	3	8	8	2	2	7	1	4	4



Prim-Jarník-Dijkstra

D

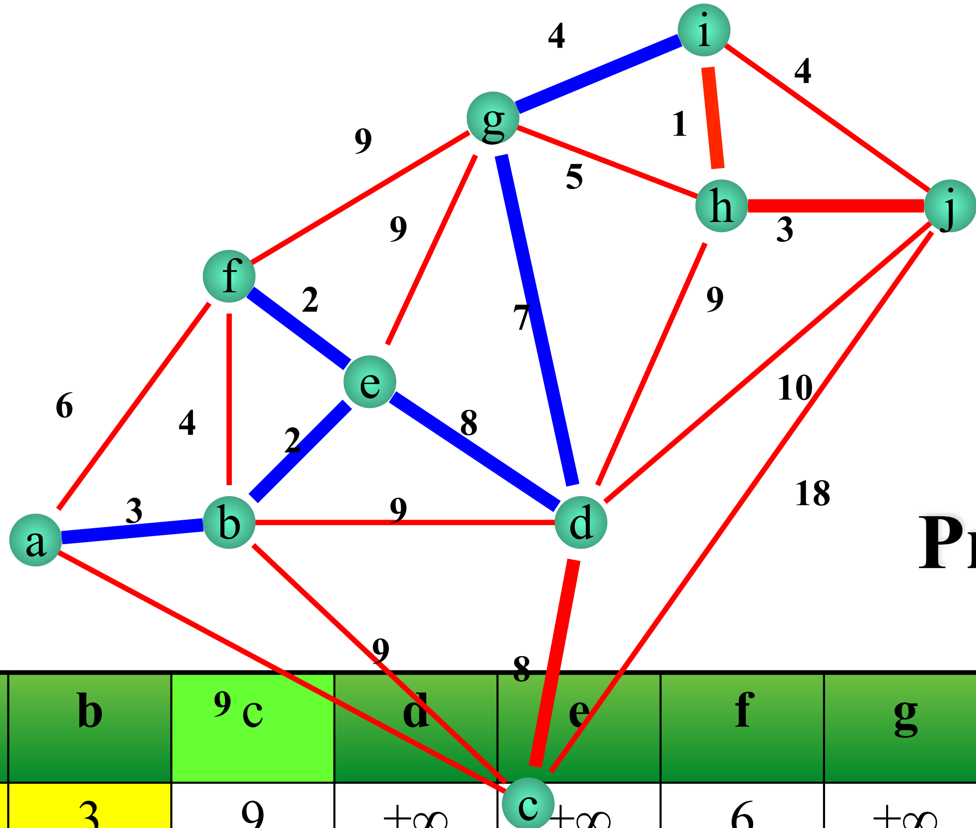
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	5	4	10
0	3	8	8	2	2	7	1	4	4



