Minimum Spanning Trees

Prim Kruskal NP-complete problems

Lecturer: Georgy Gimel'farb

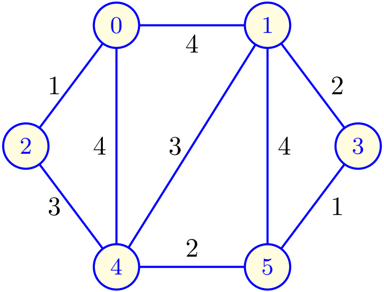
COMPSCI 220 Algorithms and Data Structures

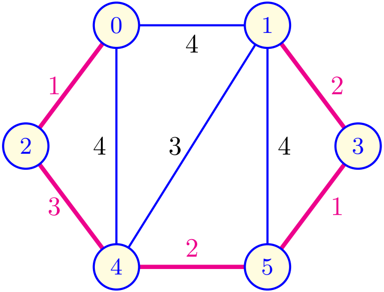
1. Minimum spanning tree problem
2. Prim's MST Algorithm
3. Kruskal's MST algorithm
4. Other graph/network optimisation problems

Minimum Spanning Tree

Minimum spanning tree (MST) of a weighted graph G:

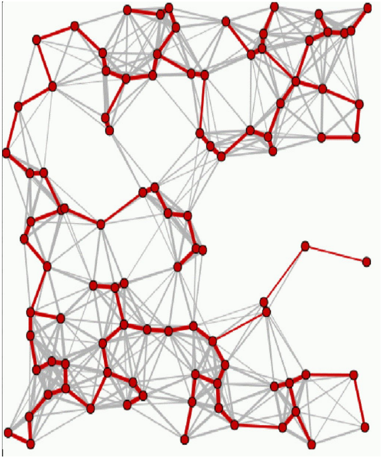
A **spanning tree**, i.e., a subgraph, being a tree and containing all vertices, having minimum total weight (sum of all edge weights).





The **MST** with the total weight 9

Finding Minimum Spanning Tree



Many applications:

- Electrical, communication, road etc network design.
- Data coding and clustering.
- Approximate NP-complete graph optimisation.
  - Travelling salesman problem: the MST is within a factor of two of the optimal path.
- Image analysis.

http://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/

Finding Minimum Spanning Tree

Two efficient **greedy** Prim's and Kruskal's MST algorithms:

- Each algorithm selects edges in order of their increasing weight, but avoids creating a cycle.
- The Prim's algorithm maintains a tree at each stage that grows to span.
- The Kruskal's algorithm maintains a forest whose trees coalesce into one spanning tree.
- The Prim's algorithm implemented with a priority queue is very similar to the Dijkstra's algorithm.
  - This implementation of the Prim's algorithm runs in time $\mathrm{O}(m + n \log n)$.
- The Kruskal's algorithm uses disjoint sets ADT and can be implemented to run in time $\mathrm{O}(m \log n)$.

Finding Minimum Spanning Tree

Two efficient **greedy** Prim's and Kruskal's MST algorithms:

- Each algorithm selects edges in order of their increasing weight, but avoids creating a cycle.
- The Prim's algorithm maintains a tree at each stage that grows to span.
- The Kruskal's algorithm maintains a forest whose trees coalesce into one spanning tree.
- The Prim's algorithm implemented with a priority queue is very similar to the Dijkstra's algorithm.
  - This implementation of the Prim's algorithm runs in time $\mathrm{O}(m + n \log n)$.
- The Kruskal's algorithm uses disjoint sets ADT and can be implemented to run in time $\mathrm{O}(m \log n)$.

Finding Minimum Spanning Tree

Two efficient **greedy** Prim's and Kruskal's MST algorithms:

- Each algorithm selects edges in order of their increasing weight, but avoids creating a cycle.
- The Prim's algorithm maintains a tree at each stage that grows to span.
- The Kruskal's algorithm maintains a forest whose trees coalesce into one spanning tree.
- The Prim's algorithm implemented with a priority queue is very similar to the Dijkstra's algorithm.
  - This implementation of the Prim's algorithm runs in time $\mathrm{O}(m + n \log n)$.
- The Kruskal's algorithm uses disjoint sets ADT and can be implemented to run in time $\mathrm{O}(m \log n)$.

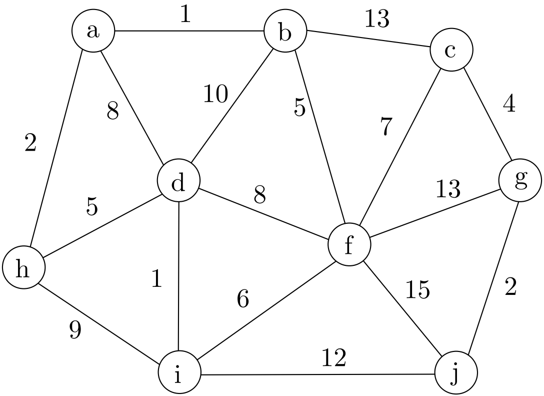
Prim's MST Algorithm

algorithm `Prim`(weighted graph (G, c), vertex s)
    array $w[n] = \{c[s,0], c[s,1], \ldots, c[s,n-1]\}$
    $S \leftarrow \{s\}$ first vertex added to MST
    while $S \neq V(G)$ **do**
        find $u \in V(G) \setminus S$ so that $w[u]$ is minimum
        $S \leftarrow S \cup \{u\}$ adding an edge adjacent to u to MST
        for $x \in V(G) \setminus S$ **do**
            $w[x] \leftarrow \min\{w[x], c[u,x]\}$
        end for
    end while

Very similar to the Dijkstra's algorithm:

- Priority queue should be used for selecting the lowest edge weights $w[\ldots]$.
- In the priority queue implementation, most time is taken by EXTRACT-MIN and DECREASE-KEY operations.

