Chapter 3: Algorithm Design Strategies

(Part II)

Contents

Traversal techniques

Graph traversal

Process of visiting each vertex in a graph

Given a graph, $G = (V, E)$, and a vertex, $v \in V(G)$, visit all vertices in G that are reachable from v

2 ways of doing this:

1. Depth-first search (DFS)
2. Breadth-first search (BFS)

Depth-first search (DFS)

Process all descendants of a vertex before we move to an adjacent vertex.

DFS in a graph is similar to DFS in a tree. Since graphs may contain cycles unlike trees, we may come to the same node again. To avoid processing a node more than once, we keep track of visited nodes.

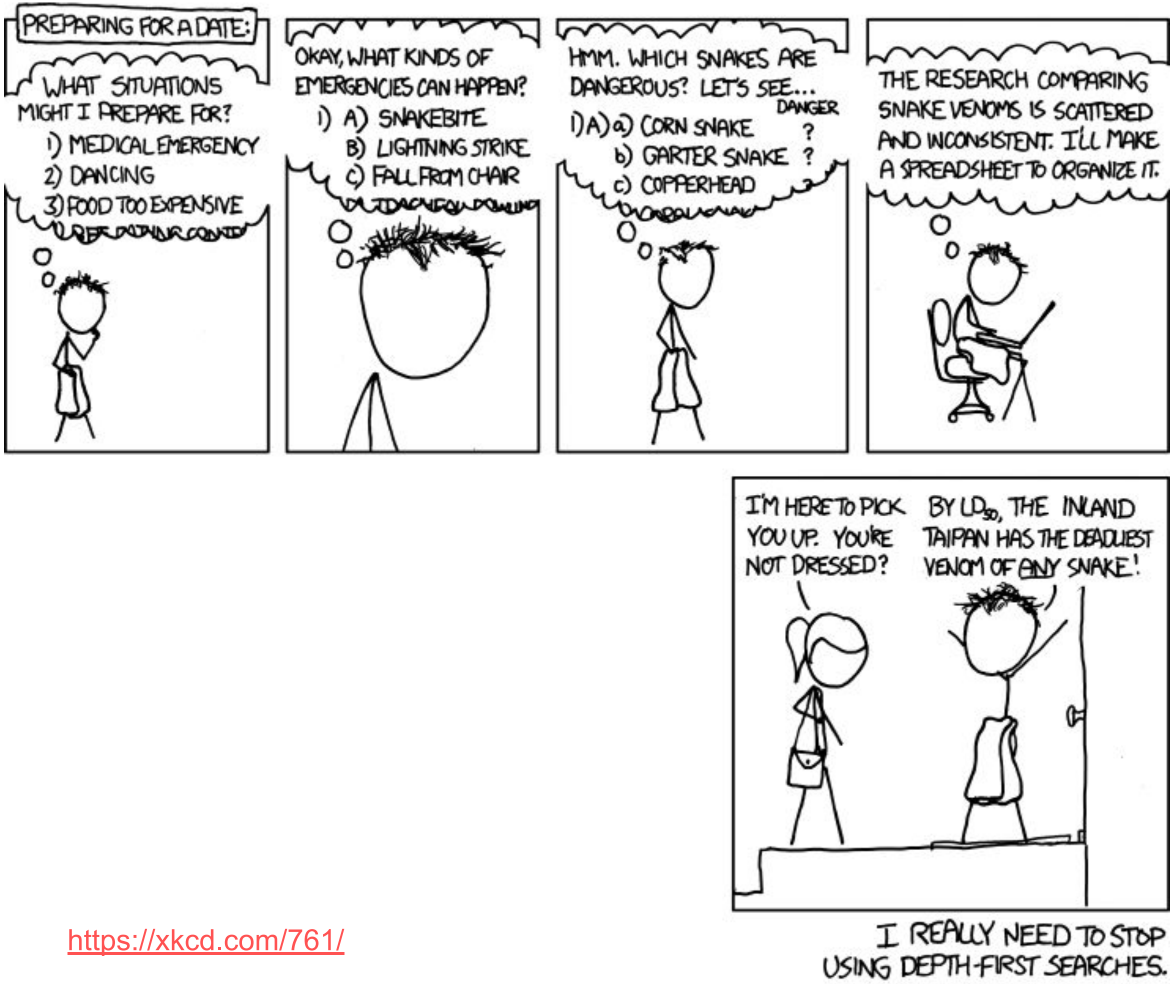
Uses a stack data structure to perform the search.