Prim-Jarník-Dijkstra

D

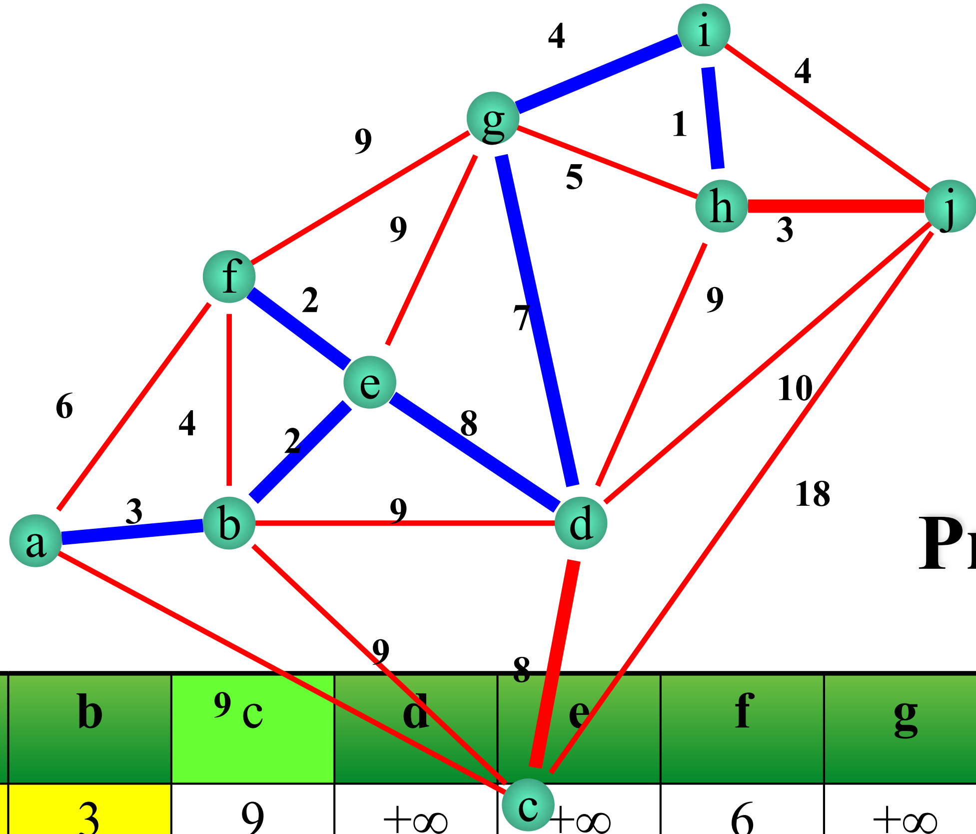
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	5	4	10
0	3	8	8	2	2	7	1	4	4
0	3	8	8	2	2	7	1	4	3



Prim-Jarník-Dijkstra

D

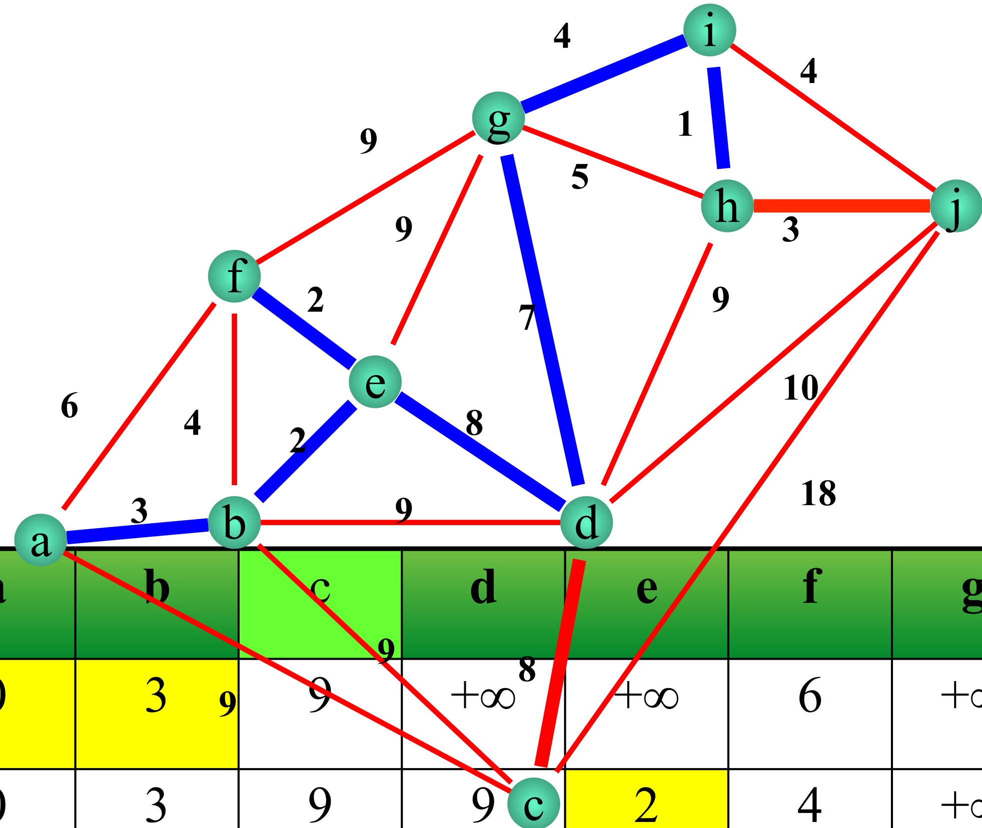
a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	5	4	10
0	3	8	8	2	2	7	1	4	4
0	3	8	8	2	2	7	1	4	3