Illustrating the Prim's Algorithm

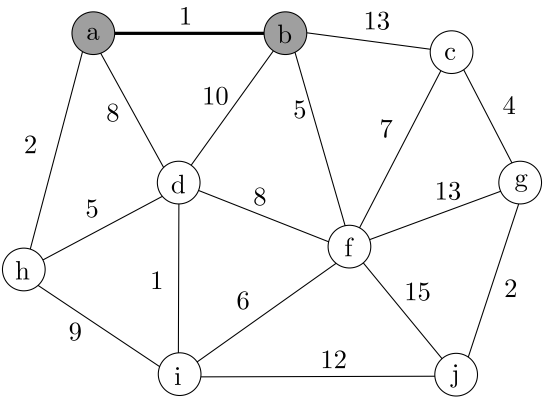


$S = \{a\}$

$w =$

a	b	c	d	f	g	h	i	j
0	1	∞	8	∞	∞	2	∞	∞

Illustrating the Prim's Algorithm

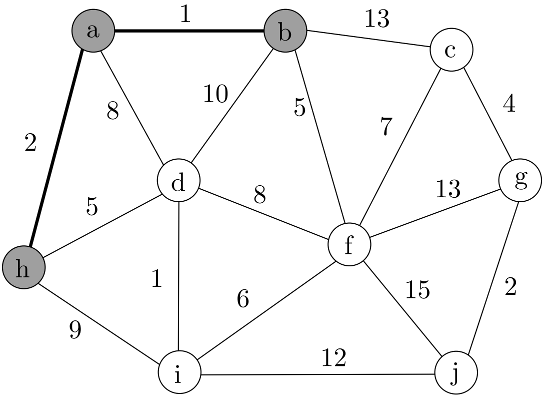


$S = \{a,b\}$

$w =$

a	b	c	d	f	g	h	i	j
0	1_a	13_b	8_a	5_b	∞	2_a	∞	∞

Illustrating the Prim's Algorithm

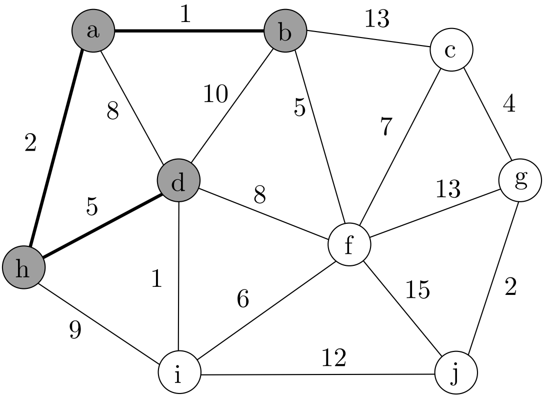


$S = \{a,b,h\}$

$w =$

a	b	c	d	f	g	h	i	j
0	1_a	13_b	5_h	5_b	∞	2_a	9_h	∞

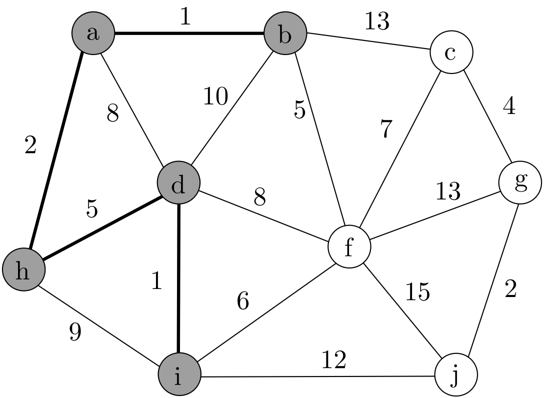
Illustrating the Prim's Algorithm



$S = \{a, b, d, h\}$ $w =$

a	b	c	d	f	g	h	i	j
0	1_a	13_b	5_h	5_b	∞	2_a	1_d	∞

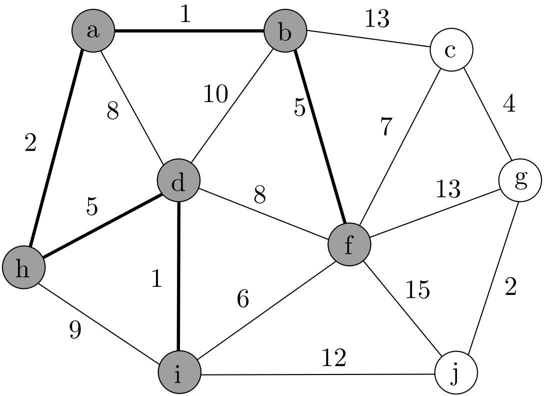
Illustrating the Prim's Algorithm



$S = \{a,b,d,h,i\}$ $w =$

a	b	c	d	f	g	h	i	j
0	1_a	13_b	5_h	5_b	∞	2_a	1_d	12_i

Illustrating the Prim's Algorithm

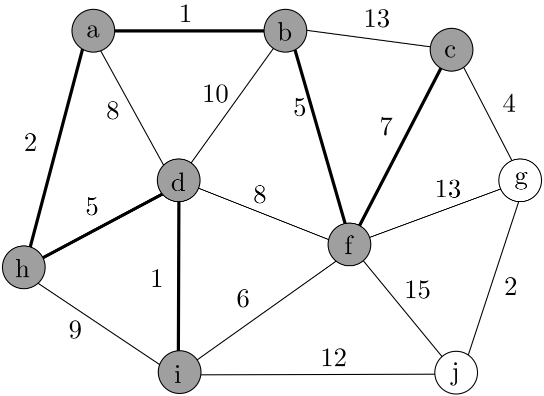


$S = \{a,b,d,f,h,i\}$

$w =$

a	b	c	d	f	g	h	i	j
0	1_a	7_f	5_h	5_b	13_f	2_a	1_d	12_i

Illustrating the Prim's Algorithm

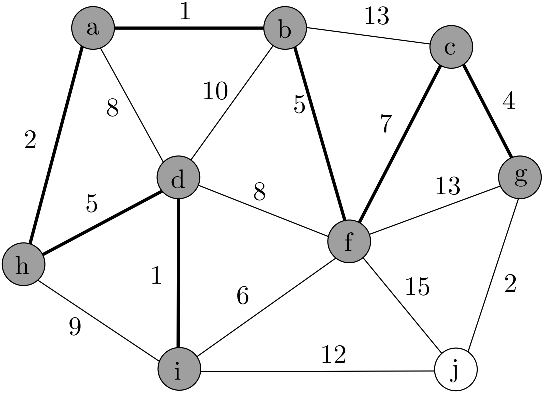


$S = \{a,b,c,d,f,h,i\}$

$w =$

a	b	c	d	f	g	h	i	j
0	1_a	7_f	5_h	5_b	4_c	2_a	1_d	12_i

Illustrating the Prim's Algorithm

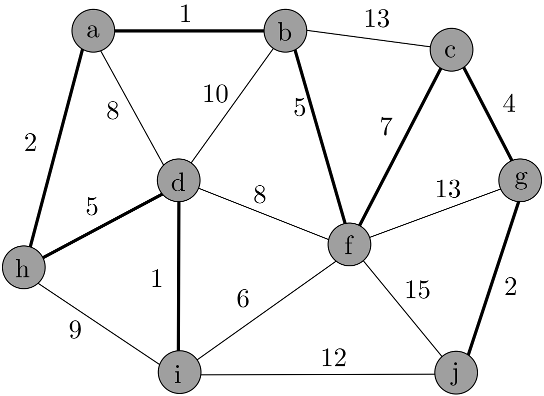


$S = \{a, b, c, d, f, g, h, i\}$

$w =$

a	b	c	d	f	g	h	i	j
0	1_a	7_f	5_h	5_b	4_c	2_a	1_d	2_g

Illustrating the Prim's Algorithm



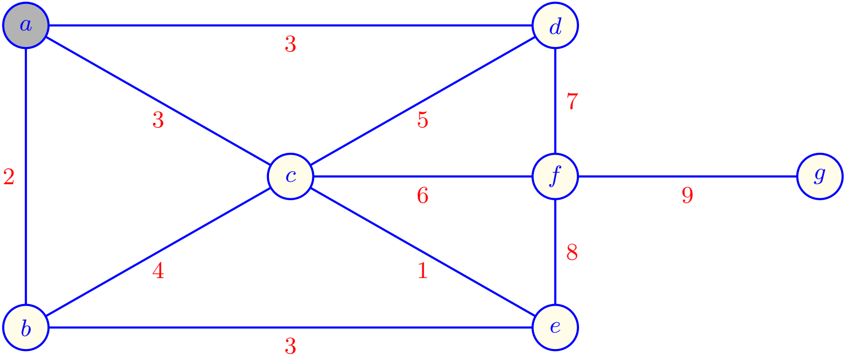
$S = \{a,b,c,d,f,g,h,i,j\}$ $w =$

a	b	c	d	f	g	h	i	j
0	1_a	7_f	5_h	5_b	4_c	2_a	1_d	2_g

Prim's Algorithm (Priority Queue Implementation)

algorithm `Prim` (weighted graph (G, c), vertex $s \in V(G)$)
  priority queue Q, arrays $colour[n]$, $pred[n]$
  for $u \in V(G)$ **do**
    $pred[u] \leftarrow$ NULL; $colour[u] \leftarrow$ WHITE
  end for
  $colour[s] \leftarrow$ GREY; $Q.\mathtt{insert}(s, 0)$
  while not $Q.\mathtt{isEmpty}()$ **do**
    $u \leftarrow Q.\mathtt{peek}()$
    for each x adjacent to u **do**
      $t \leftarrow c(u, x)$;
      if $colour[x] =$ WHITE **then**
        $colour[x] \leftarrow$ GREY; $pred[x] \leftarrow u$; $Q.\mathtt{insert}(x, t)$
      else if $colour[x] =$ GREY **and** $Q.\mathtt{getKey}(x) > t$ **then**
        $Q.\mathtt{decreaseKey}(x, t)$; $pred[x] \leftarrow u$
      end if
    end for
    $Q.\mathtt{delete}()$; $colour[u] \leftarrow$ BLACK
  end while
  return $pred$

MST: Prim's Algorithm – Starting Vertex a

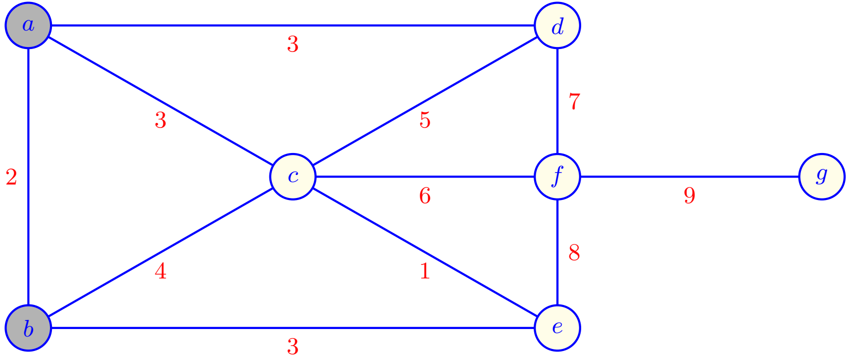


Initialisation:

Priority queue Q: $\{a_{\text{key}=0}\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	—	—	—	—	—	—	—
key_v	0						

MST: Prim's Algorithm: 1 – 2 $u \leftarrow a = Q.\texttt{peek}()$; **for**-loop



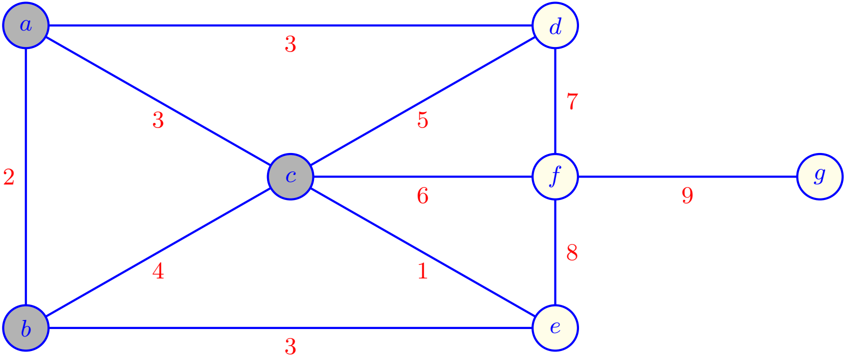
$u = a$; adjacent $x \in \{b, c, d\}$; $x \leftarrow b$; $\text{key}_b \leftarrow \text{cost}(a, b) = 2$

Priority queue Q: $\{a_0, b_2\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	–	–	–	–	–
key_v	0	2					

MST: Prim's Algorithm: 3

$u = a$; **for**-loop



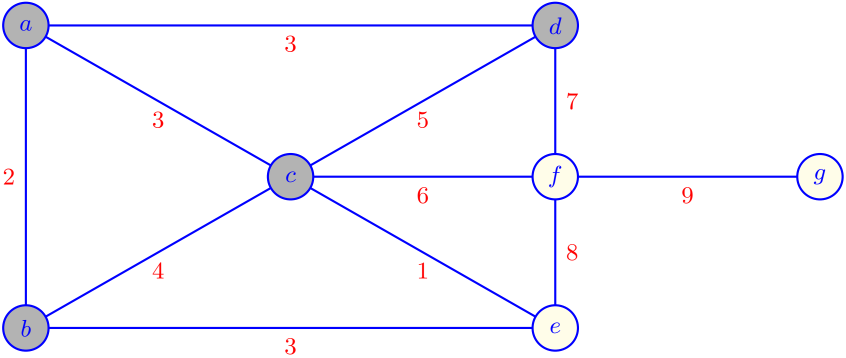
adjacent $x \in \{b, c, d\}$; $x \leftarrow c$; $\text{key}_c \leftarrow \text{cost}(a, c) = 3$

Priority queue Q: $\{a_0, b_2, c_3\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	—	a	a	—	—	—	—
key_v	0	2	3				

MST: Prim's Algorithm: 4

$u = a$; **for**-loop



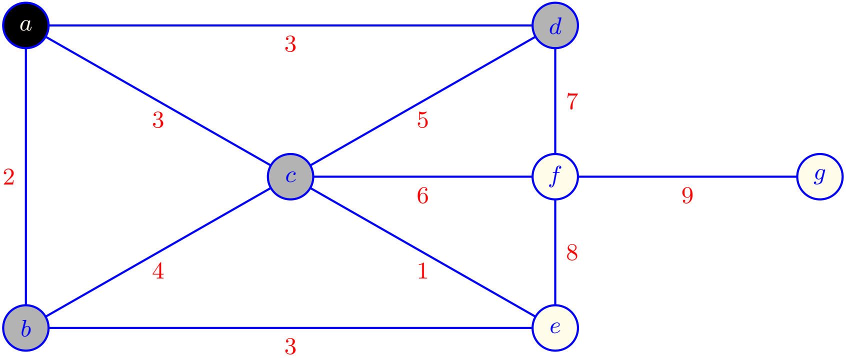
adjacent $x \in \{b, c, d\}$; $x \leftarrow d$; $\text{key}_d \leftarrow \text{cost}(a, d) = 3$

Priority queue Q: $\{a_0, b_2, c_3, d_3\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	—	a	a	a	—	—	—
key_v	0	2	3	3			

MST: Prim's Algorithm: 5 – 6

Q.delete()



