Computer Science 331

Computation of Minimum-Cost Spanning Trees — Prim's Algorithm

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #35

Outline

Computation of Min-Cost Spanning Trees

Motivation: Given a set of sites (represented by vertices of a graph), connect these all as cheaply as possible (using connections represented by the edges of a weighted graph).

Goals for Today:

- presentation of the definitions needed to formally define a problem motivated by the above
- presentation of an algorithm (Prim's Algorithm) for solving the problem

Costs of Spanning Trees in Weighted Graphs

Recall that if $G = (V, E)$ is a connected, undirected graph, then a *spanning tree* of G is a subgraph $\widehat{G} = (\widehat{V}, \widehat{E})$ such that

- $\widehat{V} = V$ (so $\widehat{G}$ includes all the vertices in G)
- $\widehat{G}$ is a tree

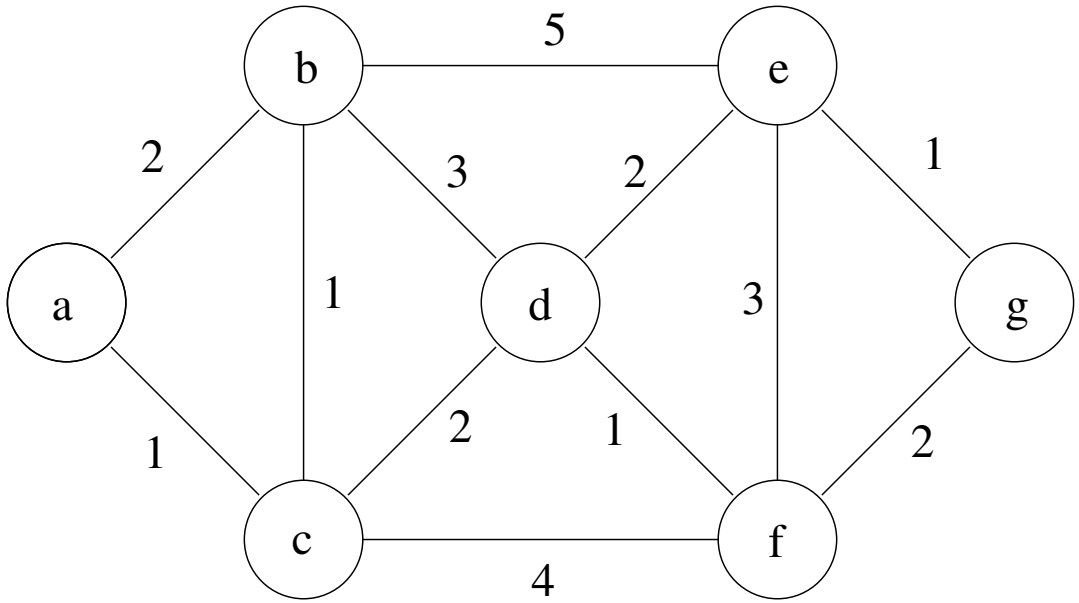
Suppose now that $G = (V, E)$ is a connected *weighted* graph with weight function $w : E \mapsto \mathbb{N}$, and that $G_1 = (V_1, E_1)$ is a spanning tree of G

The *cost* of G_1, $w(G_1)$, is the sum of the weights of the edges in G_1, that is,

$$w(G_1) = \sum_{e \in E_1} w(e).$$

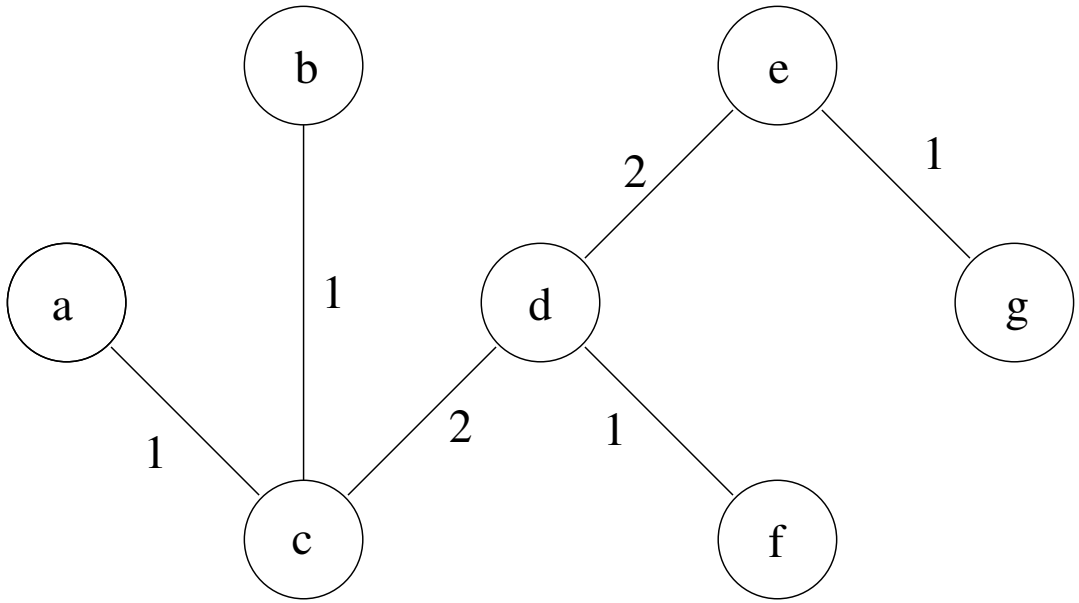
Example

Suppose G is a weighted graph with weights as shown below.



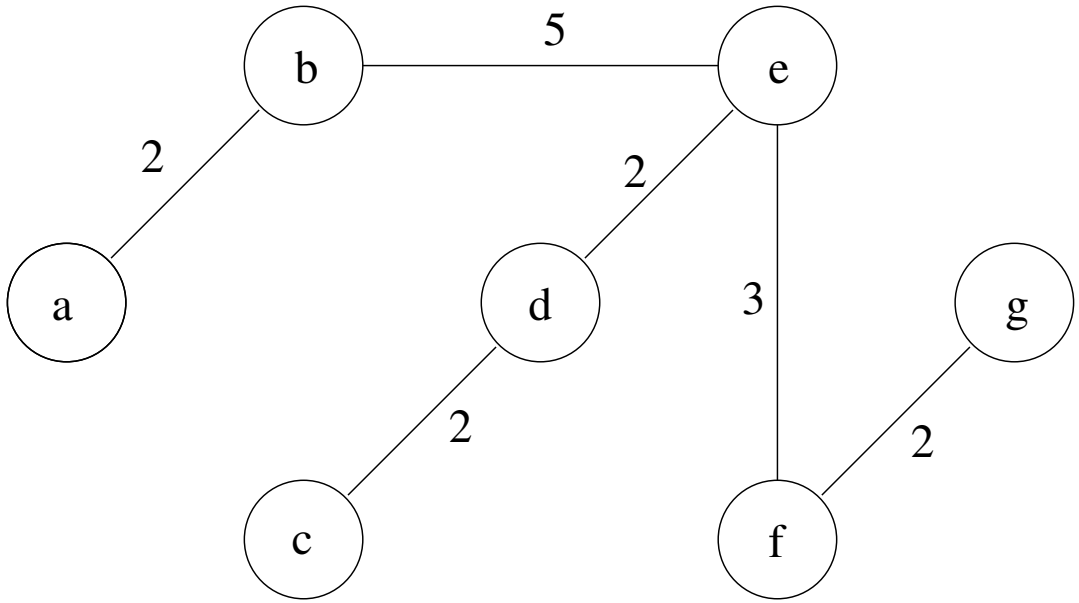
Example

The cost of the following spanning tree, $G_1 = (V_1, E_1)$, is 8.



Example

The cost of the following spanning tree, $G_2 = (V_2, E_2)$, is 16.



Minimum-Cost Spanning Trees

Suppose (G, w) is a weighted graph.

A subgraph G_1 of G is a *minimum-cost spanning tree* of (G, w) if the following properties are satisfied.

1. G_1 is a spanning tree of G.
2. $w(G_1) \leq w(G_2)$ for every spanning tree G_2 of G.

Example: In the previous example, G_2 is clearly *not* a minimum-cost spanning tree, because G_1 is a spanning tree of G such that $w(G_2) > w(G_1)$.

- It can be shown that G_1 *is* a minimum-cost spanning tree of (G, w).

Building a Minimum-Cost Spanning Tree

To construct a minimum-cost spanning tree of $G = (V, E)$:

1. Start with $\widehat{G} = (\widehat{V}, \widehat{E})$, where $\widehat{V} \subseteq V$ and $\widehat{E} = \emptyset$.

   Note: $\widehat{G}$ is a subgraph of some minimum-cost spanning tree of (G, w).

2. Repeatedly add vertices (if necessary) and edges — ensuring that $\widehat{G}$ is still a subgraph of a minimum-cost spanning tree as you do so.

   Continue doing this until $\widehat{V} = V$ and $|\widehat{E}| = |V| - 1$ (so that $\widehat{G}$ is a spanning tree of $\widehat{G}$).

Building a Minimum-Cost Spanning Tree

Additional Notes:

- This can be done in several different ways, and there are at least two different algorithms that use this approach to solve this problem.

  The algorithm to be presented here begins with $\widehat{V} = \{s\}$ for some vertex $s \in V$, and makes sure that $\widehat{G}$ is always a *tree*.
- As a result, this algorithm is structurally very similar to *Dijkstra's Algorithm* to compute minimum-cost paths (which we have already discussed in class).

Specification of Requirements

Pre-Condition

- $G = (V, E)$ is a **connected** graph with weight function w

Post-Condition:

- π is a function $\pi : V \to V \cup \{\text{NIL}\}$
- If
$$\widehat{E} = \{(\pi(v), v) \mid v \in V \text{ and } \pi(v) \neq \text{NIL}\}$$
then $(V, \widehat{E})$ is a minimum-cost spanning tree for G
- The graph $G = (V, E)$ and its weight function have not been changed

Data Structures

The algorithm (to be presented next) will use a **priority queue** to store information about weights of edges that are being considered for inclusion

- The priority queue will be a *MinHeap*: the entry with the *smallest* priority will be at the top of the heap
- Each node in the priority queue will store a *vertex* in G and the *weight* of an edge incident to this vertex
- The *weight* will be used as the vertex's priority
- An array-based representation of the priority queue will be used

A second array will be used to locate each entry of the priority queue for a given node in constant time

Note: The data structures will, therefore, look very much like the data structures used by Dijkstra's algorithm.