Prim-Jarník-Dijkstra

D

a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	5	4	10
0	3	8	8	2	2	7	1	4	4
0	3	8	8	2	2	7	1	4	3

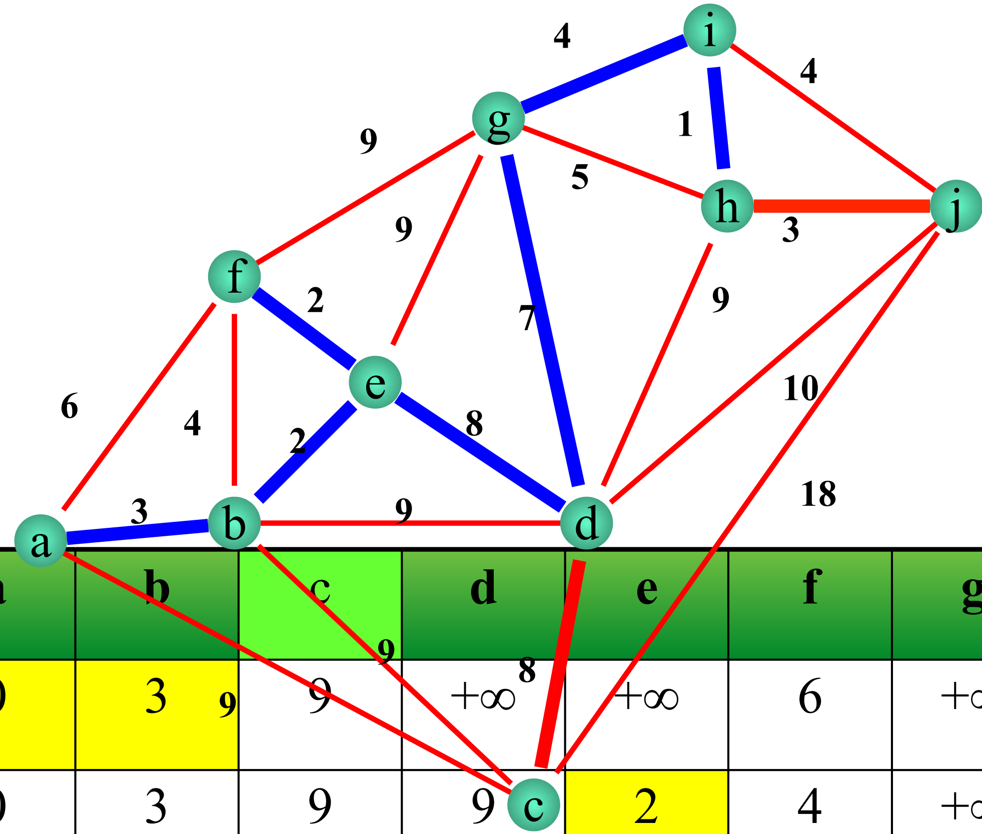


Prim-Jarník-Dijkstra

D

a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	5	4	10
0	3	8	8	2	2	7	1	4	4
0	3	8	8	2	2	7	1	4	3
0	3	8	8	2	2	7	1	4	3