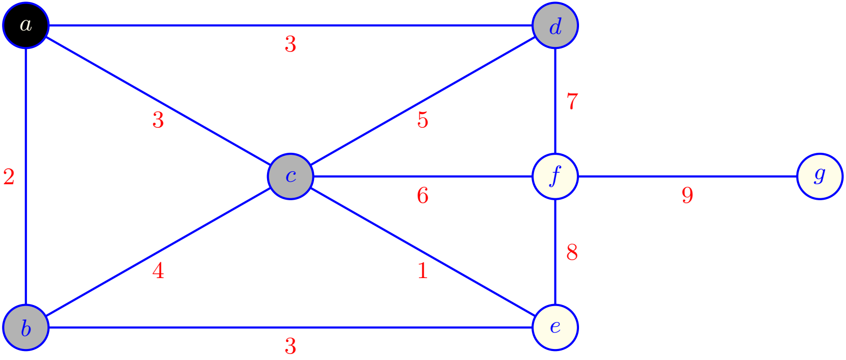
Q.delete() – excluding the vertex a

Priority queue Q: $\{b_2, c_3, d_3\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	a	a	–	–	–
key$_v$	0	2	3	3			

MST: Prim's Algorithm: 7

$u \leftarrow b = Q.\texttt{peek}()$; **for**-loop

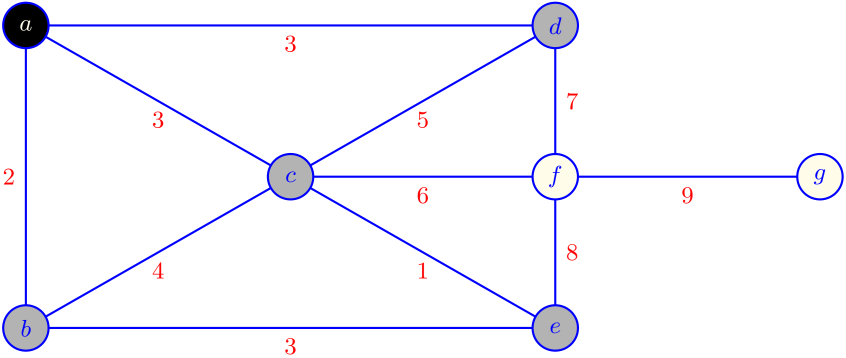


adjacent $x \in \{c, e\}$; $x \leftarrow c$; $\text{key}_c = 3 < \text{cost}(b, c) = 4$

Priority queue Q: $\{b_2, c_3, d_3\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	—	a	a	a	—	—	—
key_v	0	2	3	3			

MST: Prim's Algorithm: 8 $u = b$; **for**-loop

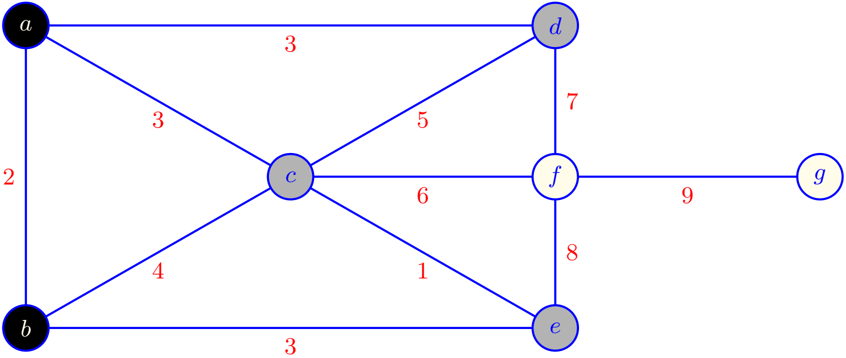


adjacent $x \in \{c, e\}$; $x \leftarrow e$; $\text{key}_e \leftarrow \text{cost}(b, e) = 3$

Priority queue Q: $\{b_2, c_3, d_3, e_3\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	—	a	a	a	b	—	—
key_v	0	2	3	3	3		

MST: Prim's Algorithm: 9 – 10 Q.delete()



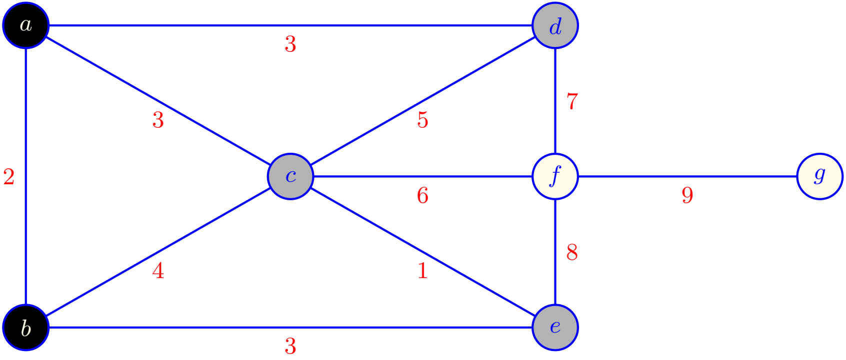
Q.delete() – excluding the vertex b

Priority queue Q: $\{c_3, d_3, e_3\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	a	a	b	–	–
key_v	0	2	3	3	3		

MST: Prim's Algorithm: 11

$u \leftarrow c = Q.\texttt{peek}()$; **for**-loop



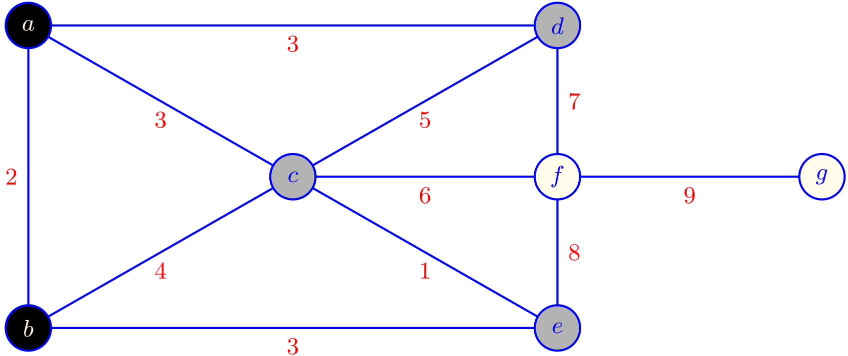
adjacent $x \in \{d, e, f\}$; $x \leftarrow d$; $\text{key}_d = 3 < \text{cost}(c, d) = 5$

Priority queue Q: $\{c_3, d_3, e_3\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	a	a	b	–	–
key_v	0	2	3	3	3		

MST: Prim's Algorithm: 12

$u = c$; **for**-loop



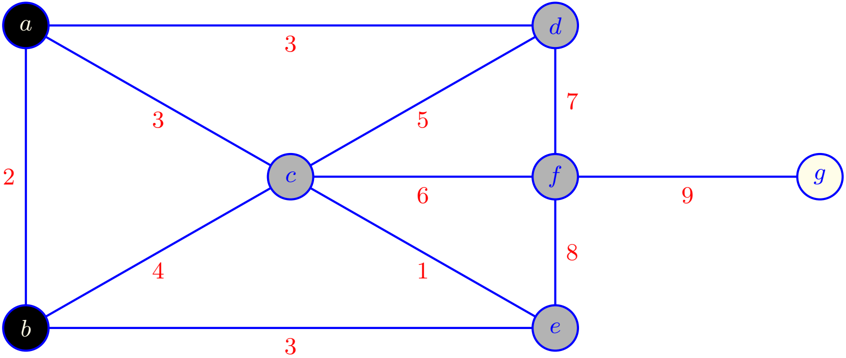
adjacent $x \in \{d, e, f\}$; $x \leftarrow e$; $\underline{\text{key}_e = 3 > \text{cost}(c, e) = 1}$; $\text{key}_e \leftarrow 1$

Priority queue Q: $\{c_3, d_3, e_1\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	a	a	c	–	–
key_v	0	2	3	3	1		

MST: Prim's Algorithm: 13

$u = c$; **for**-loop



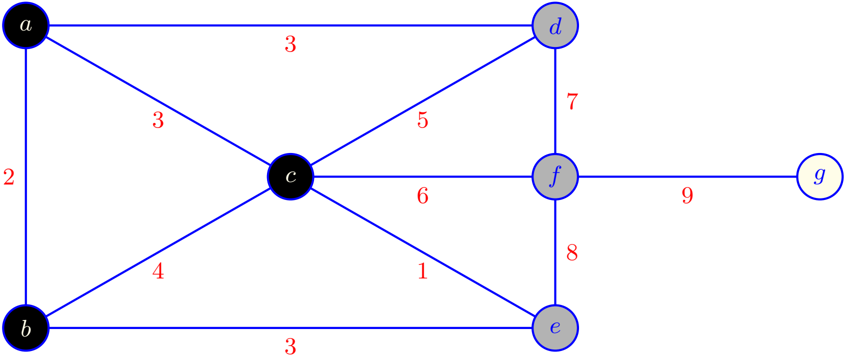
adjacent $x \in \{d, e, f\}$; $x \leftarrow f$; $\text{key}_f \leftarrow 6$

Priority queue Q: $\{c_3, d_3, e_1, f_6\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	a	a	c	c	–
key_v	0	2	3	3	1	6	

MST: Prim's Algorithm: 14 – 15

Q.delete()



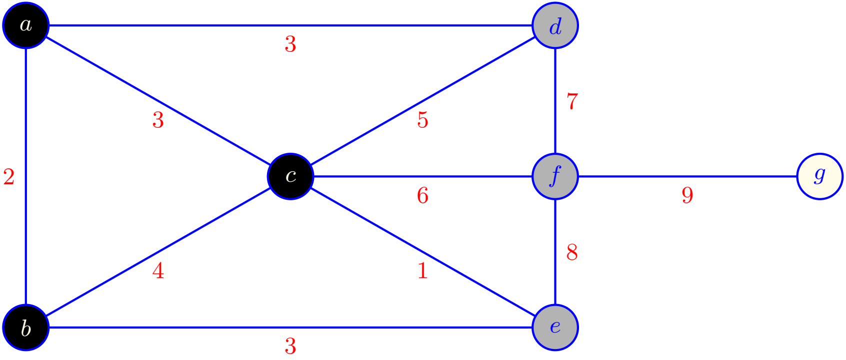
Q.delete() – excluding the vertex c

Priority queue Q: $\{e_1, d_3, f_6\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	a	a	c	c	–
key_v	0	2	3	3	1	6	