Depth-first search (DFS)

Basic idea:

1. Start by putting any one of the graph's vertices (starting vertex) on top of a stack.
2. Pop the topmost item from the stack and add it to the visited list.
3. Push the popped vertex’s unvisited neighbors into the top of stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

DFS



https://xkcd.com/761/

Depth-first search (DFS)

Algorithm: (Recursive) **DFS(G, s)**

Input: A graph, G, and a starting vertex, s

Output: A sequence of processed vertices

Steps:

1. mark(s); // Mark s as visited
2. $\forall (s, v) \in E(G)$
    a. DFS (G, v)

Depth-first search (DFS)

Algorithm: (Iterative) **DFS(G, s)**

Input: A graph, G, and a starting vertex, s

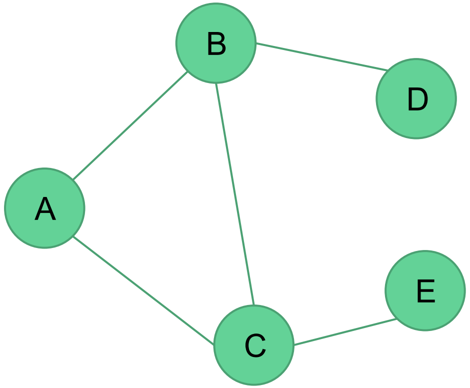
Output: A sequence of processed vertices

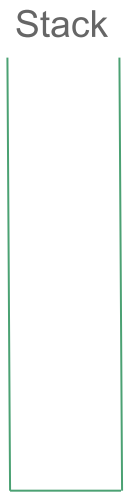
Steps:

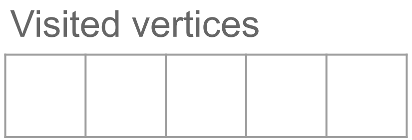
1. mark(s); // Mark s as visited
2. $L \coloneqq \{s\}$ // Push s into the stack
3. **while** $L \neq \emptyset$ **do**
   a. $u \coloneqq \mathrm{last}(L)$ // Top of the stack
   b. **if** $\exists\,(u, v)$ such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); $L \coloneqq L \cup \{v\}$
   c. else
      i. $L \coloneqq L \setminus \{u\}$ // Pop from the stack
   d. endif
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A