Pseudocode

MST-Prim(G, w, s)

for $v \in V$ **do**
  $colour[v] = \text{white}$
  $d[v] = +\infty$
  $\pi[v] = \text{NIL}$
end for
Initialize an empty priority queue Q
$colour[s] = \text{grey}$
$d[s] = 0$
add s with priority 0 to Q

Pseudocode, Continued

while (Q is not empty) **do**
  $(u, c) =$ extract-min(Q) {Note: $c = d[u]$}
  for each $v \in Adj[u]$ **do**
    if ($colour[v] ==$ white) **then**
      $d[v] = w((u, v))$
      $colour[v] =$ grey; $\pi[v] = u$
      add v with priority $d[v]$ to Q
    else if ($colour[v] ==$ grey) **then**
      Update information about v (Shown on next slide)
    end if
  end for
  $colour[u] =$ black
end while
return π

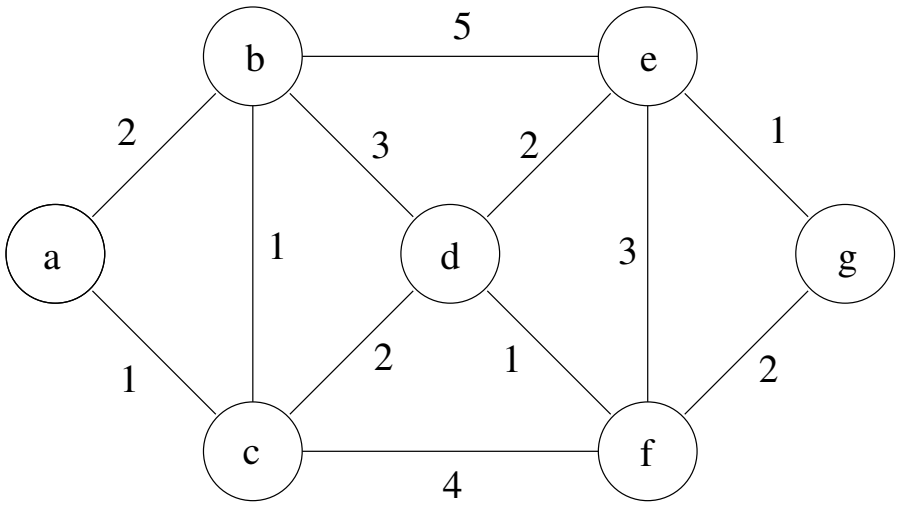
Pseudocode, Concluded

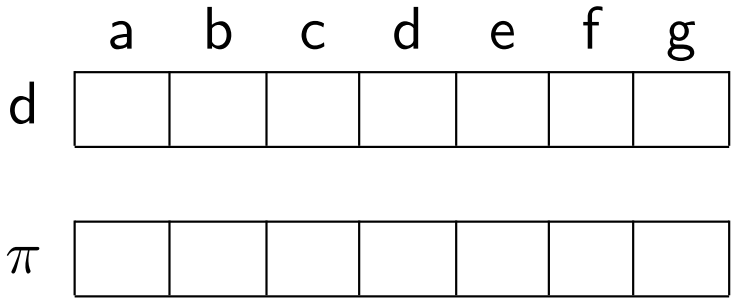
Updating Information About v

if $(w((u,v)) < d[v])$ **then**
  $old = d[v]$
  $d[v] = w((u,v))$
  $\pi[v] = u$
  Use `Decrease-Priority` to replace (v, old)
  in Q with $(v, d[v])$
end if

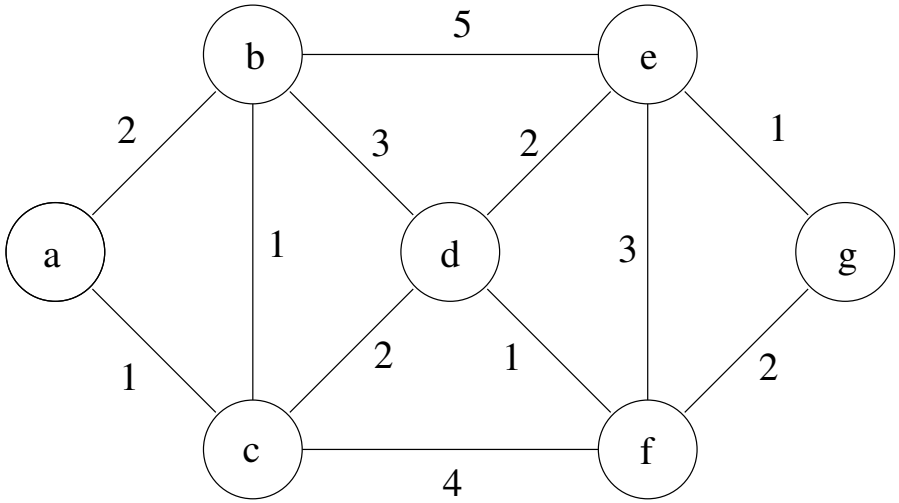
Example

Consider the execution of MST-Prim(G):





Example



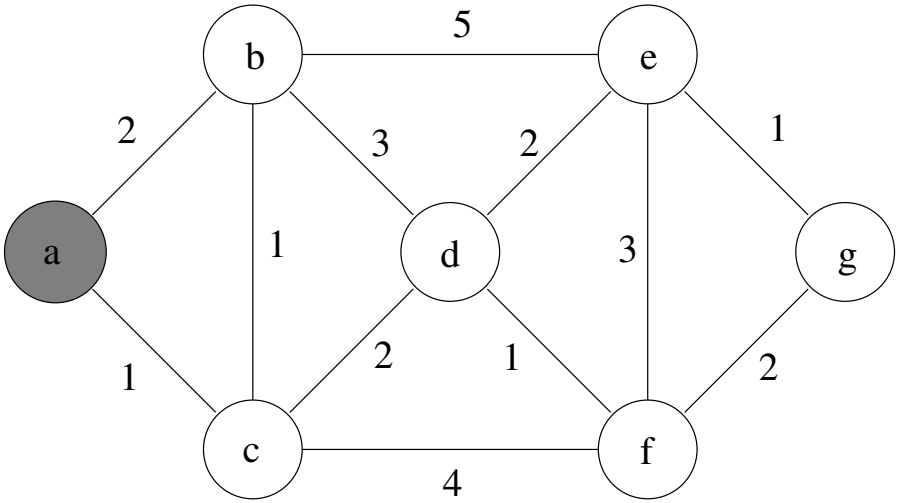
	a	b	c	d	e	f	g
d	-	-	-	-	-	-	-
π	-	-	-	-	-	-	-

Q: (empty)

Step 0:

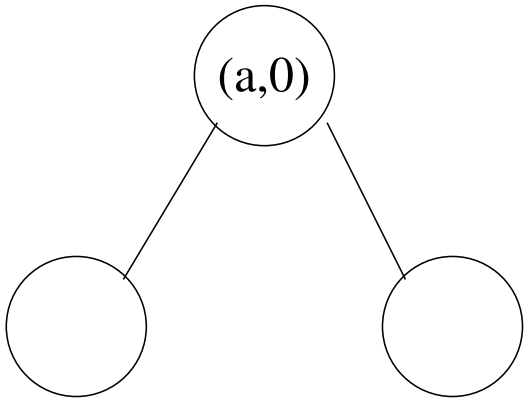
- initialization

Example



	a	b	c	d	e	f	g
d	0	-	-	-	-	-	-
π	-	-	-	-	-	-	-

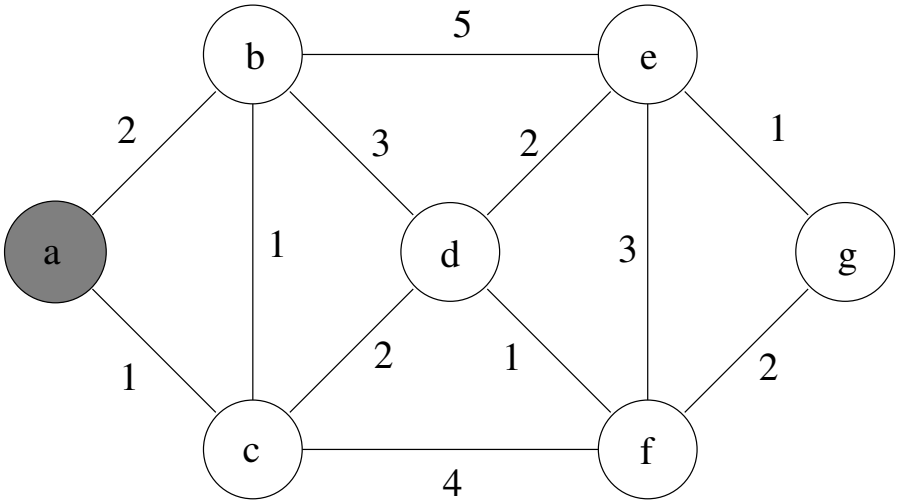
Q:



Step 0:

- initialization
- enqueue$(a, 0)$

Example



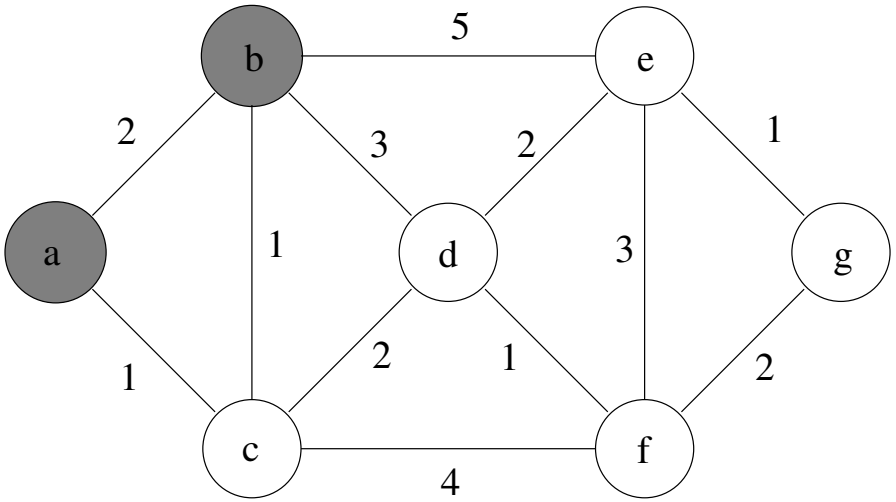
	a	b	c	d	e	f	g
d	0	-	-	-	-	-	-
π	-	-	-	-	-	-	-