MST: Prim's Algorithm: 16

$u \leftarrow e = Q.\mathtt{peek}()$; **for**-loop

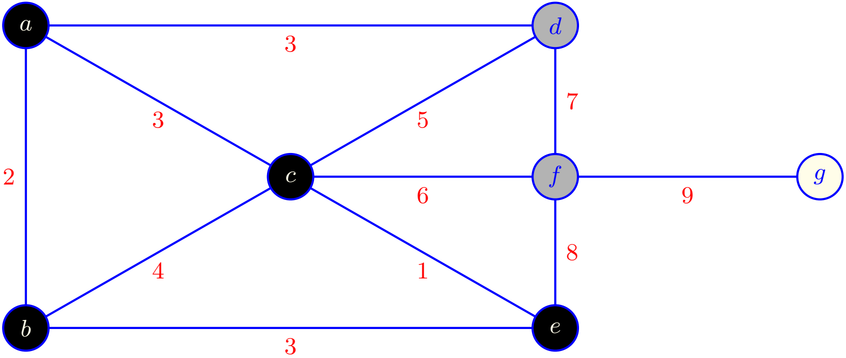


adjacent $x \in \{f\}$; $x \leftarrow f$; $\mathrm{key}_f \leftarrow 6 < \mathrm{cost}(e, f) = 8$

Priority queue Q: $\{e_1, d_3, f_6\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	a	a	c	c	–
key_v	0	2	3	3	1	6	

MST: Prim's Algorithm: 17 – 18 Q.delete()

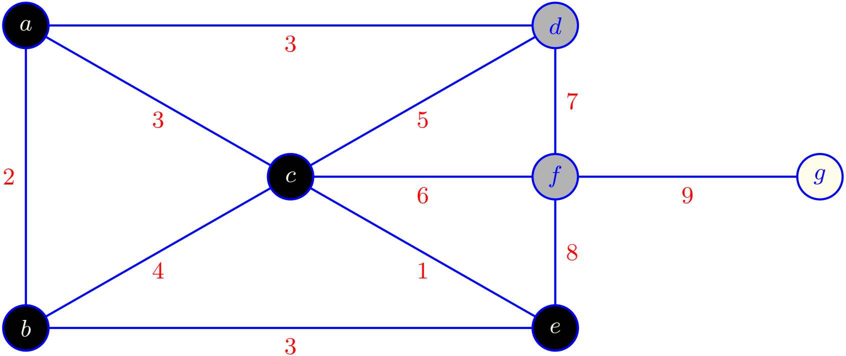


Q.delete() – excluding the vertex e

Priority queue Q: $\{d_3, f_6\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	a	a	c	c	–
key$_v$	0	2	3	3	1	6	

MST: Prim's Algorithm: 19 $u \leftarrow d = Q.\texttt{peek}()$; **for**-loop



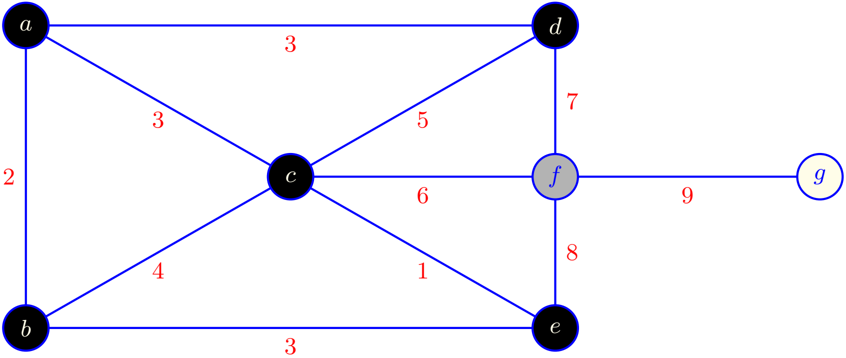
adjacent $x \in \{f\}$; $x \leftarrow f$; $\text{key}_f \leftarrow 6 < \text{cost}(d, f) = 7$

Priority queue Q: $\{d_3, f_6\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	a	a	c	c	–
key_v	0	2	3	3	1	6	

MST: Prim's Algorithm: 20 – 21

Q.delete()

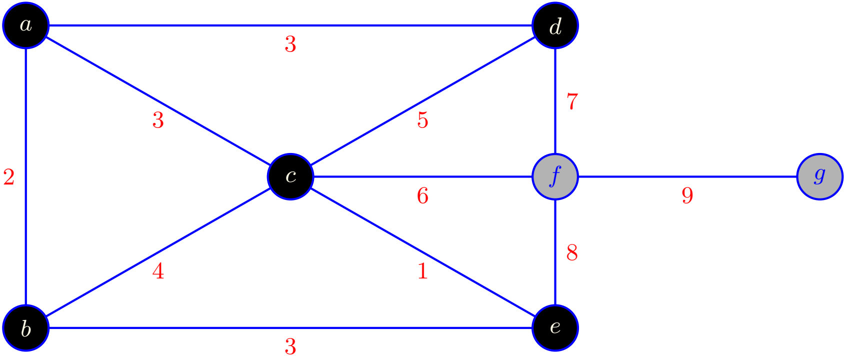


Q.delete() – excluding the vertex d

Priority queue Q: $\{f_6\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	—	a	a	a	c	c	—
key_v	0	2	3	3	1	6	

MST: Prim's Algorithm: 22 $u \leftarrow f = Q.\texttt{peek}()$; **for**-loop



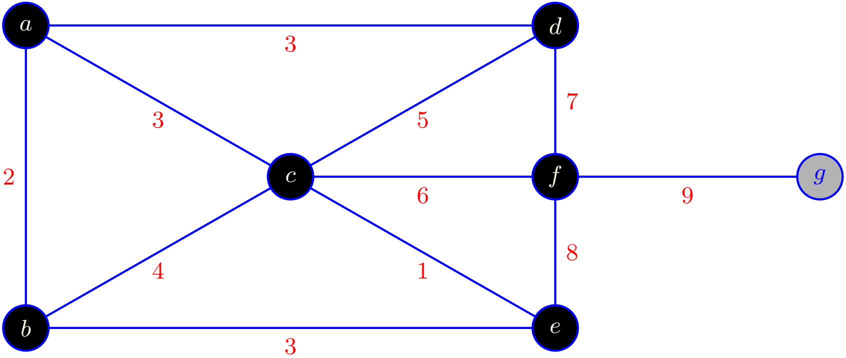
adjacent $x \in \{g\}$; $x \leftarrow g$; $\text{key}_g \leftarrow \text{cost}(f, g) = 9$

Priority queue Q: $\{f_6, g_9\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	$-$	a	a	a	c	c	f
key_v	0	2	3	3	1	6	9

MST: Prim's Algorithm: 23

Q.delete()

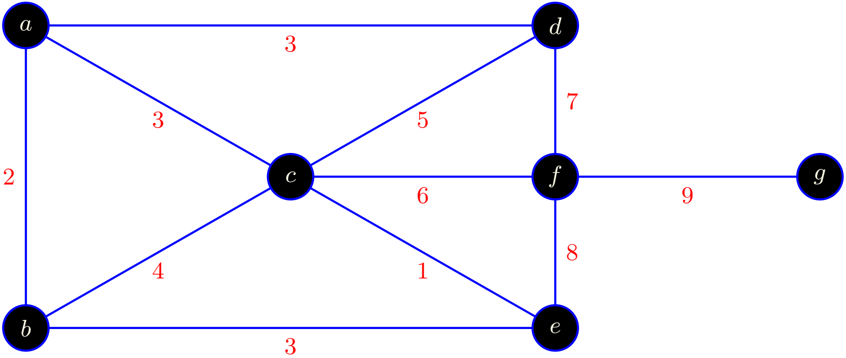


Q.delete() – excluding the vertex f

Priority queue Q: $\{g_9\}$

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	a	a	c	c	f
key$_v$	0	2	3	3	1	6	9

MST: Prim's Algorithm: 24 – 25 $u \leftarrow g = Q.\texttt{peek}()$

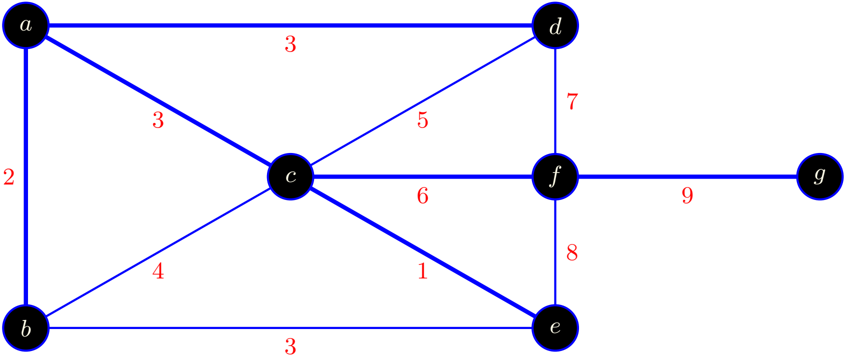


no adjacent vertices x; $Q.\texttt{delete}()$ – excluding the vertex g

Priority queue Q: empty

$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	a	a	c	c	f
key_v	0	2	3	3	1	6	9

MST: Prim's Algorithm: Output



MST edges $\{e = (pred[v], v) : v \in V \backslash a\}$:

$\{(a,b), (a,c), (a,d), (c,e), (c,f), (f,g)\}$; total cost 24

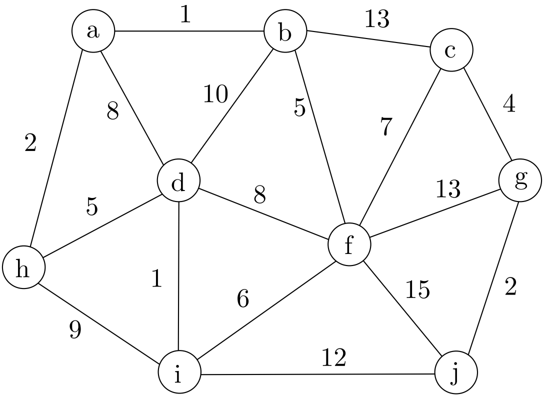
$v \in V$	a	b	c	d	e	f	g
$pred[v]$	–	a	a	a	c	c	f
key_v	0	2	3	3	1	6	9

Kruskal's MST Algorithm

algorithm `Kruskal`(weighted graph (G, c))
$T \leftarrow \emptyset$
insert $E(G)$ into a priority queue
for $e = \{u, v\} \in E(G)$ in increasing order of weight **do**
    if u and v are not in the same tree **then**
        $T \leftarrow T \cup \{e\}$
        merge the trees of u and v
    end if
end for

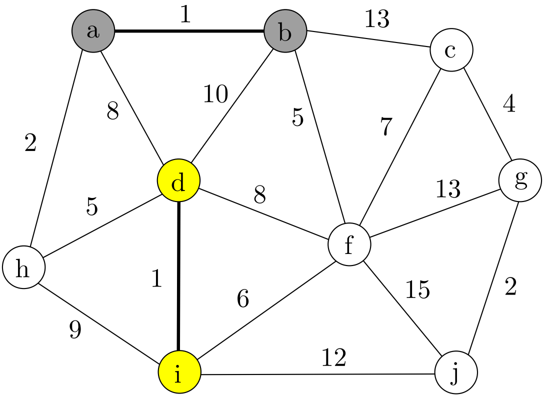
- Keeping track of the trees using the disjoint sets ADT, with standard operations FIND and UNION.
- They can be implemented efficiently so that the main time taken is at the sorting step.