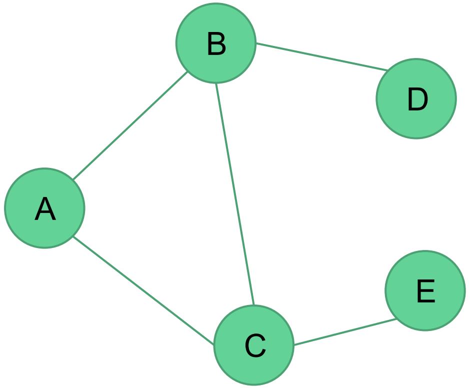




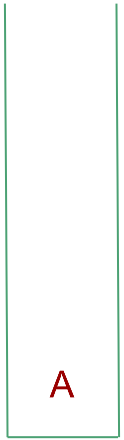


Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack



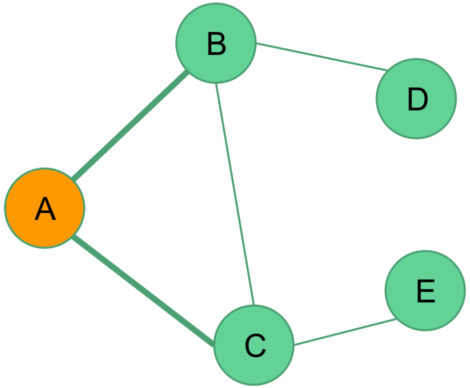
Visited vertices

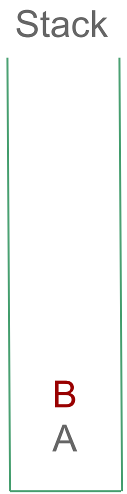
A				

1. mark(s); // Mark s as visited
2. L ≔ {s} // Push s into the stack

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A





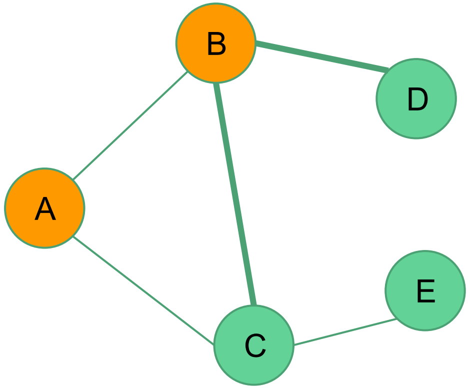
Visited vertices

A	B			

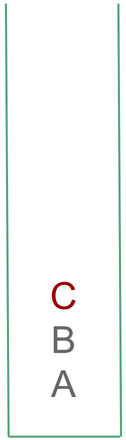
3. **while** L ≠ ∅ **do**
   a. u ≔ last(L) // Top of the stack
   b. **if** ∃ (u, v) such that v is unmarked **then**
      // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Pop from the stack
   d. **endif**
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack



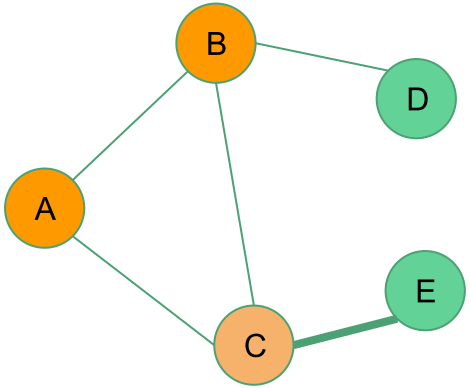
Visited vertices

A	B	C		

3. **while** L ≠ ∅ **do**
   a. u ≔ last(L) // Top of the stack
   b. **if** ∃ (u, v) such that v is unmarked **then**
      // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Pop from the stack
   d. **endif**
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack

E
C
B
A

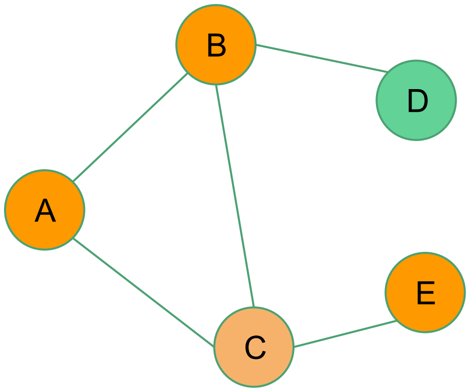
Visited vertices

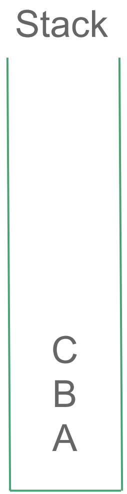
A	B	C	E	

3. **while** $L \neq \varnothing$ **do**
   a. $u \coloneqq \text{last}(L)$ // Top of the stack
   b. **if** $\exists\,(u, v)$ such that v is unmarked **then**
      // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); $L \coloneqq L \cup \{v\}$
   c. else
      i. $L \coloneqq L \setminus \{u\}$ // Pop from the stack
   d. endif
4. endwhile

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A





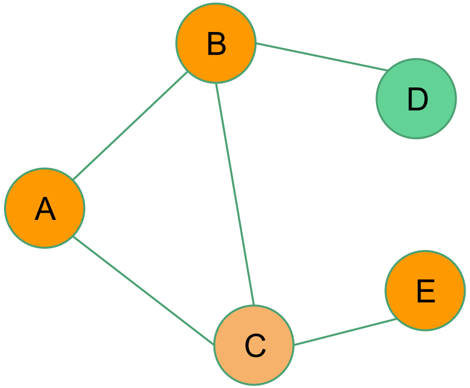
Visited vertices

A	B	C	E	

3. **while** L ≠ ∅ **do**
   a. u ≔ last(L) // Top of the stack
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Pop from the stack
   d. **endif**
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack

B
A

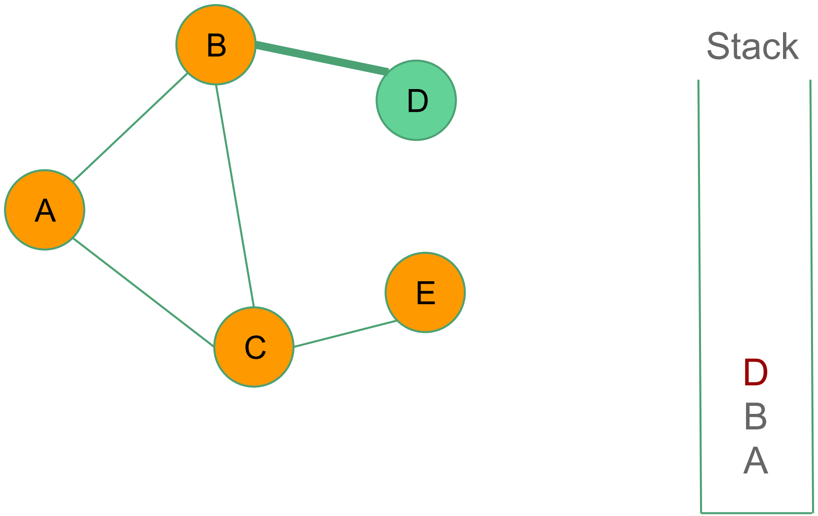
Visited vertices

A	B	C	E	

3. **while** L ≠ ∅ **do**
   a. u ≔ last(L) // Top of the stack
   b. **if** ∃ (u, v) such that v is unmarked **then**
      // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. else
      i. L ≔ L ∖ {u} // Pop from the stack
   d. endif
4. endwhile

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



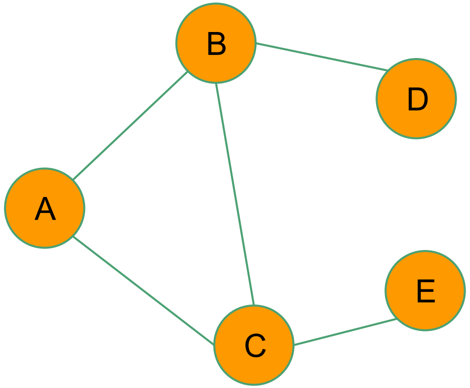
Visited vertices

A	B	C	E	D

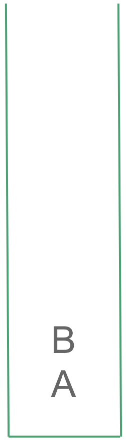
3. **while** L ≠ ∅ **do**
   a. u ≔ last(L) // Top of the stack
   b. **if** ∃ (u, v) such that v is unmarked **then**
      // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Pop from the stack
   d. **endif**
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack



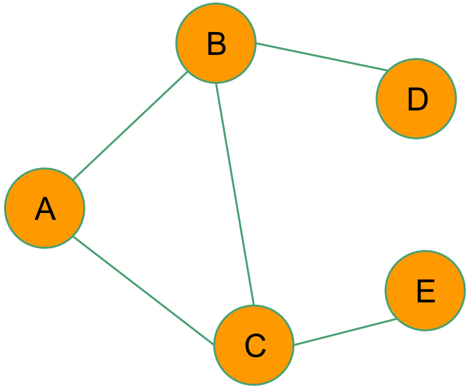
Visited vertices

A	B	C	E	D

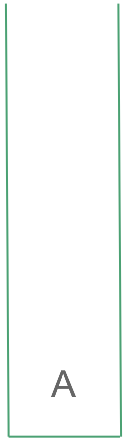
3. **while** L ≠ ∅ **do**
   a. u ≔ last(L) // Top of the stack
   b. **if** ∃ (u, v) such that v is unmarked **then**
      // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Pop from the stack
   d. **endif**
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack



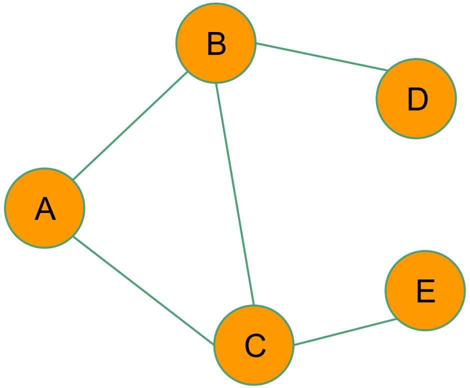
Visited vertices

A	B	C	E	D

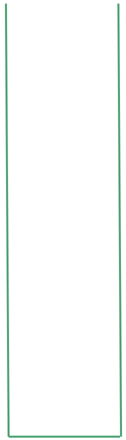
3. **while** $L \neq \emptyset$ **do**
   a. $u \coloneqq last(L)$ // Top of the stack
   b. **if** $\exists (u, v)$ such that v is unmarked **then**
      // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); $L \coloneqq L \cup \{v\}$
   c. else
      i. $L \coloneqq L \setminus \{u\}$ // Pop from the stack
   d. endif
4. endwhile

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack



Visited vertices

A	B	C	E	D

3. **while** L $\neq \varnothing$ **do**
   - a. u ≔ last(L) // Top of the stack
   - b. **if** $\exists$ (u, v) such that v is unmarked **then**
     // Find neighbors of u
     - i. choose v of the smallest index;
     - ii. mark(v); L ≔ L $\cup$ {v}
   - **c. else**
     - i. L ≔ L $\setminus$ {u} // Pop from the stack
   - **d. endif**
4. **endwhile**

Applications of DFS

- Finding a minimum spanning tree for unweighted graphs
- Detecting a cycle in the graph
- Finding a path from one node to another
- Topological ordering: determining the order of compilation tasks, resolving symbol dependencies in linkers etc.
- Solving problems with only one solution, such as maze
- etc.

Breadth-first search (BFS)

Process all adjacent vertices of a vertex before going to the next level.

Uses a queue data structure to perform the search.

Breadth-first search (BFS)

Basic idea:

1. Start by putting any one of the graph's vertices (starting vertex) at the back of a queue.
2. Dequeue the queue (take the vertex at the front of the queue) and add it to the visited list.
3. Enqueue the dequeued vertex’s unvisited neighbours to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

Breadth-first search (BFS)

Algorithm: **BFS(G, s)**

Input: A graph, G, and a starting vertex, s

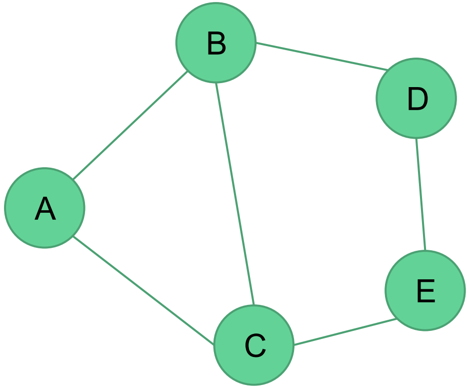
Output: A sequence of processed vertices

Steps:

1. mark(s); // Mark s as visited
2. $L \coloneqq \{s\}$ // Push s into the queue
3. **while** $L \neq \varnothing$ **do**
   a. $u \coloneqq \text{first}(L)$ // Front of the queue
   b. **if** $\exists\,(u, v)$ such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); $L \coloneqq L \cup \{v\}$
   c. **else**
      i. $L \coloneqq L \setminus \{u\}$ // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A				

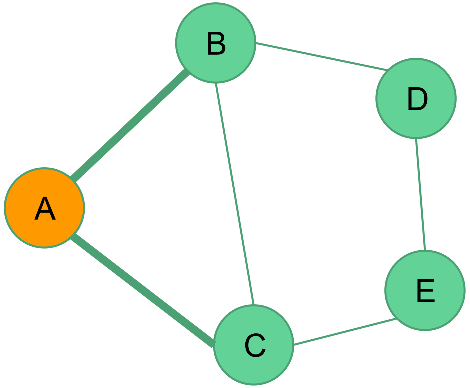
List

A				

```
mark(s);      // Mark s as visited
L ≔ {s}       // Push s into the queue
```