Q: (empty)

Step 1:

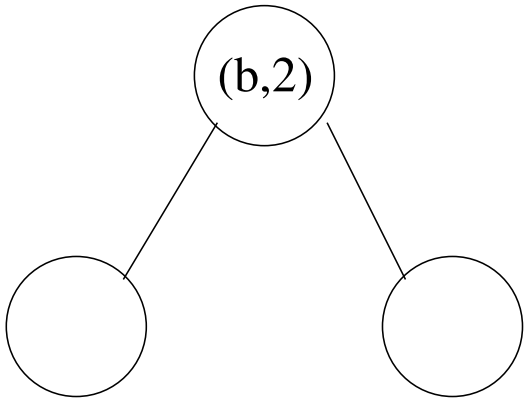
- Extract-Min (returns $(a, 0)$)

Example



	a	b	c	d	e	f	g
d	0	2	-	-	-	-	-
π	-	*a*	-	-	-	-	-

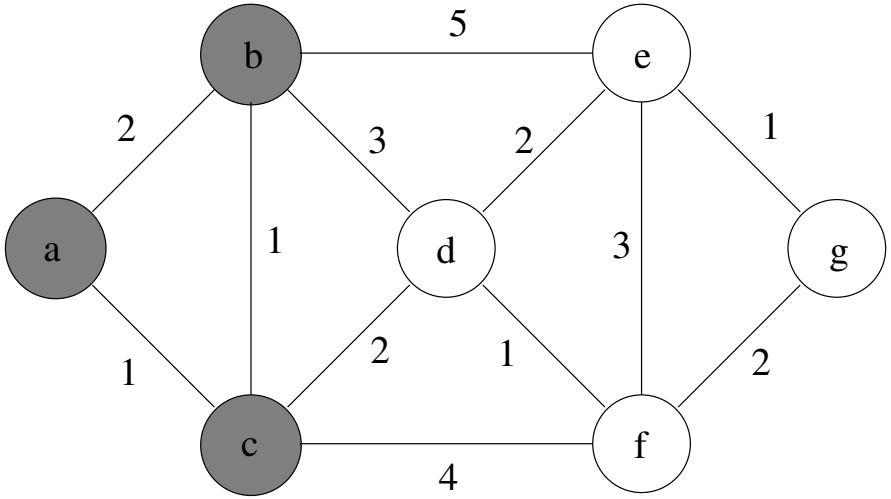
Q:



Step 1:

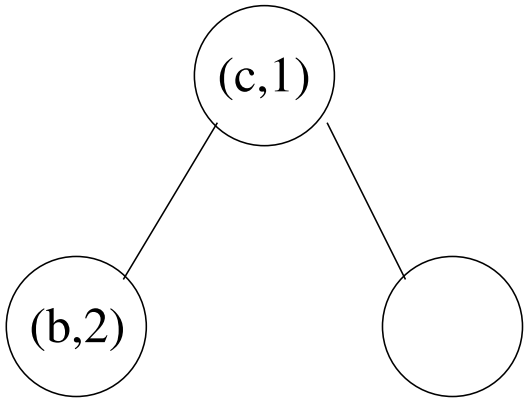
- Extract-Min (returns $(a, 0)$)
- add *b*

Example



	a	b	c	d	e	f	g
d	0	2	1	-	-	-	-
π	-	*a*	*a*	-	-	-	-

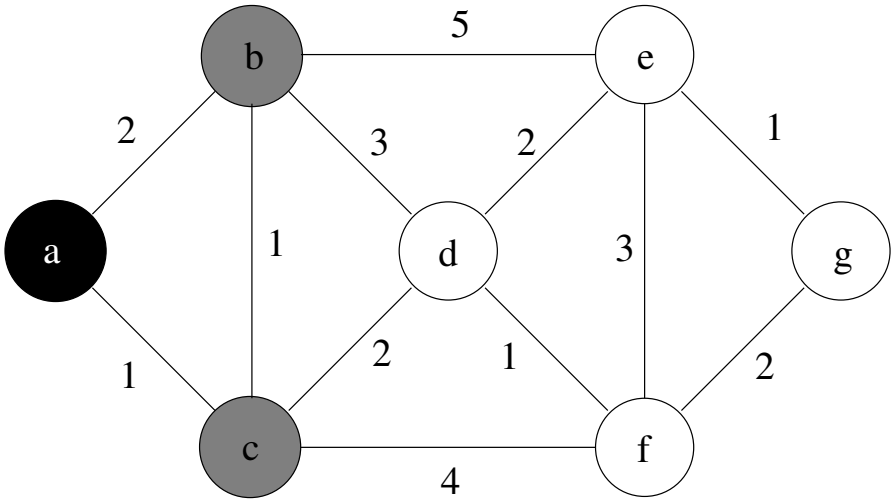
Q:



Step 1:

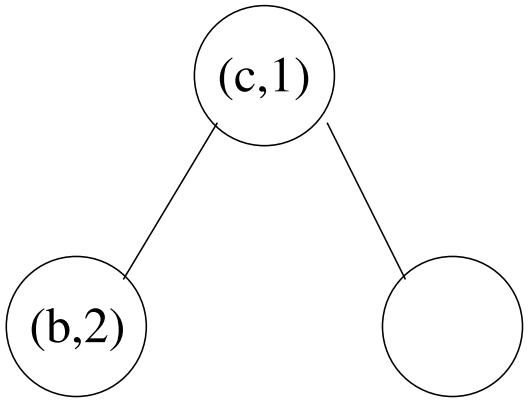
- Extract-Min (returns $(a, 0)$)
- add b
- add c

Example



	a	b	c	d	e	f	g
d	0	2	1	-	-	-	-
π	-	*a*	*a*	-	-	-	-

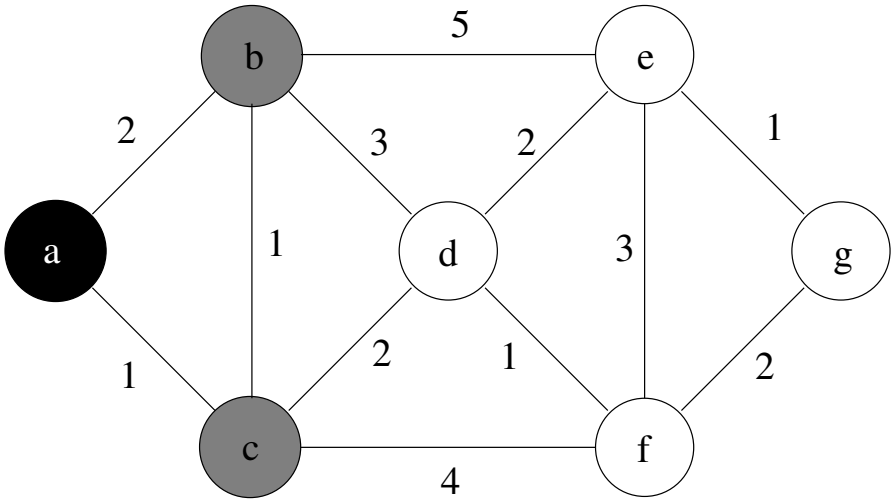
Q:



Step 1:

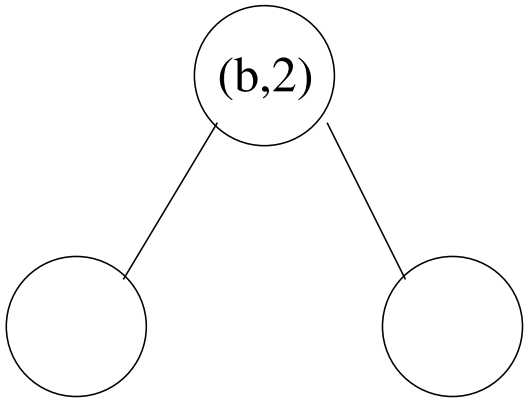
- Extract-Min (returns $(a, 0)$)
- add *b*
- add *c*
- color *a* black

Example



	a	b	c	d	e	f	g
d	0	2	1	-	-	-	-
π	-	*a*	*a*	-	-	-	-

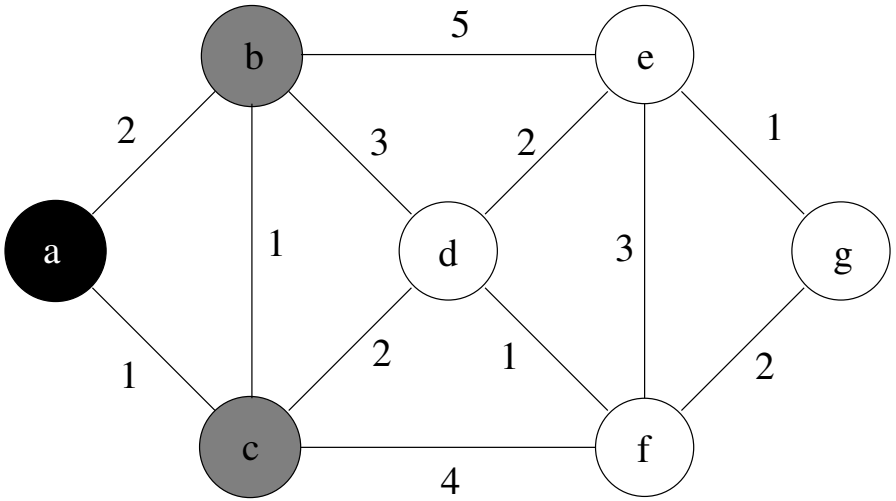
Q:



Step 2:

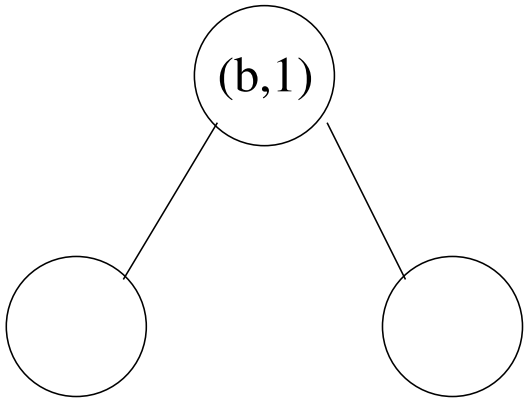
- Extract-Min (returns $(c, 1)$)

Example



	a	b	c	d	e	f	g
d	0	1	1	-	-	-	-
π	-	c	a	-	-	-	-

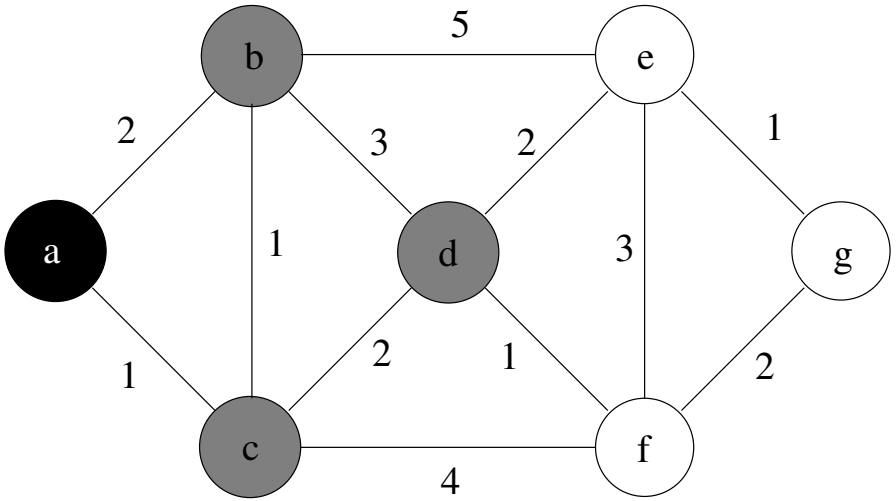
Q:



Step 2:

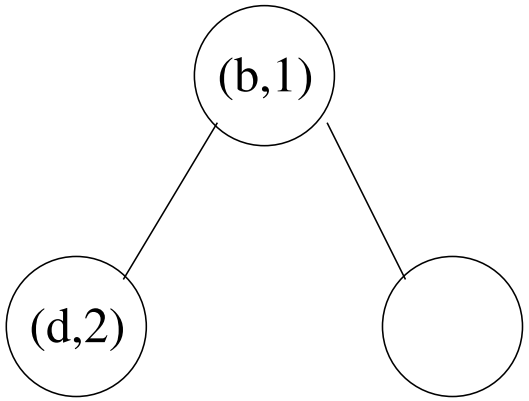
- Extract-Min (returns $(c, 1)$)
- update b

Example



	a	b	c	d	e	f	g
d	0	1	1	2	-	-	-
π	-	c	a	c	-	-	-

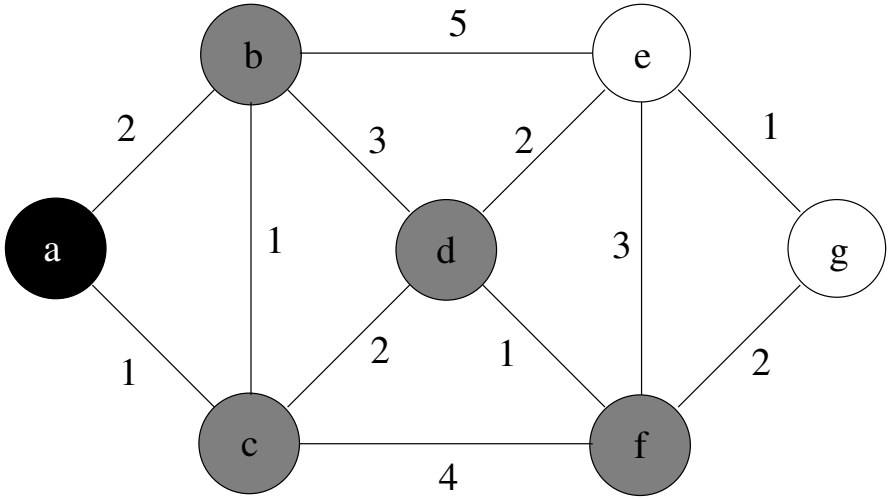
Q:



Step 2:

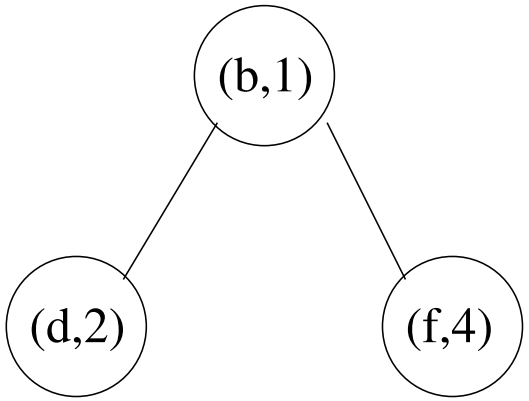
- Extract-Min (returns $(c, 1)$)
- update b
- add d

Example



	a	b	c	d	e	f	g
d	0	1	1	2	-	4	-
π	-	c	a	c	-	c	-

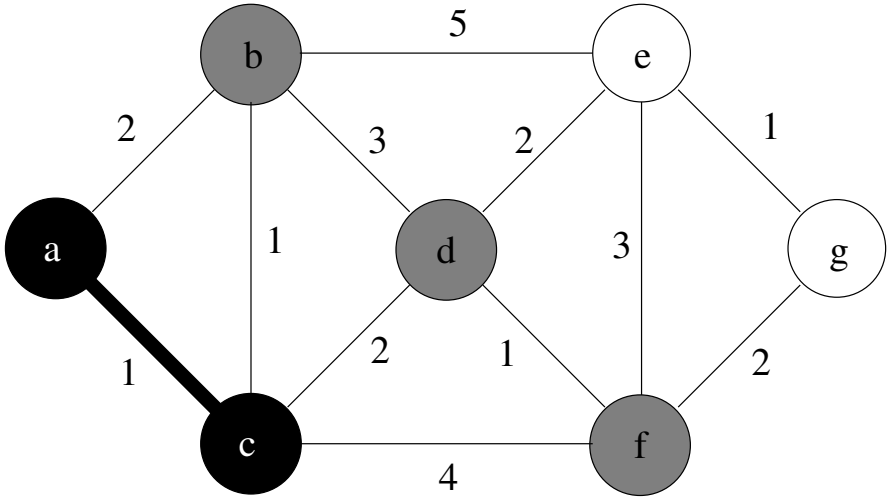
Q:



Step 2:

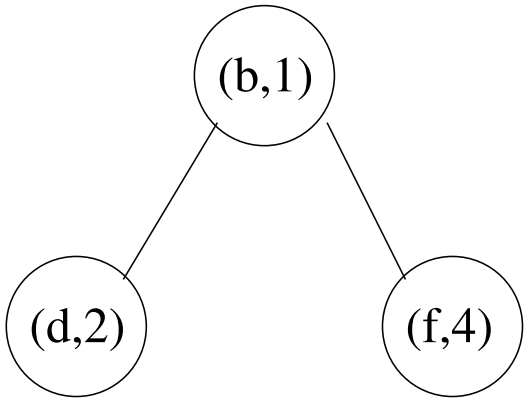
- Extract-Min (returns $(c, 1)$)
- update b
- add d
- add f

Example



	a	b	c	d	e	f	g
d	0	1	1	2	-	4	-
π	-	c	a	c	-	c	-

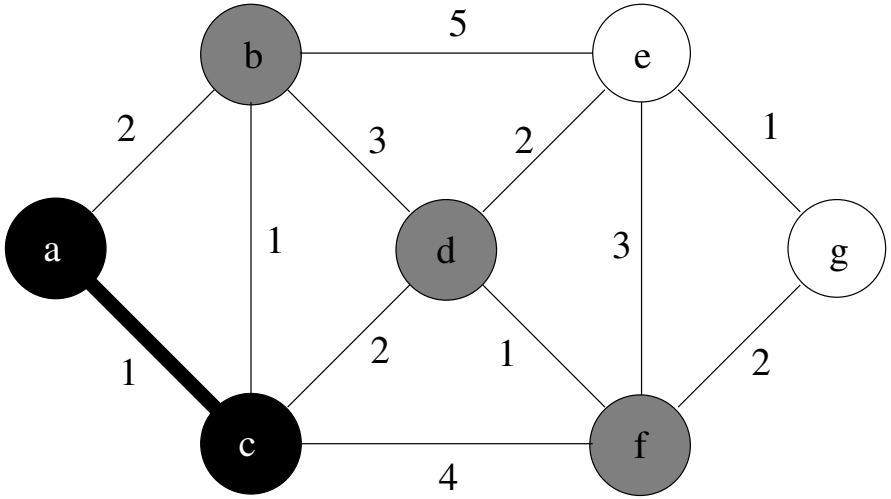
Q:



Step 2:

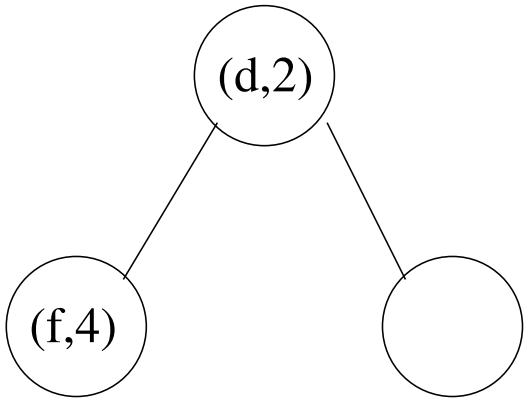
- Extract-Min (returns $(c, 1)$)
- update b
- add d
- add f
- color c black