Illustrating Kruskal's Algorithm



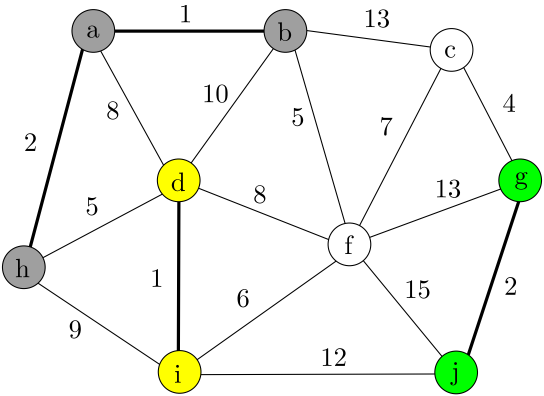
e	(a,b)	(d,i)	(a,h)	(j,g)	(c,g)	(d,h)	(b,f)	(f,i)	(c,f)	(a,d)	(d,f)	(h,i)	(b,d)	(i,j)	(b,c)	(f,g)	(f,j)
$c(e)$	1	1	2	2	4	5	5	6	7	8	8	9	10	12	13	13	15

Illustrating Kruskal's Algorithm



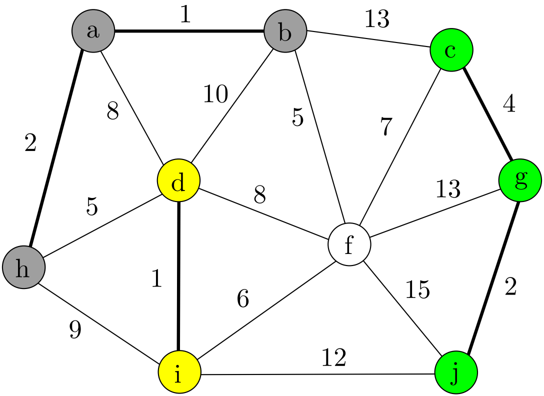
e	(a,b)	(d,i)	(a,h)	(j,g)	(c,g)	(d,h)	(b,f)	(f,i)	(c,f)	(a,d)	(d,f)	(h,i)	(b,d)	(i,j)	(b,c)	(f,g)	(f,j)
$c(e)$	1	1	2	2	4	5	5	6	7	8	8	9	10	12	13	13	15

Illustrating Kruskal's Algorithm



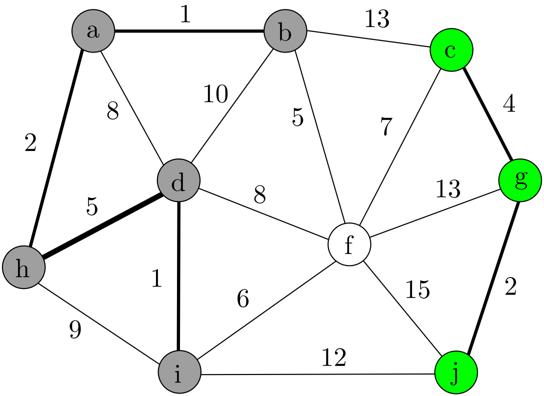
e	(a,b)	(d,i)	(a,h)	(j,g)	(c,g)	(d,h)	(b,f)	(f,i)	(c,f)	(a,d)	(d,f)	(h,i)	(b,d)	(i,j)	(b,c)	(f,g)	(f,j)
$c(e)$	1	1	2	2	4	5	5	6	7	8	8	9	10	12	13	13	15

Illustrating Kruskal's Algorithm



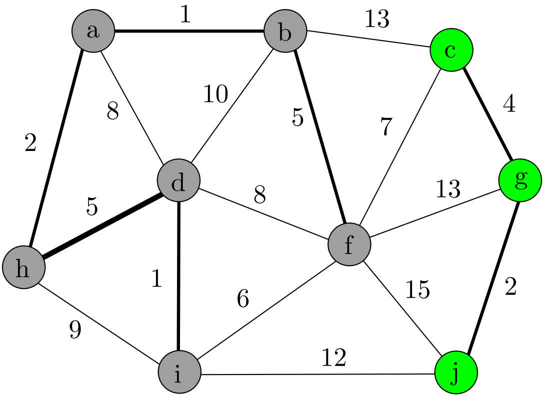
e	(a,b)	(d,i)	(a,h)	(j,g)	(c,g)	(d,h)	(b,f)	(f,i)	(c,f)	(a,d)	(d,f)	(h,i)	(b,d)	(i,j)	(b,c)	(f,g)	(f,j)
$c(e)$	1	1	2	2	4	5	5	6	7	8	8	9	10	12	13	13	15

Illustrating Kruskal's Algorithm



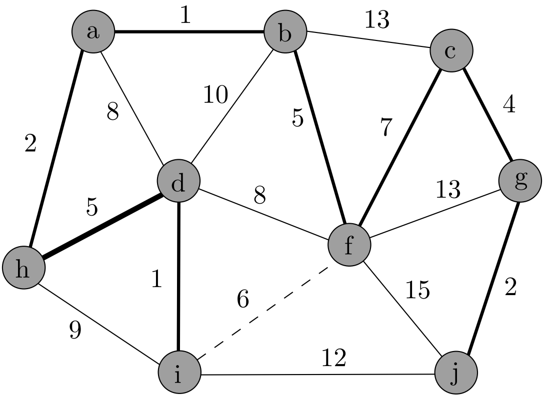
e	(a,b)	(d,i)	(a,h)	(j,g)	(c,g)	(d,h)	(b,f)	(f,i)	(c,f)	(a,d)	(d,f)	(h,i)	(b,d)	(i,j)	(b,c)	(f,g)	(f,j)
$c(e)$	1	1	2	2	4	5	5	6	7	8	8	9	10	12	13	13	15

Illustrating Kruskal's Algorithm



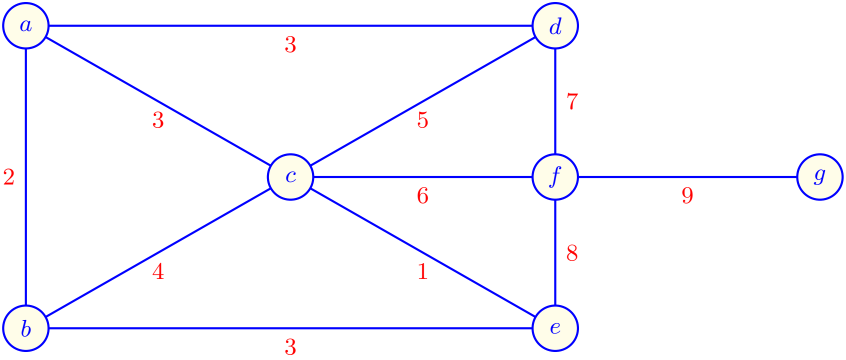
e	(a,b)	(d,i)	(a,h)	(j,g)	(c,g)	(d,h)	(b,f)	(f,i)	(c,f)	(a,d)	(d,f)	(h,i)	(b,d)	(i,j)	(b,c)	(f,g)	(f,j)
$c(e)$	1	1	2	2	4	5	5	6	7	8	8	9	10	12	13	13	15

Illustrating Kruskal's Algorithm



e	(a,b)	(d,i)	(a,h)	(j,g)	(c,g)	(d,h)	(b,f)	(f,i)	(c,f)	(a,d)	(d,f)	(h,i)	(b,d)	(i,j)	(b,c)	(f,g)	(f,j)
$c(e)$	1	1	2	2	4	5	5	6	7	8	8	9	10	12	13	13	15

MST: Kruskal's Algorithm

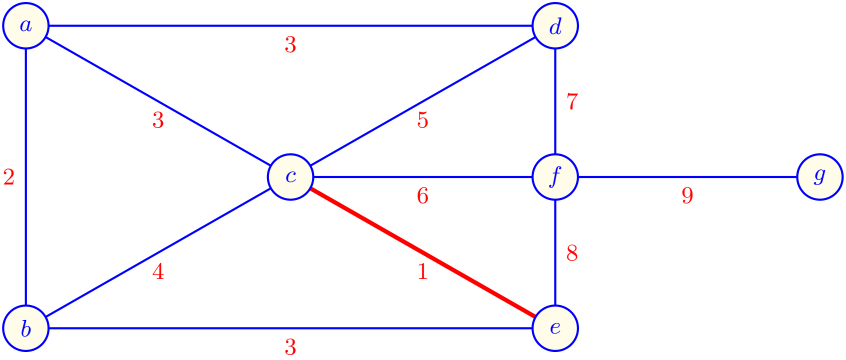


Initialisation:

Disjoint-sets ADT $A = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c,e)	(a,b)	(a,c)	(a,d)	(b,e)	(b,c)	(c,d)	(c,f)	(d,f)	(e,f)	(f,g)

MST: Kruskal's Algorithm 1

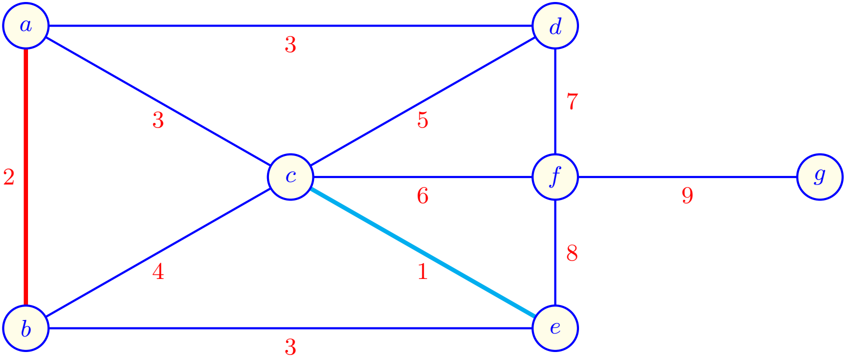


Step 1: $\{S_c = A.\texttt{set}(c)\} \neq \{S_e = A.\texttt{set}(e)\}$; **add** (c, e); $A.\texttt{union}(S_c, S_e)$