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B			

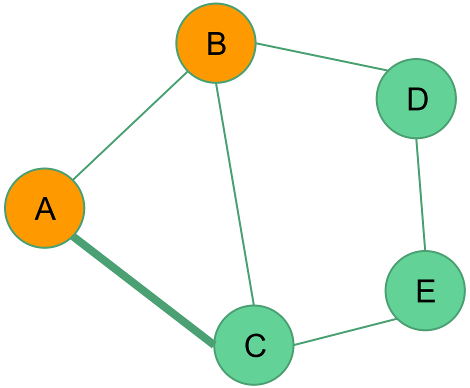
List

A	B			

3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C		

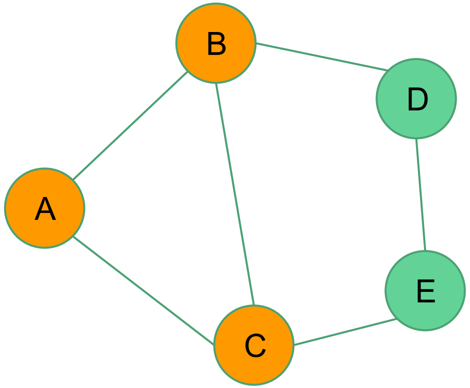
List

A	B	C		

3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C		

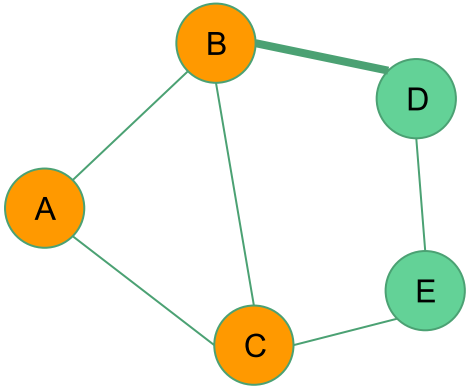
List

B	C			

3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C	D	

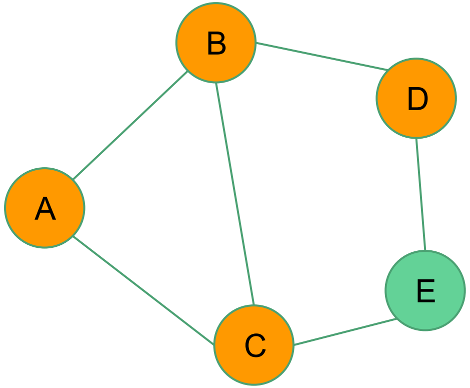
List

B	C	D		

3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C	D	

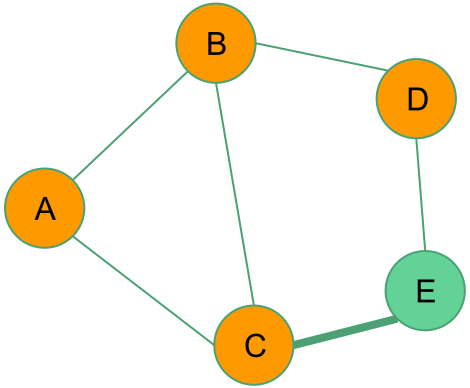
List

C	D			

3. **while** L $\neq \emptyset$ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** $\exists$ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L $\cup$ {v}
   c. **else**
      i. L ≔ L∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C	D	E