Example



	a	b	c	d	e	f	g
d	0	1	1	2	-	4	-
π	-	c	a	c	-	c	-

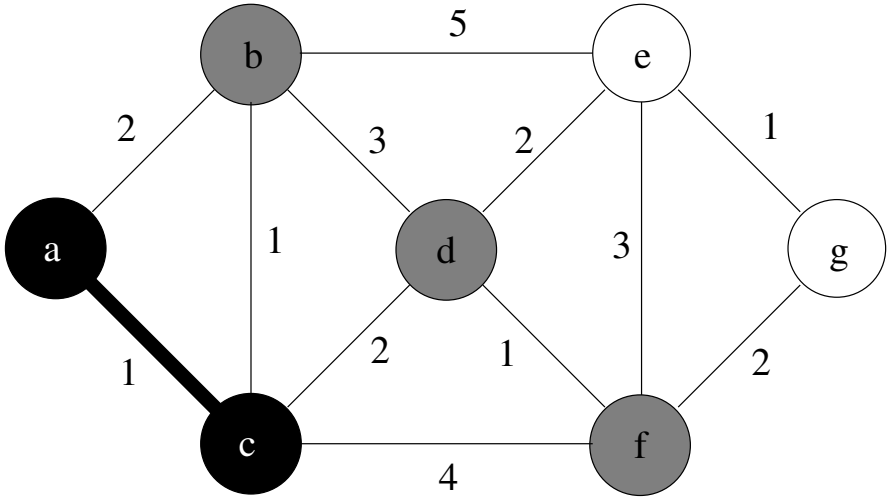
Q:



Step 3:

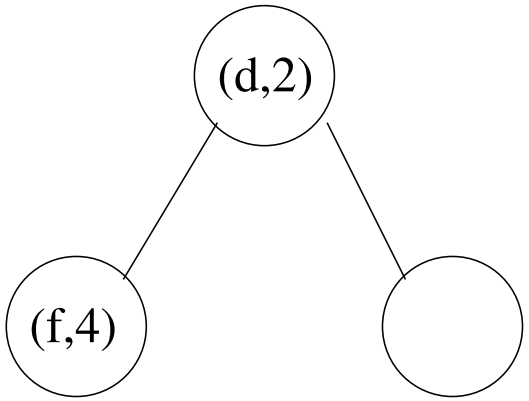
- Extract-Min (returns $(b, 1)$)

Example



	a	b	c	d	e	f	g
d	0	1	1	2	-	4	-
π	-	*c*	*a*	*c*	-	*c*	-

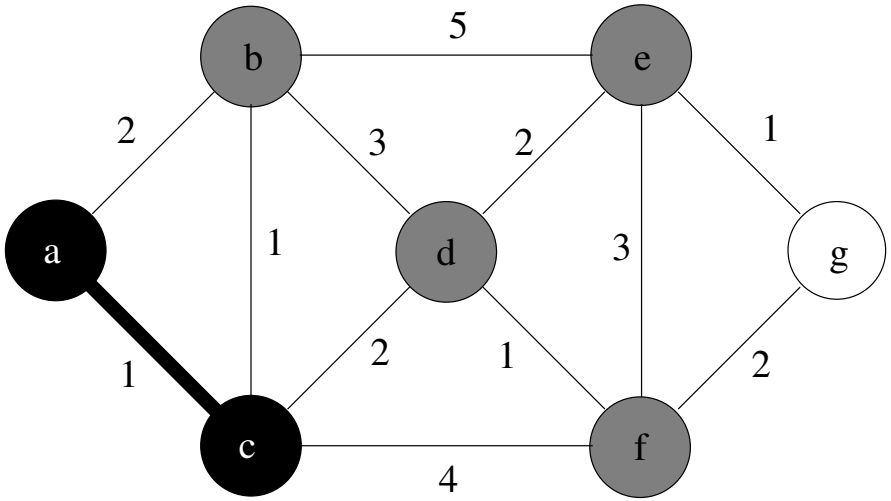
Q:



Step 3:

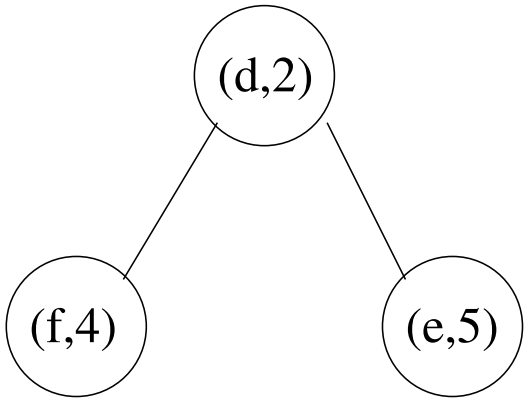
- Extract-Min (returns $(b, 1)$)
- update d (no change)

Example



	a	b	c	d	e	f	g
d	0	1	1	2	5	4	-
π	-	c	a	c	b	c	-

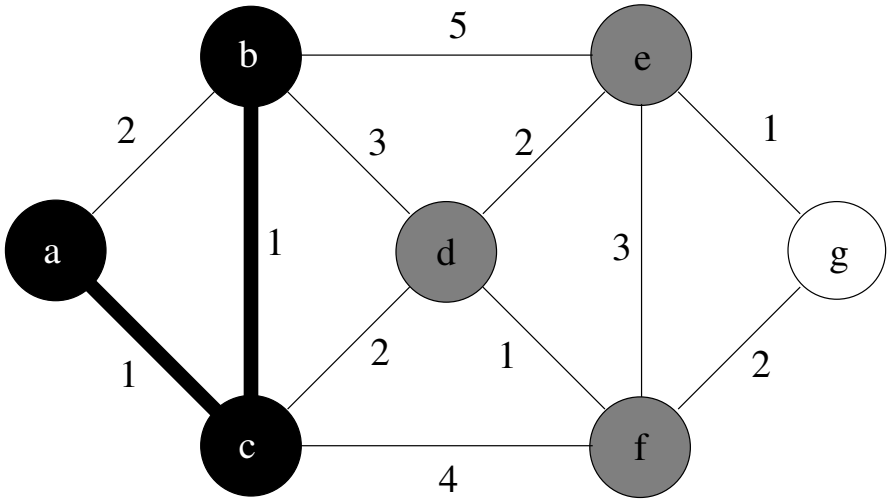
Q:



Step 3:

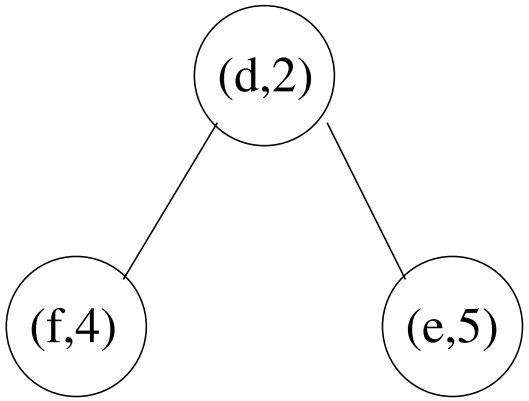
- Extract-Min (returns $(b, 1)$)
- update d (no change)
- add e

Example



	a	b	c	d	e	f	g
d	0	1	1	2	5	4	-
π	-	*c*	*a*	*c*	*b*	*c*	-

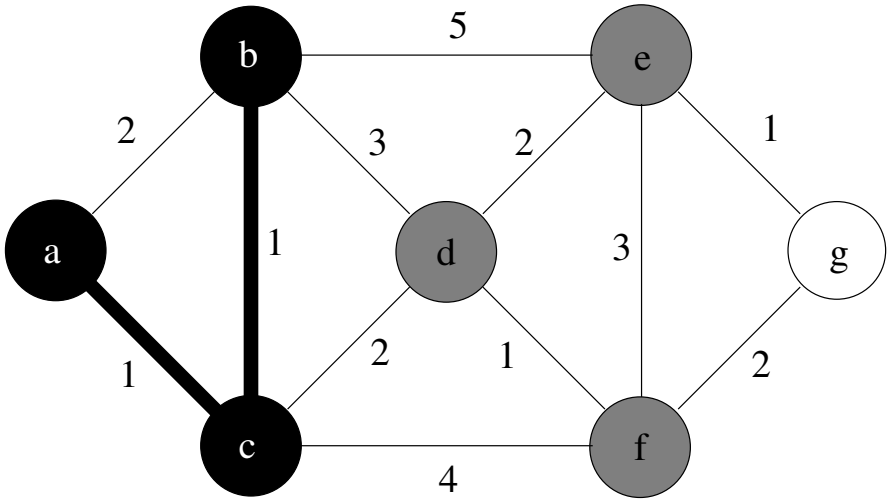
Q:



Step 3:

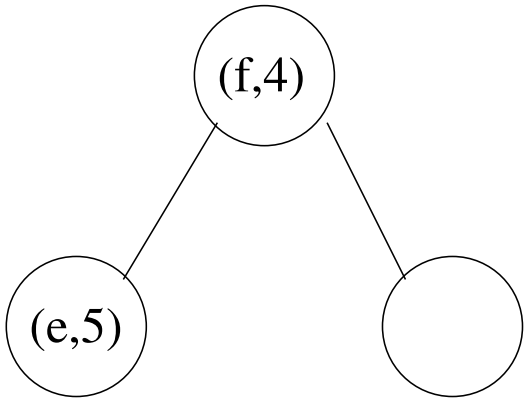
- Extract-Min (returns $(b, 1)$)
- update d (no change)
- add e
- color b black