Disjoint-sets ADT $A = \{\{a\}, \{b\}, \{c, e\}, \{d\}, \{f\}, \{g\}\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c, e)	(a, b)	(a, c)	(a, d)	(b, e)	(b, c)	(c, d)	(c, f)	(d, f)	(e, f)	(f, g)

MST: Kruskal's Algorithm 2

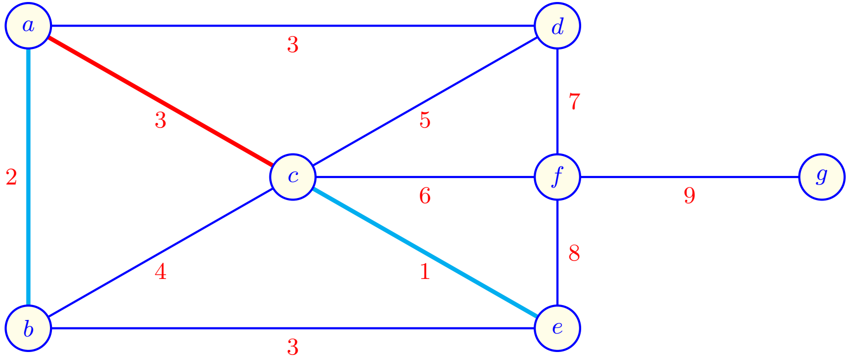


Step 2: $\{S_a = A.\mathtt{set}(a)\} \neq \{S_b = A.\mathtt{set}(b)\}$; **add** (a, b); $A.\mathtt{union}(S_a, S_b)$

Disjoint-sets ADT $A = \big\{\{a, b\}, \{c, e\}, \{d\}, \{f\}, \{g\}\big\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c, e)	(a, b)	(a, c)	(a, d)	(b, e)	(b, c)	(c, d)	(c, f)	(d, f)	(e, f)	(f, g)

MST: Kruskal's Algorithm 3

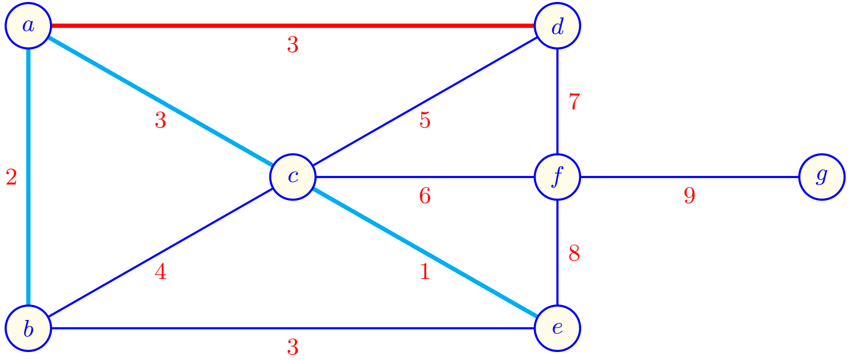


Step 3: $\{S_a = A.\mathtt{set}(a)\} \neq \{S_c = A.\mathtt{set}(c)\}$; **add** (a, c); $A.\mathtt{union}(S_a, S_c)$

Disjoint-sets ADT $A = \{\{a, b, c, e\}, \{d\}, \{f\}, \{g\}\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c, e)	(a, b)	(a, c)	(a, d)	(b, e)	(b, c)	(c, d)	(c, f)	(d, f)	(e, f)	(f, g)

MST: Kruskal's Algorithm 4

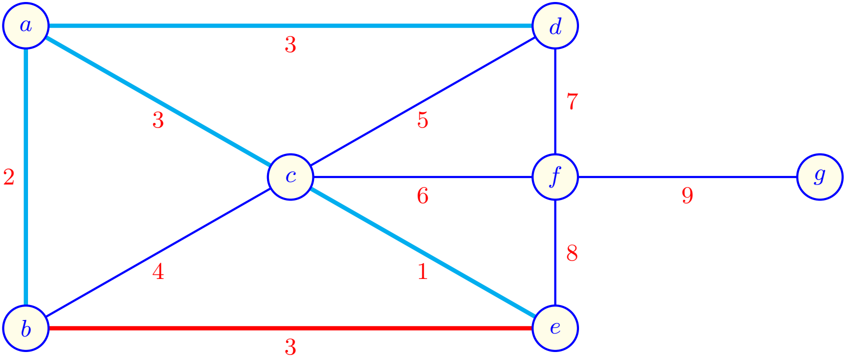


Step 4: $\{S_a = A.\texttt{set}(a)\} \neq \{S_d = A.\texttt{set}(d)\}$; **add** (a,d); $A.\texttt{union}(S_a, S_d)$

Disjoint-sets ADT $A = \big\{\{a,b,c,d,e\},\ \{f\},\ \{g\}\big\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c,e)	(a,b)	(a,c)	(a,d)	(b,e)	(b,c)	(c,d)	(c,f)	(d,f)	(e,f)	(f,g)

MST: Kruskal's Algorithm 5

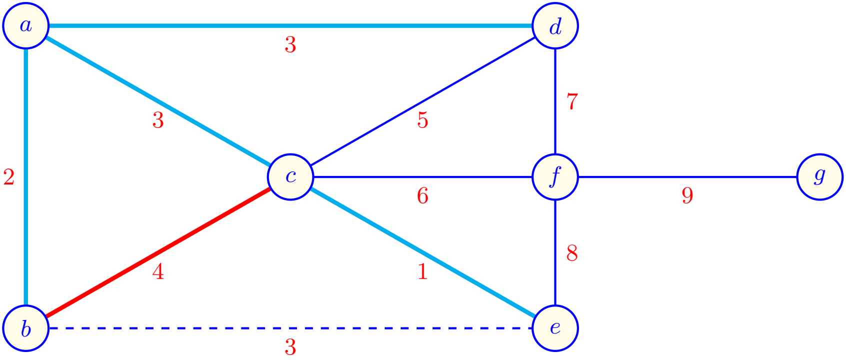


Step 5: $\{S_b = A.\mathtt{set}(b)\} = \{S_e = A.\mathtt{set}(e)\}$; **skip** (b, e)

Disjoint-sets ADT $A = \{\{a, b, c, d, e\}, \{f\}, \{g\}\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c, e)	(a, b)	(a, c)	(a, d)	(b, e)	(b, c)	(c, d)	(c, f)	(d, f)	(e, f)	(f, g)

MST: Kruskal's Algorithm 6

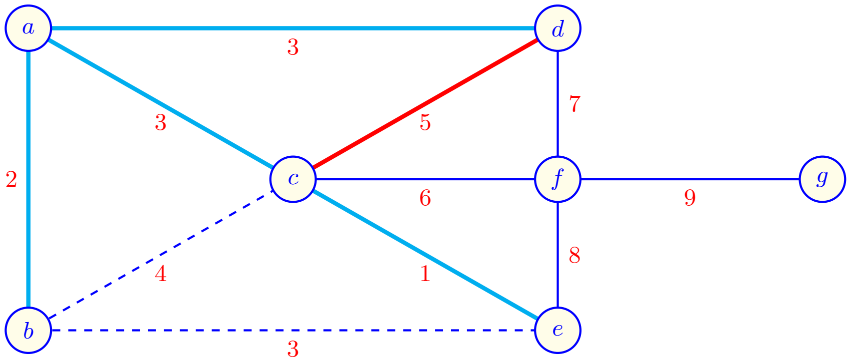


Step 6: $\{S_b = A.\mathtt{set}(b)\} = \{S_c = A.\mathtt{set}(c)\}$; **skip** (b, c)

Disjoint-sets ADT $A = \{\{a, b, c, d, e\}, \{f\}, \{g\}\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c, e)	(a, b)	(a, c)	(a, d)	(b, e)	(b, c)	(c, d)	(c, f)	(d, f)	(e, f)	(f, g)

MST: Kruskal's Algorithm 7

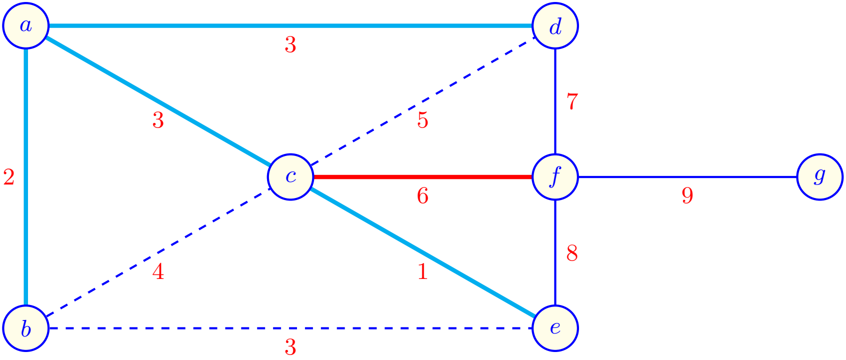


Step 7: $\{S_c = A.\mathtt{set}(c)\} = \{S_d = A.\mathtt{set}(d)\}$; **skip** (c, d)

Disjoint-sets ADT $A = \{\{a, b, c, d, e\}, \{f\}, \{g\}\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c, e)	(a, b)	(a, c)	(a, d)	(b, e)	(b, c)	(c, d)	(c, f)	(d, f)	(e, f)	(f, g)

MST: Kruskal's Algorithm 8

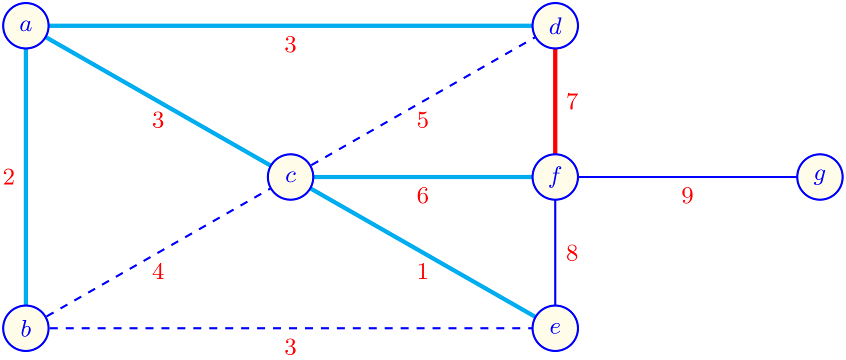


Step 8: $\{S_c = A.\texttt{set}(c)\} \neq \{S_f = A.\texttt{set}(f)\}$; **add** (c, f); $A.\texttt{union}(S_c, S_f)$