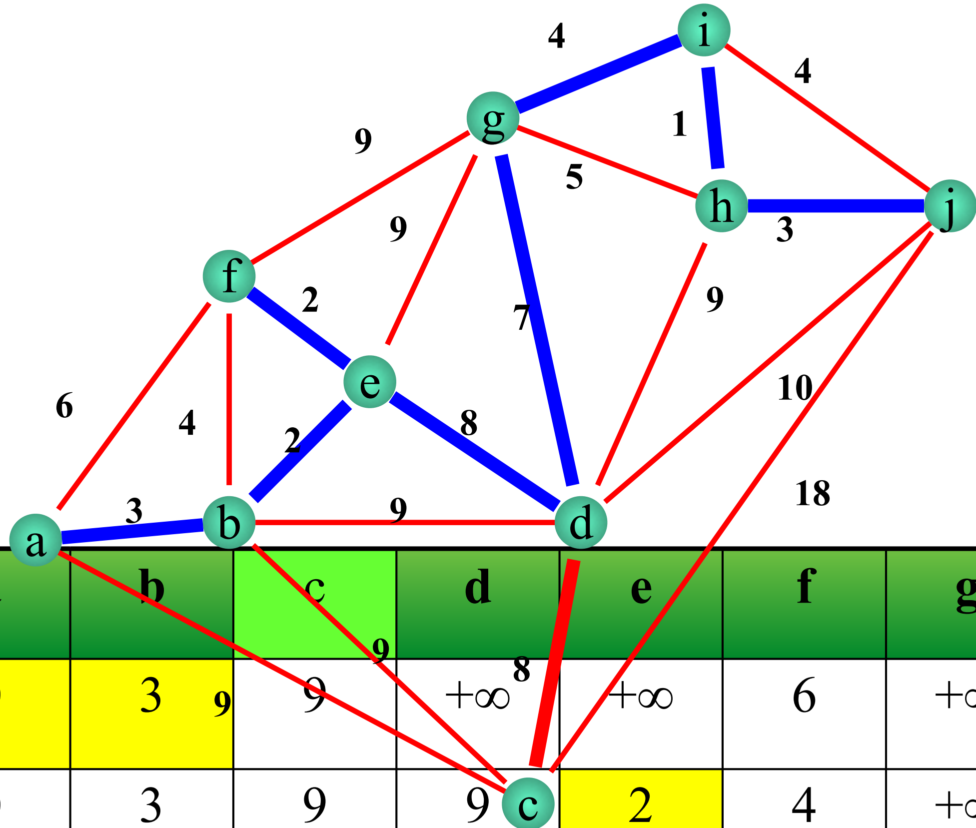
Prim-Jarník-Dijkstra



D

a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	5	4	10
0	3	8	8	2	2	7	1	4	4
0	3	8	8	2	2	7	1	4	3
0	3	8	8	2	2	7	1	4	3