Example



	a	b	c	d	e	f	g
d	0	1	1	2	5	4	-
π	-	c	a	c	b	c	-

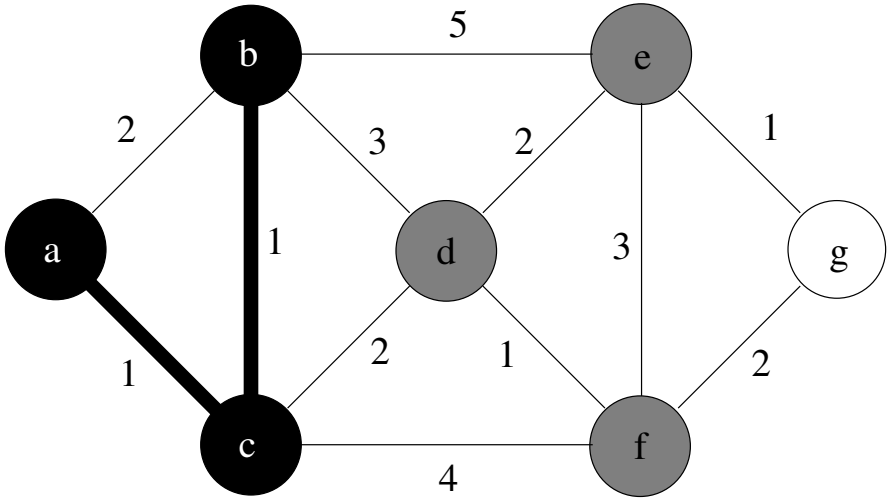
Q:



Step 4:

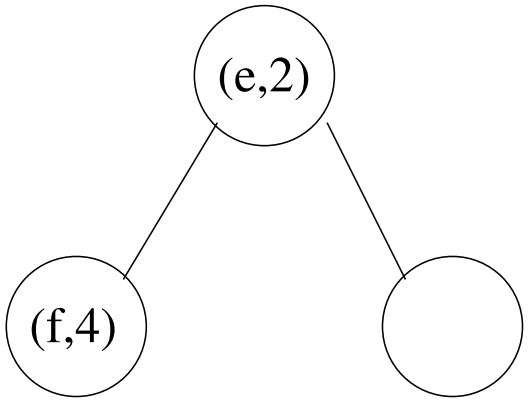
- Extract-Min (returns $(d, 2)$)

Example



	a	b	c	d	e	f	g
d	0	1	1	2	2	4	-
π	-	*c*	*a*	*c*	*d*	*c*	-

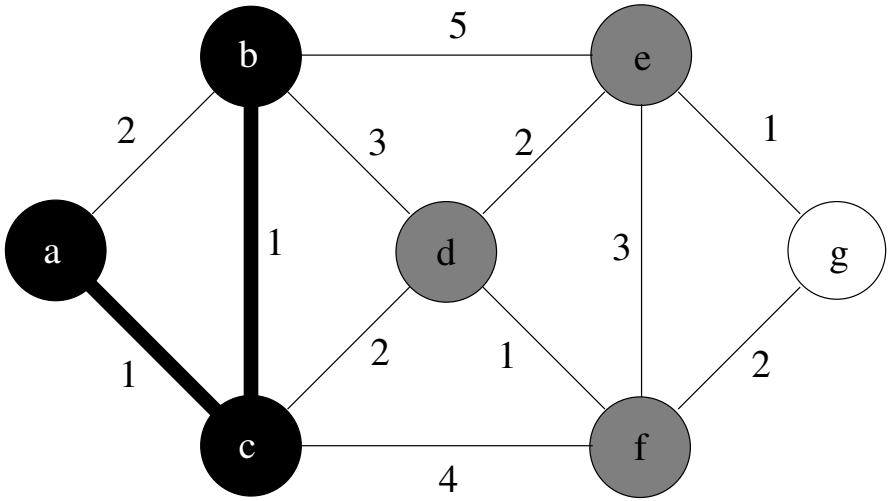
Q:



Step 4:

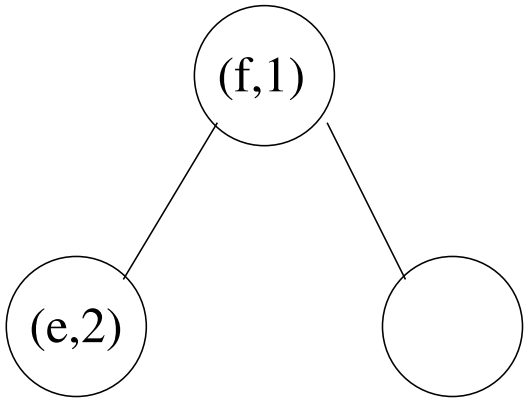
- Extract-Min (returns $(d, 2)$)
- update e

Example



	a	b	c	d	e	f	g
d	0	1	1	2	2	1	-
π	-	*c*	*a*	*c*	*d*	*d*	-

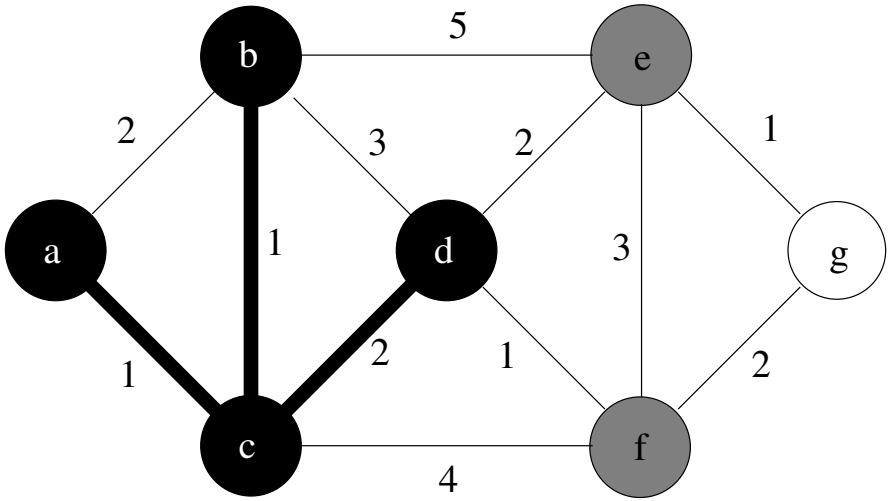
Q:



Step 4:

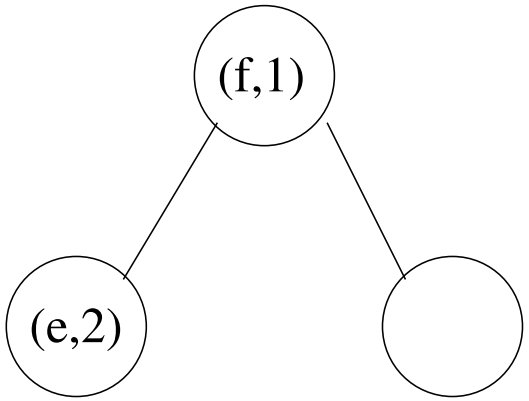
- Extract-Min (returns $(d, 2)$)
- update e
- update f

Example



	a	b	c	d	e	f	g
d	0	1	1	2	2	1	-
π	-	*c*	*a*	*c*	*d*	*d*	-

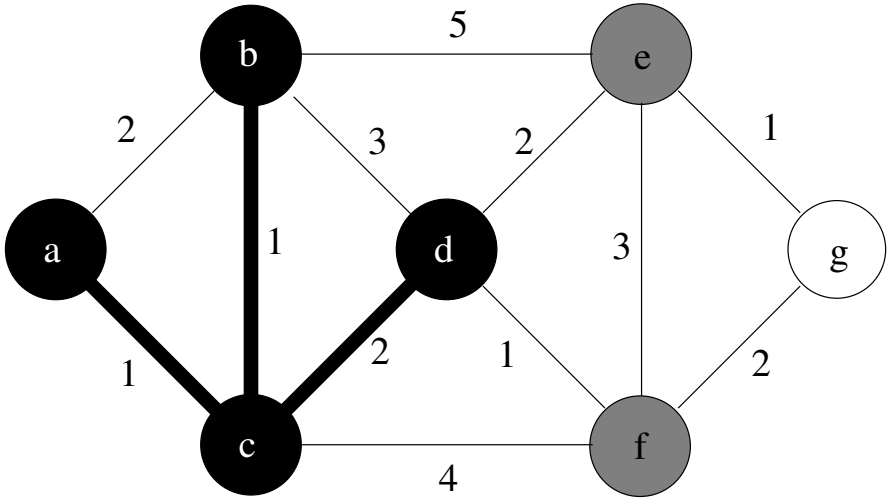
Q:



Step 4:

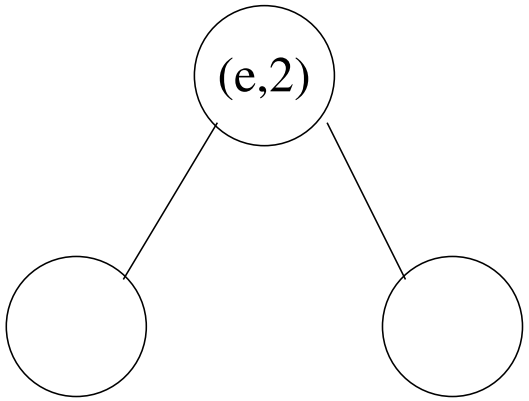
- Extract-Min (returns $(d, 2)$)
- update e
- update f
- color d black

Example



	a	b	c	d	e	f	g
d	0	1	1	2	2	1	-
π	-	c	a	c	d	d	-

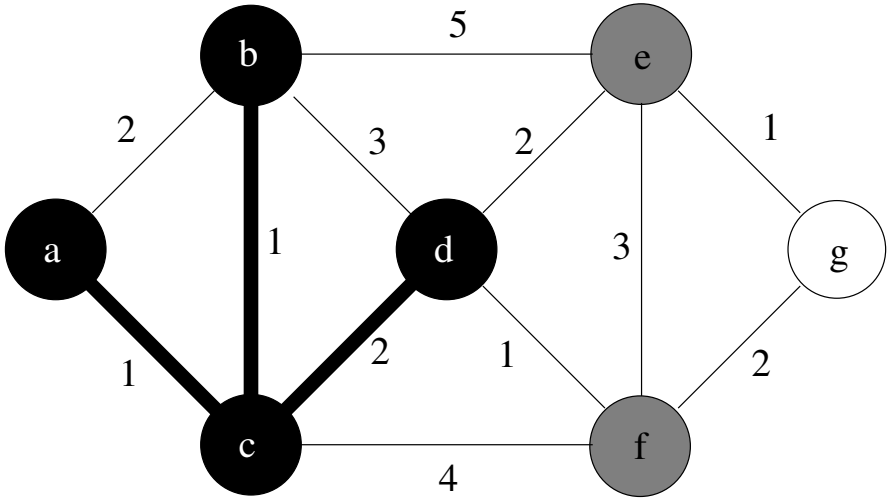
Q:



Step 5:

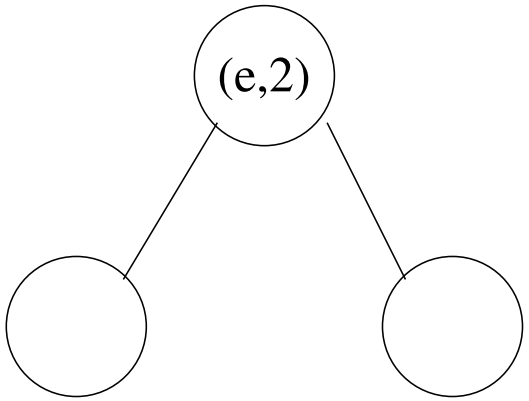
- Extract-Min (returns $(f, 1)$)

Example



	a	b	c	d	e	f	g
d	0	1	1	2	2	1	-
π	-	c	a	c	d	d	-

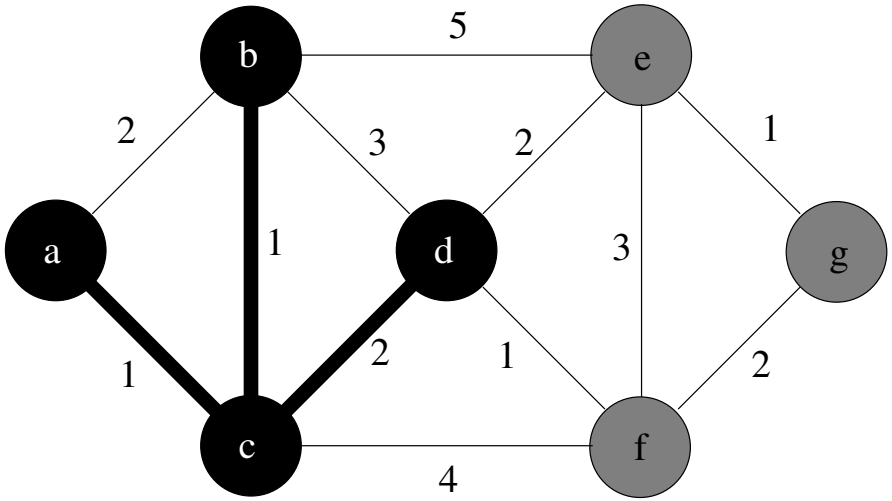
Q:



Step 5:

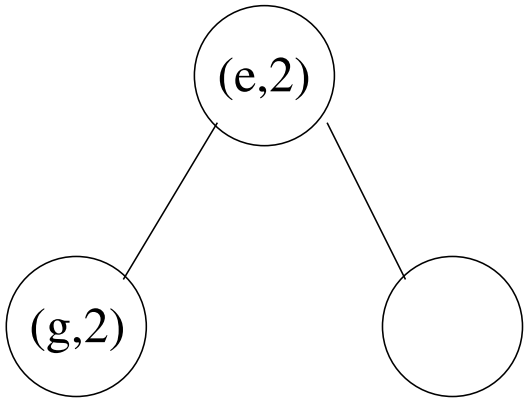
- Extract-Min (returns $(f, 1)$)
- update e (no change)

Example



	a	b	c	d	e	f	g
d	0	1	1	2	2	1	2
π	-	*c*	*a*	*c*	*d*	*d*	*f*

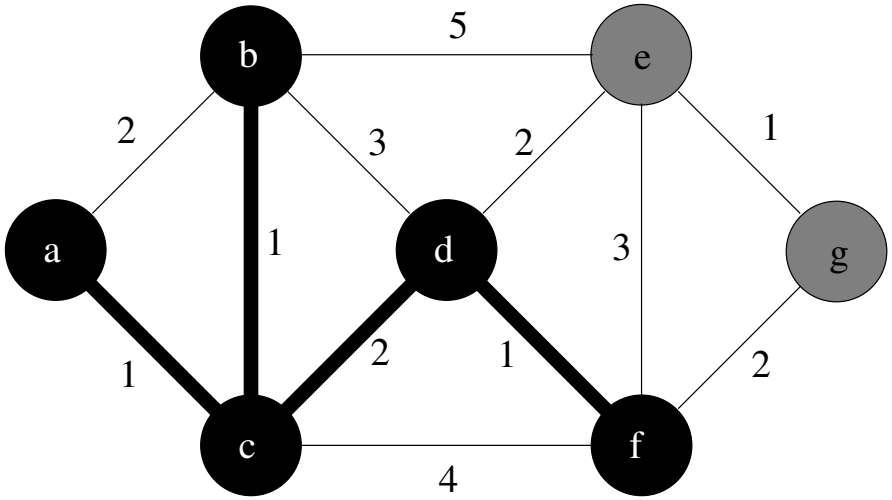
Q:



Step 5:

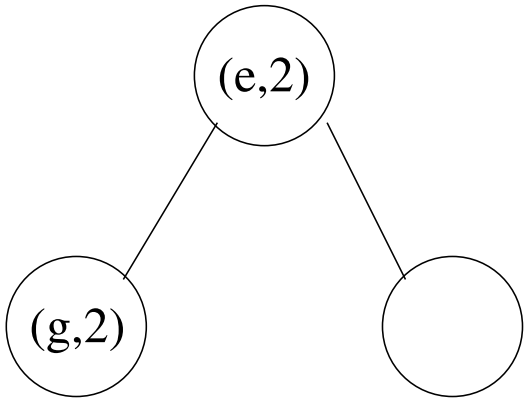
- Extract-Min (returns $(f, 1)$)
- update e (no change)
- add g

Example



	a	b	c	d	e	f	g
d	0	1	1	2	2	1	2
π	-	c	a	c	d	d	f

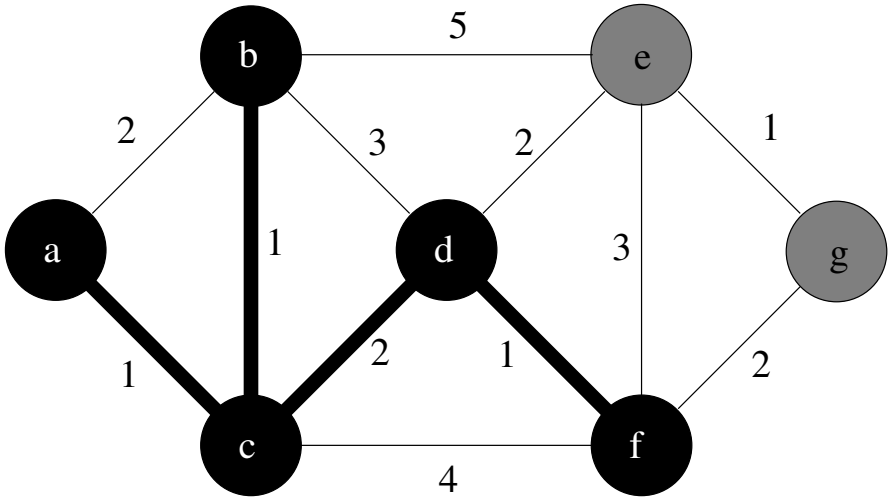
Q:



Step 5:

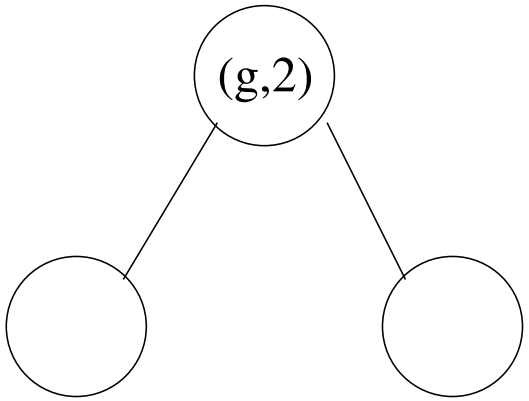
- Extract-Min (returns $(f, 1)$)
- update e (no change)
- add g
- color f black

Example



	a	b	c	d	e	f	g
d	0	1	1	2	2	1	2
π	-	*c*	*a*	*c*	*d*	*d*	*f*

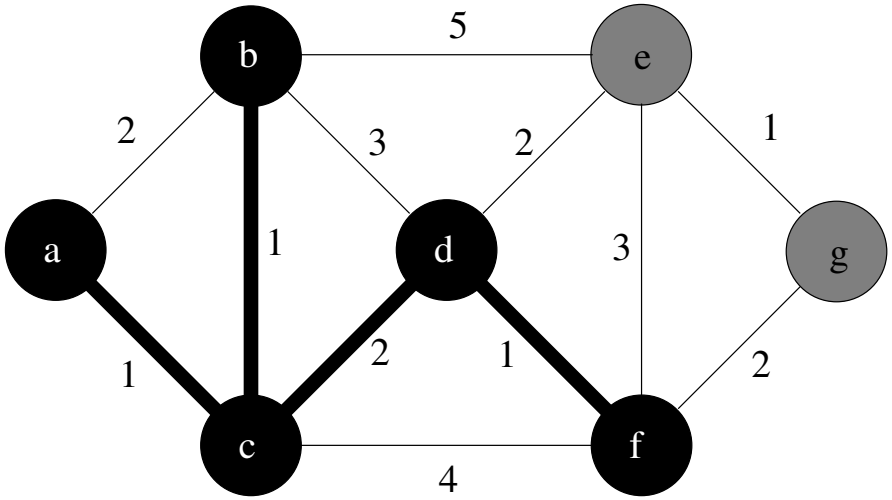
Q:



Step 6:

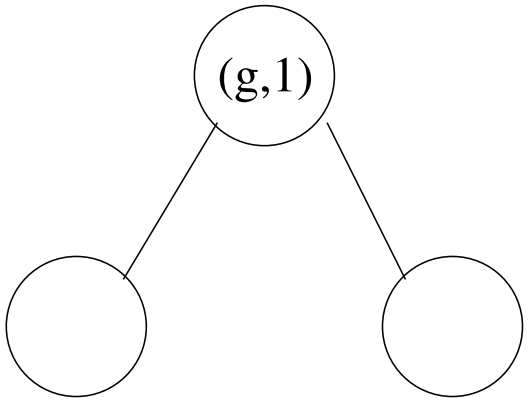
- Extract-Min (returns $(e, 2)$)

Example



	a	b	c	d	e	f	g
d	0	1	1	2	2	1	1
π	-	*c*	*a*	*c*	*d*	*d*	*e*

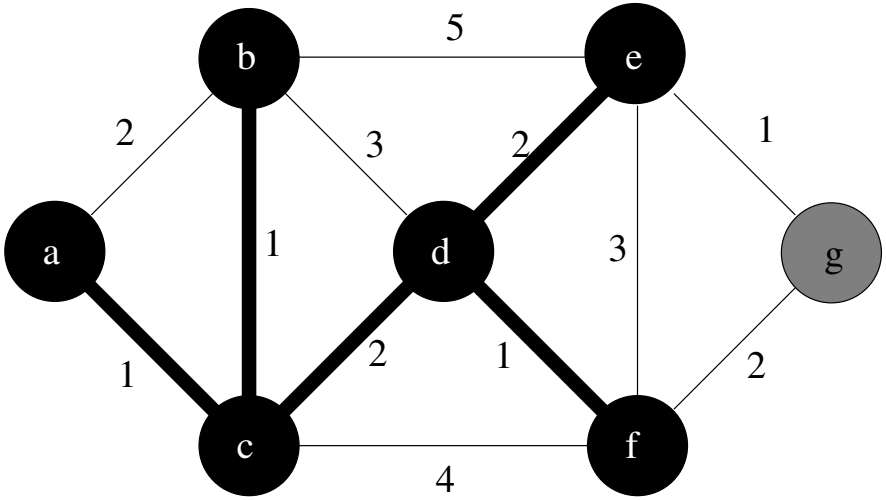
Q:



Step 6:

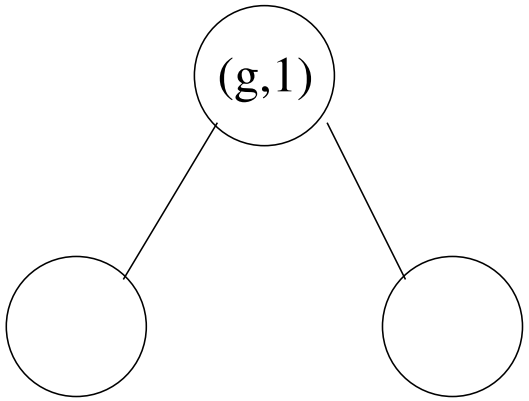
- Extract-Min (returns $(e, 2)$)
- update g

Example



	a	b	c	d	e	f	g
d	0	1	1	2	2	1	1
π	-	*c*	*a*	*c*	*d*	*d*	*e*

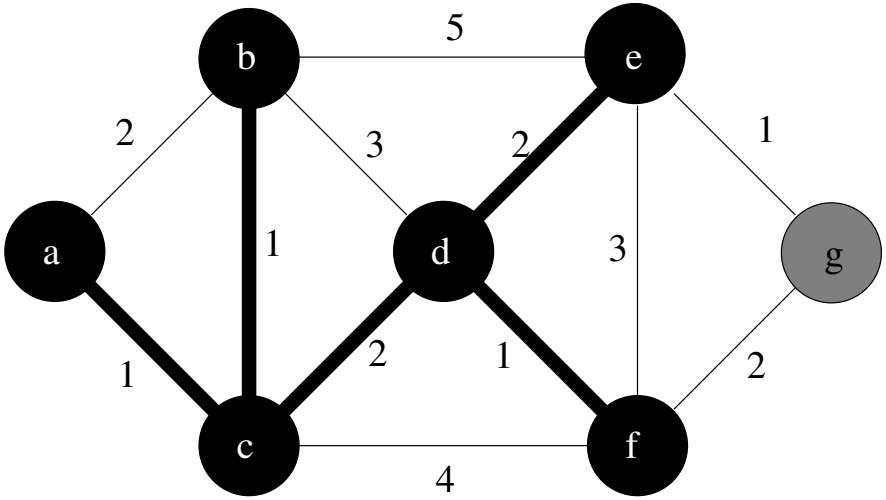
Q:



Step 6:

- Extract-Min (returns $(e, 2)$)
- update g
- color e black

Example



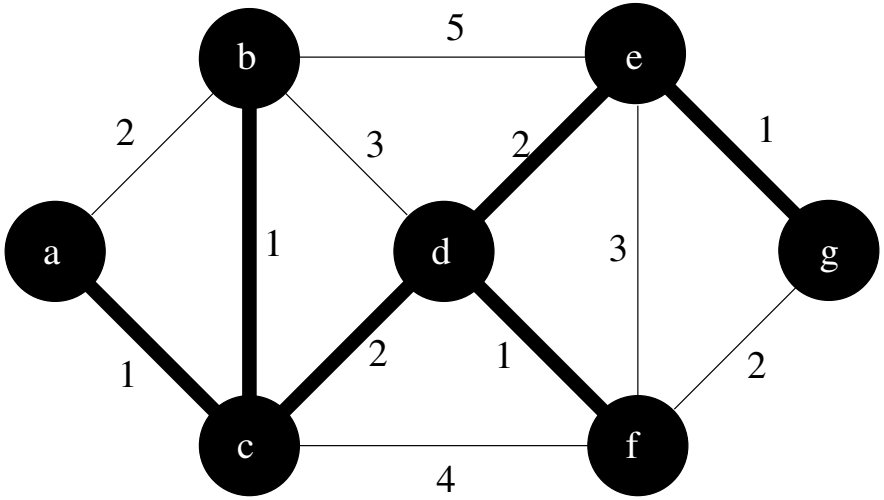
	a	b	c	d	e	f	g
d	0	1	1	2	2	1	1
π	-	c	a	c	d	d	e

Q: (empty)

Step 7:

- Extract-Min (returns $(g, 1)$)

Example



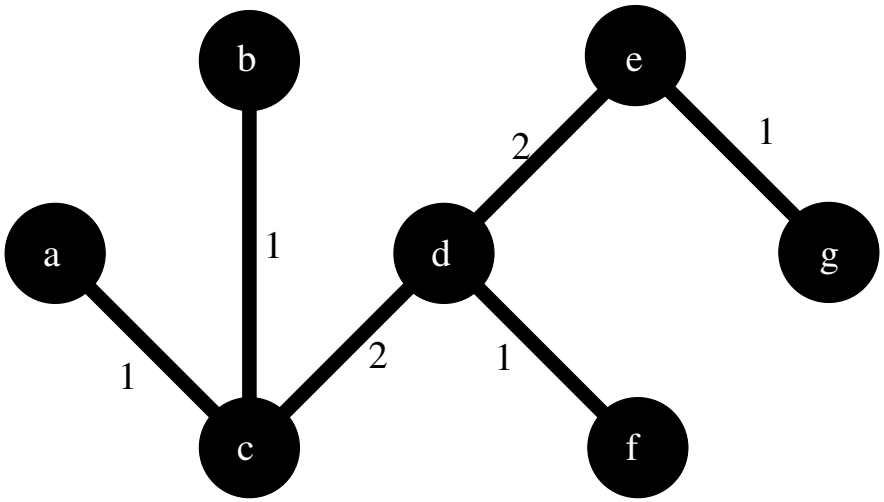
	a	b	c	d	e	f	g
d	0	1	1	2	2	1	1
π	-	c	a	c	d	d	e

Q: (empty)

Step 7:

- Extract-Min (returns $(g, 1)$)
- color g black — done!

Example



	a	b	c	d	e	f	g
d	0	1	1	2	2	1	1

π	-	c	a	c	d	d	e

Q: (empty)

Step 7:

- Extract-Min (returns $(g, 1)$)
- color g black — done!

Eg. one MST (total cost is 8):

$$\{(\pi(b), b), (\pi(c), c), (\pi(d), d), (\pi(e), e), (\pi(f), f), (\pi(g), g)\}$$
$$= \{(c, b), (a, c), (c, d), (d, e), (d, f), (e, g)\}$$

Termination and Efficiency

Claim:

If MST-Prim is executed on a weighted undirected graph $G = (V, E)$ then the algorithm terminates after performing $O((|V| + |E|) \log |V|)$ steps in the worst case.

Proof.

This is virtually identical to the proof of the corresponding result for Dijkstra's algorithm (to compute minimum-cost paths).

- The number of operations on the priority queue, and the number of operations that do not involve this data structure, are each in $\Theta(|V| + |E|)$ in the worst case (by the argument that has been applied to the last three algorithms considered).

Termination and Efficiency (cont.)

Proof (continued).

- Since the size of the priority queue never exceeds $|V|$ and since the only operations on the priority queue used are insertions, decreases of key values, and extractions of the minimum (top priority) element, the cost of each operation on the data structure is in $O(\log |V|)$.
- It follows immediately that the total number of steps is in $O((|V| + |E|) \log |V|)$, as claimed. □

$O(|V| \log |V| + |E|)$ using a Fibonacci heap (amortized)

Additional Comments

On Greedy Algorithms

- Prim's algorithm is an example of a **greedy** algorithm: A "global" *optimization* problem (finding a minimum-cost spanning tree) is solved by making a sequence of "local" *greedy choices* (by extending a tree with edges whose weights are as small as possible).
- Proving correctness of greedy algorithms is often challenging. Indeed, greedy heuristics are often *incorrect*.
- On the other hand, *when they are correct*, greedy algorithms are frequently simpler and more efficient than other algorithms for the same computation.
- See CPSC 413 for more about greedy algorithms!

References

Further Reading and Java Code:

- **Introduction to Algorithms**, Chapter 23
- Chapter 23 includes Prim's algorithm along with another greedy algorithm for this problem (Kruskal's algorithm).
- **Data Structures & Algorithms in Java**, Chapter 10.6