Disjoint-sets ADT $A = \{\{a, b, c, d, e, f\}, \{g\}\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c,e)	(a,b)	(a,c)	(a,d)	(b,e)	(b,c)	(c,d)	(c,f)	(d,f)	(e,f)	(f,g)

MST: Kruskal's Algorithm

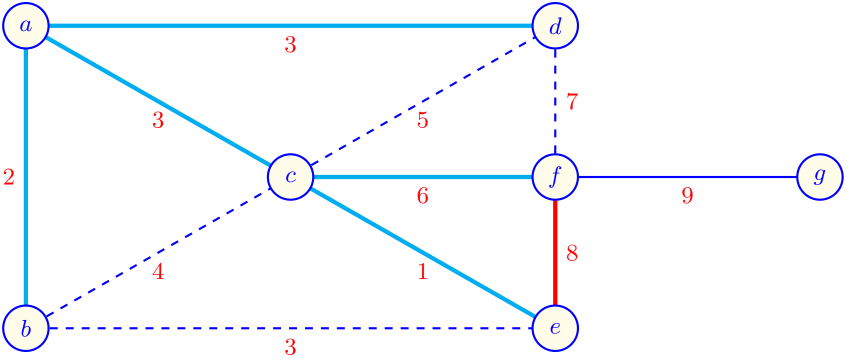


Step 9: $\{S_d = A.\mathtt{set}(d)\} = \{S_f = A.\mathtt{set}(f)\}$; **skip** (d, f)

Disjoint-sets ADT $A = \{\{a, b, c, d, e, f\}, \{g\}\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c, e)	(a, b)	(a, c)	(a, d)	(b, e)	(b, c)	(c, d)	(c, f)	(d, f)	(e, f)	(f, g)

MST: Kruskal's Algorithm 10

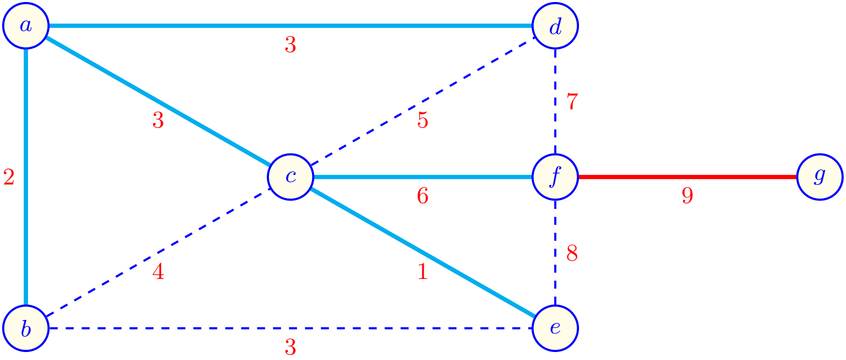


Step 10: $\{S_e = A.\mathtt{set}(e)\} = \{S_f = A.\mathtt{set}(f)\}$; **skip** (e, f)

Disjoint-sets ADT $A = \big\{\{a, b, c, d, e, f\}, \{g\}\big\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c,e)	(a,b)	(a,c)	(a,d)	(b,e)	(b,c)	(c,d)	(c,f)	(d,f)	(e,f)	(f,g)

MST: Kruskal's Algorithm 11

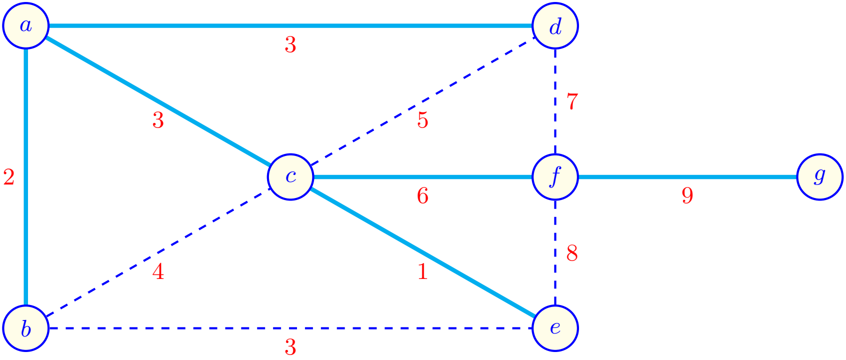


Step 11: $\{S_f = A.\texttt{set}(f)\} \neq \{S_g = A.\texttt{set}(g)\}$; **add** (f, g); $A.\texttt{union}(S_f, S_g)$

Disjoint-sets ADT $A = \Big\{\{a, b, c, d, e, f, g\}\Big\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c, e)	(a, b)	(a, c)	(a, d)	(b, e)	(b, c)	(c, d)	(c, f)	(d, f)	(e, f)	(f, g)

MST: Kruskal's Algorithm: Output



Step 11: $\{S_f = A.\texttt{set}(f)\} \neq \{S_g = A.\texttt{set}(g)\}$; add (f, g); $A.\texttt{union}(S_f, S_g)$

Disjoint-sets ADT $A = \{\{a, b, c, d, e, f, g\}\}$

cost	1	2	3	3	3	4	5	6	7	8	9
edge	(c,e)	(a,b)	(a,c)	(a,d)	(b,e)	(b,c)	(c,d)	(c,f)	(d,f)	(e,f)	(f,g)

Comparing the Prim's and Kruskal's Algorithms

Both algorithms choose and add at each step a min-weight edge from the remaining edges, subject to constraints.

Prim's MST algorithm:

- Start at a root vertex.
- Two rules for a new edge:
  - (a) No cycle in the subgraph built so far.
  - (b) The connected subgraph built so far.
- Terminate if no more edges to add can be found.

At each step: an acyclic connected subgraph being a tree.

Kruskal's MST algorithm:

- Start at a min-weight edge.
- One rule for a new edge:
  - (a) No cycle in a forest of trees built so far.
- Terminate if no more edges to add can be found.

At each step: a forest of trees merging as the algorithm progresses (can find a spanning forest for a disconnected graph).

Correctness of Prim's and Kruskal's Algorithms

Theorem 6.15: Prim's and Kruskal's algorithms are correct.

- A set of edges is promising if it can be extended to a MST.
- The theorem claims that both the algorithms
  1. choose at each step a promising set of edges and
  2. terminate with the MST as the set cannot be further extended.

Technical fact for proving these claims.

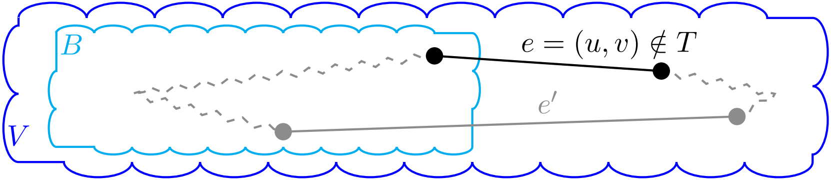
- Let $B \subset V(G)$; $|B| < n$, be a proper subset of the vertices.
- Let $T \subset E$ be a promising set of edges, such that no edge in T leaves B (i.e., If $(u, v) \in T$, then either both $u, v \in B$ or both $u, v \notin B$).
- If a minimum-weight edge e leaves B (one endpoint in B and one outside), then the set $T \bigcup \{e\}$ is also promising.

Correctness of Prim's and Kruskal's Algorithms

Proof of the technical fact that the set $T \bigcup \{e\}$ is promising.

- Since the set T is promising, it is in a some MST U.
- If $e \in U$, there is nothing to prove.
- Otherwise, adding e to U creates exactly one cycle.
  - This cycle contains at least one more edge, e', leaving B, as otherwise the cycle could not close.
- Removing the edge e' forms for the graph G a new spanning tree U'.
- Its total weight is no greater than the total weight of the MST U, and thus the tree U' is also an MST.
- Since the MST U' contains the set $T \bigcup \{e\}$ of edges, that set is promising.

Correctness of Prim's and Kruskal's Algorithms



Proof of Theorem 6.15:

- Suppose that the MST algorithm has maintained a promising set T of edges so far.
- Let an edge $e = \{u, v\}$ have been just chosen.
- Let B denote at each step
  - either the set of vertices in the tree (Prim)
  - or in the tree containing the vertex u (Kruskal).
- Then the above technical fact can be applied to conclude that $T \bigcup \{e\}$ is promising and the algorithm is correct.

Minimum Spanning Trees (MST): Some Properties

Can you prove these two facts?

1. The maximum-cost edge, if unique, of a cycle in an edge-weighted graph G is not in any MST.

   Otherwise, at least one of those equally expensive edges of the cycle must not be in each MST.

2. The minimum-cost edge, if unique, between any non-empty strict subset S of $V(G)$ and the $V(G) \setminus S$ is in any MST.

   Otherwise, at least one of these minimum-cost edges must be in each MST.

Hint: Look whether a total weight of an MST with such a maximum-cost edge or without such a minimum-cost edge can be further decreased.

Other (Di)graph Optimisation/Decision Problems

There are many more graph and network computational and optimisation problems.

Many of them do not have easy or polynomial-time solutions.

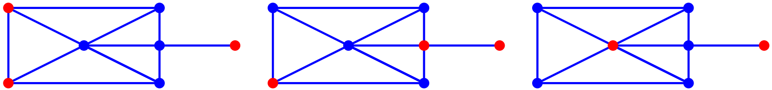
However, a few of them are in a special class in that their solutions can be verified in polynomial time.

- This class of computational problems is called the NP (nondeterministic polynomial) class.
- In addition, many of these are proven to be harder than anything else in the NP class.
- The latter NP problems are called NP-complete ones.

Other algorithm design techniques like backtracking, branch-and-bound or approximation algorithms (studied in COMPSCI 320) are needed.

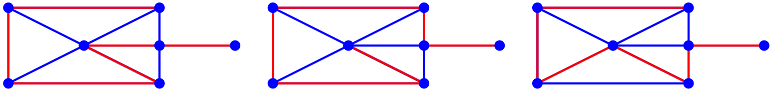
Examples of NP-complete Graph Problems

Vertex Cover, or dominating set: Finding a subset of k; $k \leq |V(G)|$, vertices such that every vertex of the graph is adjacent to one in that subset.



- Finding the smallest vertex cover in the graph is NP-complete.
- However, it is polynomial-time solvable for bipartite graphs.

Hamiltonian path: Finding a path through all the vertices of a graph.