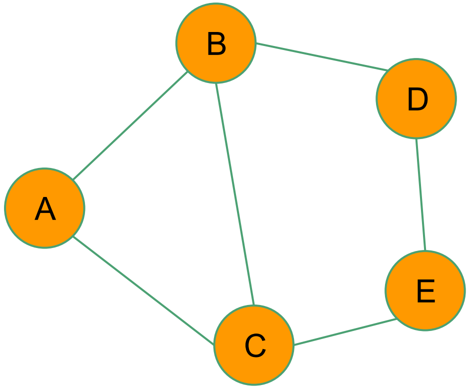
List

C	D	E		

3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C	D	E

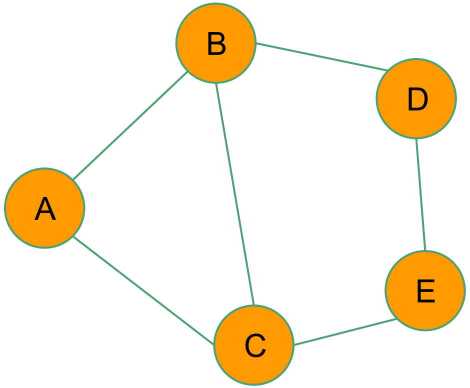
List

D	E			

3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C	D	E

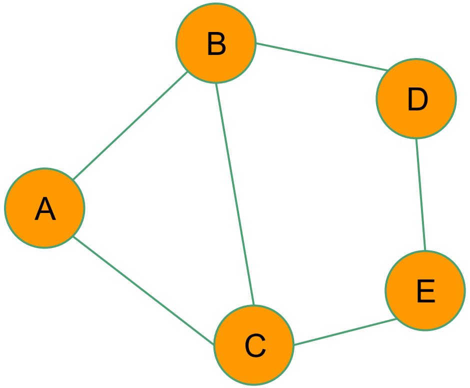
List

E				

3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C	D	E

List

3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Applications of BFS

- Finding a minimum spanning tree for unweighted graphs
- Web crawler: Begin from a starting page and follow all links from this page and keep doing the same
- Social networks: Find people within a given distance ‘k’ from a person
- Finding the shortest path to another node
- GPS navigation systems: Finding the direction to reach from one place to another
- etc.

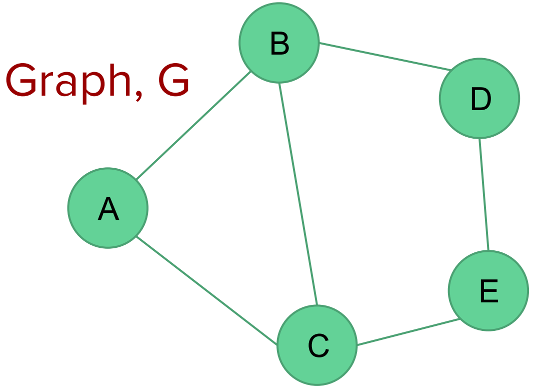
Minimum spanning tree

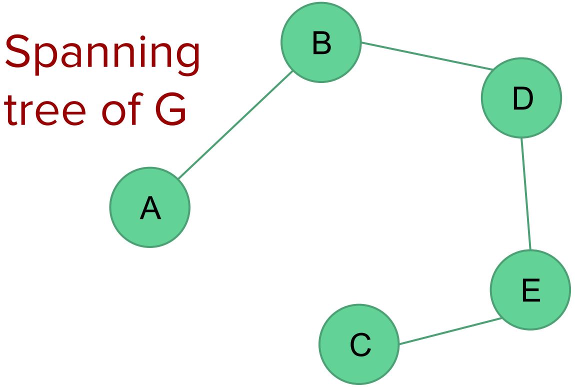
Spanning tree

A spanning tree of a connected graph G is a tree that consists solely of edges in G and that includes all of the vertices in G.

Our solution to generate a spanning tree must satisfy the following constraints:

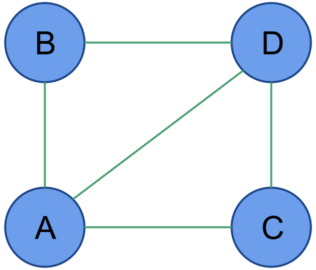
1. We must use only edges within the graph
2. We must use exactly n-1 edges
3. We may not use edges that would produce a cycle