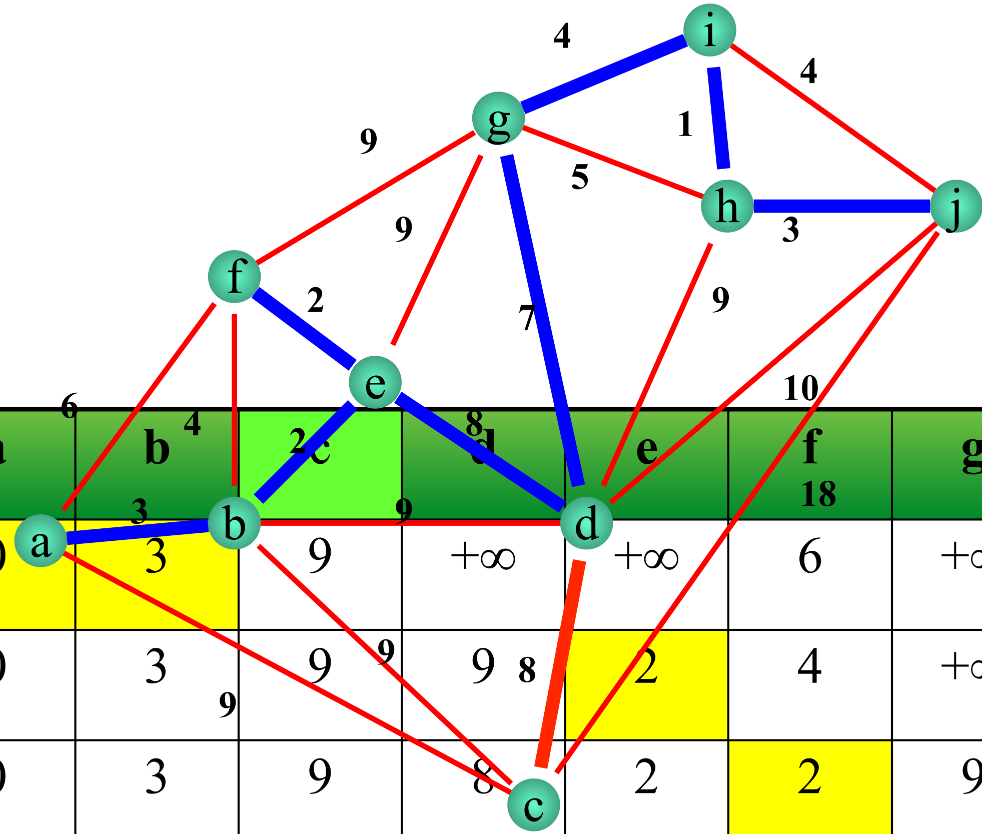
Prim-Jarník-Dijkstra



D

a	**b**	c	**d**	**e**	**f**	**g**	h	**i**	j
0	3	9	$+\infty$	$+\infty$	6	$+\infty$	$+\infty$	$+\infty$	$+\infty$
0	3	9	9	2	4	$+\infty$	$+\infty$	$+\infty$	$+\infty$
0	3	9	8	2	2	9	$+\infty$	$+\infty$	$+\infty$
0	3	8	8	2	2	8	9	$+\infty$	$+\infty$
0	3	8	8	2	2	7	5	4	10
0	3	8	8	2	2	7	1	4	4
0	3	8	8	2	2	7	1	4	3
0	3	8	8	2	2	7	1	4	3