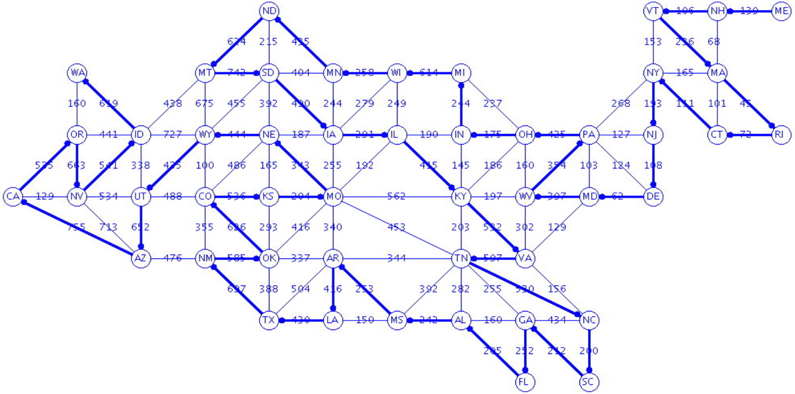


Hamiltonian cycle: Finding a cycle through all the vertices of a graph (graphs containing such a cycle are called *Hamiltonian graphs*).

Hamiltonian Paths - Examples

http://www.cs.utsa.edu/~wagner/CS3343/graphapp2/hampaths.html

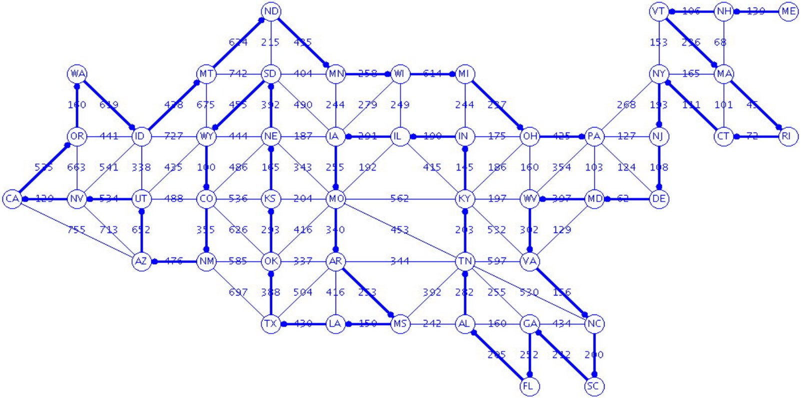
The longest (18,040 miles) Hamiltonian path from Maine (ME) between capitals of all 48 mainland US states out of the $68,656,026$ possible Hamiltonian paths:



Hamiltonian Paths - Examples

http://www.cs.utsa.edu/~wagner/CS3343/graphapp2/hampaths.html

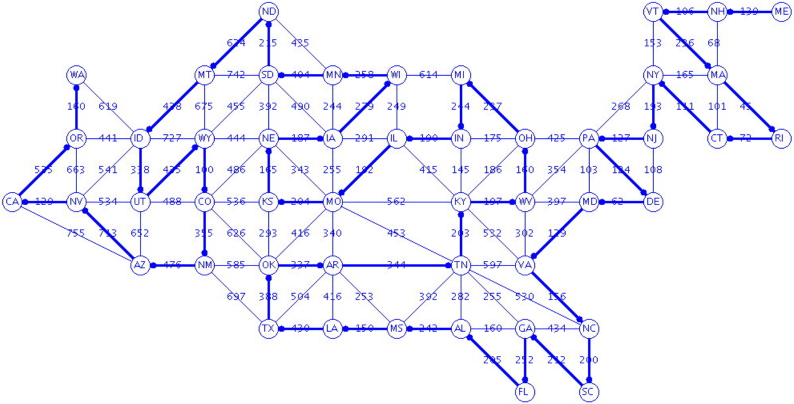
The random (13,619miles) Hamiltonian path from Maine (ME) between capitals of all 48 mainland US states out of the $68,656,026$ possible Hamiltonian paths:



Hamiltonian Paths - Examples

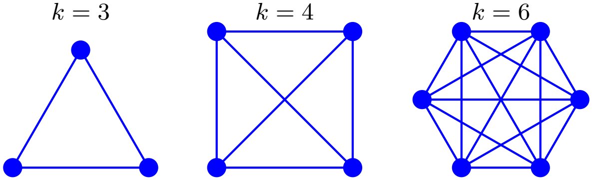
http://www.cs.utsa.edu/~wagner/CS3343/graphapp2/hampaths.html

The shortest (11,698 miles) Hamiltonian path from Maine (ME) between capitals of all 48 mainland US states out of the $68,656,026$ possible Hamiltonian paths:



Examples of NP-complete Graph Problems

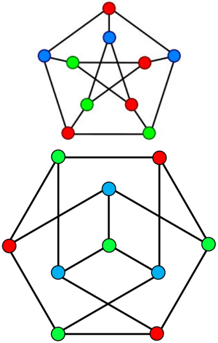
- **Longest path**: Finding the longest path between two nodes of a digraph.
- **k-colouring**: Finding a k-colouring of a graph, for fixed $k \geq 3$.
- **Feedback arc set**: Finding a subset F of k; $k \leq |V(G)|$, nodes such that $G \setminus F$ is a DAG.
- **Maximum clique**: Finding a complete subgraph of the maximum order k in a given graph $G = (V, E)$.
  - In a complete subgraph, $G' = (V', E') \subset G$, all the nodes $u, v \in V' \subseteq V$ are adjacent, i.e., $(u', v') \in E' \subseteq E$.

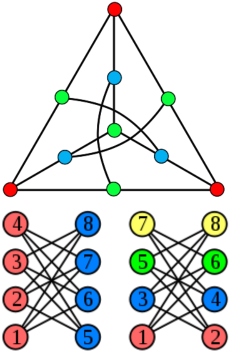


NP-complete Graph Colouring: Examples

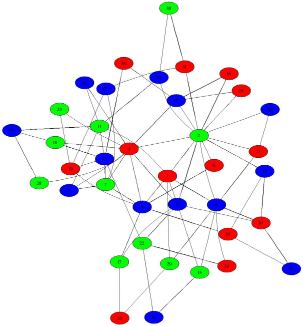
https://en.wikipedia.org/wiki/Graph_coloring
http://iasbs.ac.ir/seminar/math/combinatorics/
https://heuristicswiki.wikispaces.com/Graph+coloring

Optimisation: Colouring a general graph with the minimum number of colours.



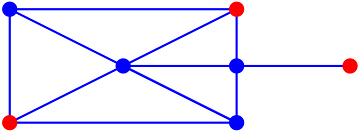


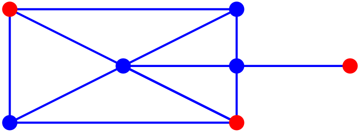
Greedy colouring



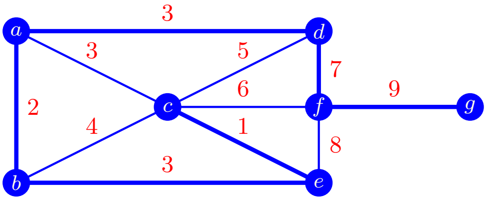
Examples of NP-complete Graph Problems

Independent set: Finding the largest subset of vertices, no two of which are connected by an edge.





Travelling salesman problem (TSP): Finding a minimum weight path through all the vertices of a weighted digraph (G, c).



Total weight of the path c, e, b, a, d, f, g: 25

TSP – NP-Hard, but not NP-Complete Problem

Blog by Jean Francois Paget: https://www.ibm.com/developerworks/community/blogs/jfp/entry/no_the_tsp_isn_t_np_complete?lang=en

- **NP problem** – its solution can be verified in polynomial time.
- **NP-hard problem** – it is as difficult as any NP problem.
- **NP-complete problem** – it is both NP and NP-hard.

For a given TSP solution:

1. Each city is visited once (easy verified in polynomial time).
2. Total travel length is minimal (no known polynomial-time check).

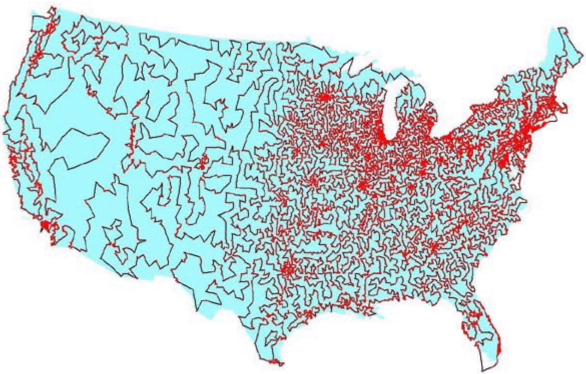
$N_n = (n-1)!$ of paths through n vertices, starting from an arbitrary vertex:

n	10	20	100	1000	10000
N_n	$3.63 \cdot 10^5$	$1.22 \cdot 10^{17}$	$9.33 \cdot 155$	$4.02 \cdot 10^{2564}$	$2.85 \cdot 10^{35655}$

TSP – NP-Hard, but not NP-Complete Problem

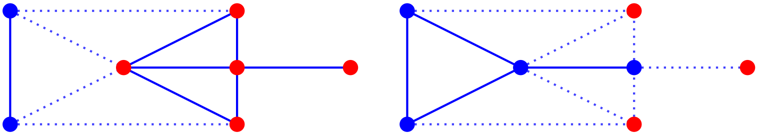
Blog by Jean Francois Paget: https://www.ibm.com/developerworks/community/blogs/jfp/entry/no_the_tsp_isn_t_np_complete?lang=en

Effective algorithms for solving the TSPs with large n exist. See, e.g., the optimal TSP solution by D. Applegate, R. Bixby, V. Chvatal, and W. Cook for $n = 13,509$ cities and towns with more than 500 residents in the USA:



Examples of NP-complete Graph Problems

Maximum Cut: Determining whether vertices of $G = (V, E)$ can be separated into two non-empty subsets V_1 and V_2; $V_1 \bigcup V_2 = V$; $V_1 \bigcap V_2 = \emptyset$, with at most k edges between V_1 and V_2.



$k = 4$ $k = 7$

- **Max / min-cut optimisation**: Maximising / minimising the number k of edges between the separated subsets.
- **Weighted max / min-cut**: Maximising / minimising the total weight of edges between the separated subsets.

Examples of NP-complete Graph Problems

- **Induced path**: Determining whether there is an induced subgraph of order k being a simple path.
- **Bandwidth**: Determining whether there is a linear ordering of V with bandwidth k or less.
  - Bandwidth k – each edge spans at most k vertices.
- **Subgraph Isomorphism**: Determining whether H is a sub(di)graph of G.
- **Minimum broadcast time**: Determining for a given source node of a digraph G, whether (point-to-point) broadcast to all other nodes can take at most k time steps.
- **Disjoint connecting paths**: Determining for given k pairs of source and sink vertices of a graph G, whether there are k vertex-disjoint paths connecting each pair.