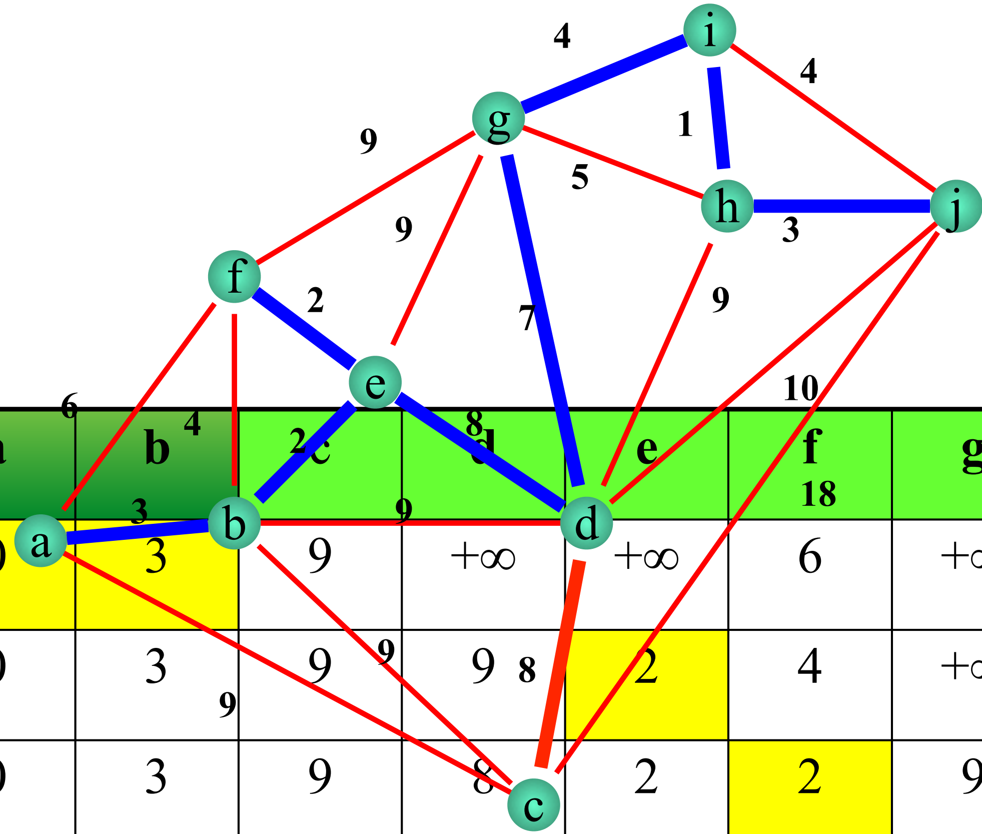
Prim-Jarník-Dijkstra



D

a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	5	4	10
0	3	8	8	2	2	7	1	4	4
0	3	8	8	2	2	7	1	4	3
0	3	8	8	2	2	7	1	4	3
0	3	8	8	2	2	7	1	4	3

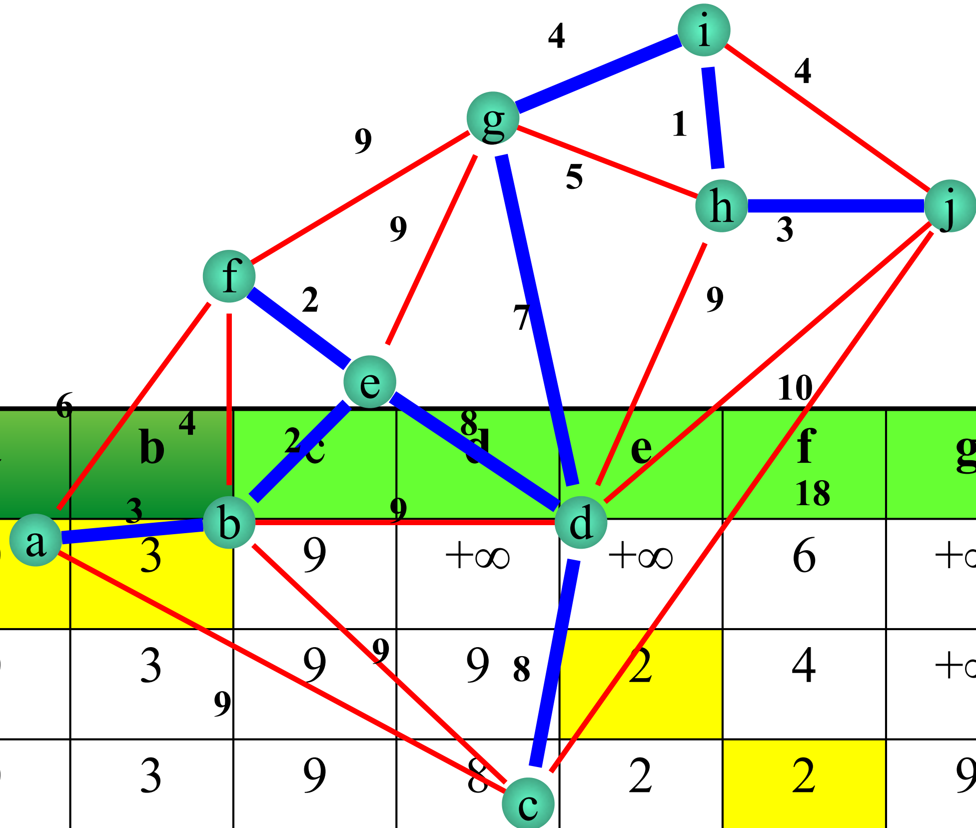
Prim-Jarník-Dijkstra



D

a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	5	4	10
0	3	8	8	2	2	7	1	4	4
0	3	8	8	2	2	7	1	4	3
0	3	8	8	2	2	7	1	4	3
0	3	8	8	2	2	7	1	4	3

Prim-Jarník-Dijkstra



D

a	b	c	d	e	f	g	h	i	j
0	3	9	+∞	+∞	6	+∞	+∞	+∞	+∞
0	3	9	9	2	4	+∞	+∞	+∞	+∞
0	3	9	8	2	2	9	+∞	+∞	+∞
0	3	8	8	2	2	8	9	+∞	+∞
0	3	8	8	2	2	7	5	4	10
0	3	8	8	2	2	7	1	4	4
0	3	8	8	2	2	7	1	4	3
0	3	8	8	2	2	7	1	4	3
0	3	8	8	2	2	7	1	4	3

Pseudocode: Prim's Algorithm

Algorithm Prim-Jarník-Dijkstra
Input: a weighted connected graph $G = (V, E)$
Output: an MST T for G
Data structure: array D; Priority Queue PQ; tree T

pick an arbitrary vertex v in G; $D[v] \leftarrow 0$
for each vertex $u \neq v$ **do** $D[u] \leftarrow +\infty$ **end**
$T \leftarrow \varnothing$
for each vertex u **do** PQ.insert({(u, (null), $D[u]$}) **end** // including v
// for each vertex u, (u, edge) is the element and $D[u]$ is the key in PQ
while not PQ.empty() **do**
  $(u, e) \leftarrow PQ$.deleteMin()
  add vertex u and edge e to T
  for each vertex z adjacent to u such that z is in PQ **do**
    if weight((u, z)) $< D[z]$ **then**
      $D[z] \leftarrow$ weight((u, z))
      in PQ, change element and key of z to {z, (u,z), $D[z]$}
      update PQ
    end
  end
end
return T

D: distance vector, maintains reachable vertices

PQ: priority queue (heap) for the edges, according to their values in D

Prim-Jarník Time Complexity

Theorem. The Prim-Jarník algorithm constructs a minimum spanning tree for a connected weighted graph $G = (V, E)$ with n vertices and m edges in $O(m \log(n))$ time.

Borůvka's Algorithm

Algorithm Borůvka
Input: a weighted connected graph $G = (V, E)$, all edge weights pairwise distinct
Output: an MST T for G
Data structure: Priority Queue $PQ;$ tree T

let T be a subgraph of G initially containing just the vertices in V
for each edge e in E **do** PQ.insert(e) **end**
while T has fewer than n-1 edges **do**
  for each connected component C_k in T **do**
    $e = (v, u) \leftarrow PQ$.deleteMin() with $v \in C_k$ and $u \notin C_k$
    add e to T unless e is already in T
  end
end
return T

Time Complexity

- Every stage of the algorithm divides number of trees by two
- No more than $O(\log(n))$ stages
- A stage may take $O(m)$ time
- Total: $O(m \log(n))$

Borůvka's Algorithm

Time complexity analysis

Algorithm Borůvka
Input: a weighted connected graph $G = (V, E)$
Output: an MST T for G
Data structure: Priority Queue PQ; tree T

let T be a subgraph of G initially containing just the vertices in V
for each edge e in E **do** PQ.insert(e) **end**
while T has fewer than n-1 edges **do**
  for each connected component C_k in T **do**
    $e = (v, u) \leftarrow PQ$.deleteMin() with $v \in C_k$ and $u \notin C_k$
    add e to T unless e is already in T
  end
end
return T

$O(\log n)$

$O(m)$

Total $O(m \log n)$

More on Implementation (Borůvka’s Algorithm)

- possible using union-find data structure

Implementation Considerations

- All three algorithms: same worst-case running time
- each uses different data structures/different approaches
- Kruskal's algorithm uses priority queue to store edges, and union-find data structure, to store clusters
- Prim-Jarník's algorithm is similar to implement as Dijkstra's single-source shortest-path algorithm (for the ones who know Dijkstra's algorithm already)
- Borůvka's algorithm is also easy to implement and stores connected components
- There is no clear winner with respect to best constant