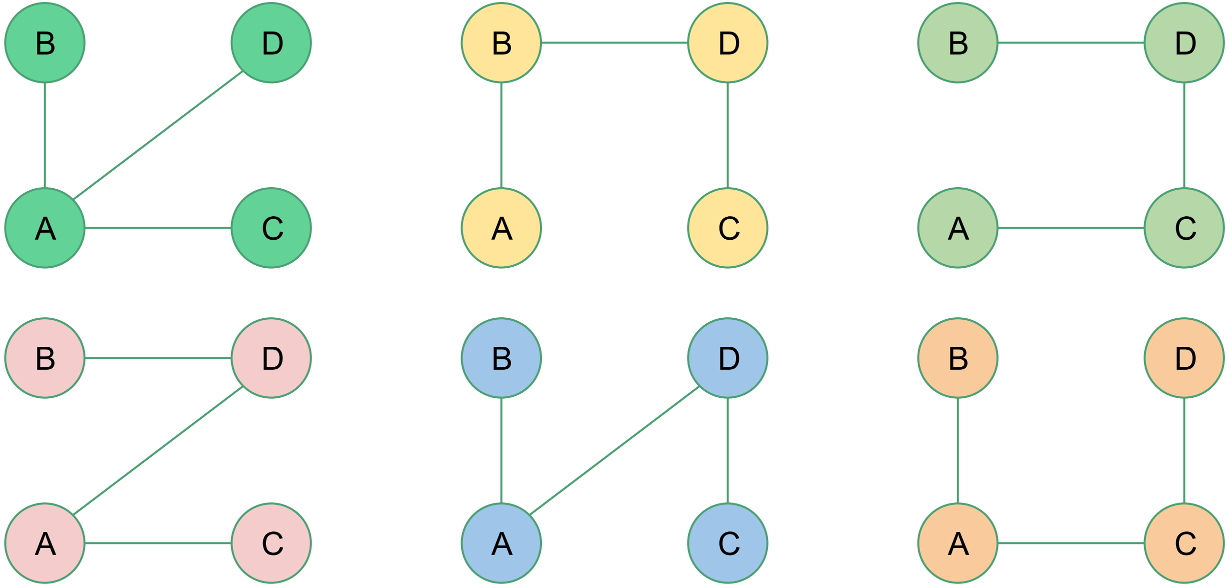


Spanning tree

A single graph can have many spanning trees.



Graph G



Spanning trees of G

Spanning tree

A spanning tree can be generated using a DFS or a BFS. The spanning tree is formed from those edges traversed during the search.

- If a breadth first search is used, the resulting spanning tree is called a **breadth first spanning tree**.
- If a depth first search is used, it is called **depth first spanning tree**.

For a disconnected / disjoint graph, a **spanning forest** is defined.

Minimum spanning tree (MST)

A **minimum spanning tree of a weighted graph** is a **spanning tree of least weight**, i.e. a spanning tree in which the total weight of the edges is guaranteed to be the minimum of all possible trees in the graph.

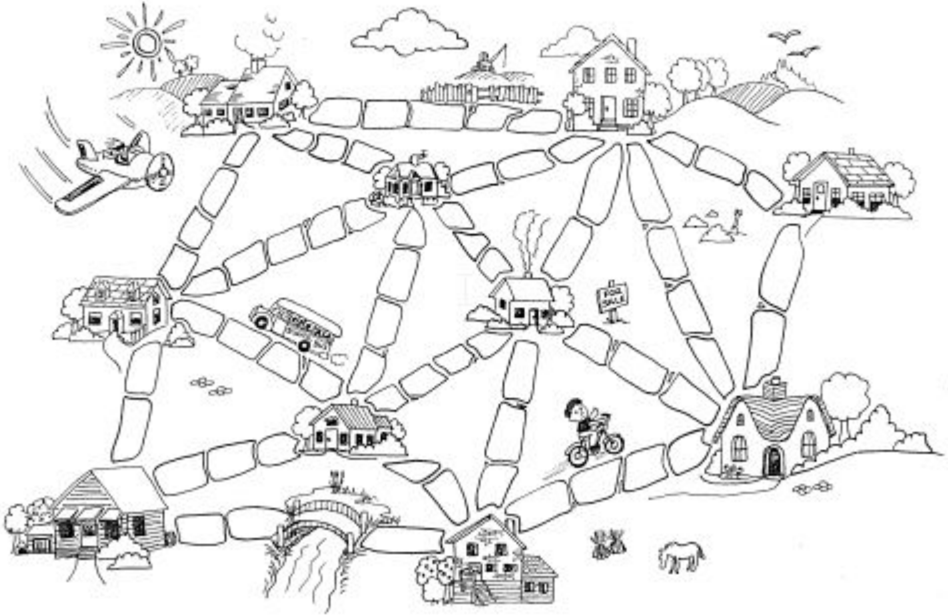
If the weights in the network/graph are unique, there is only one MST

If there are duplicate weights, there may be one or more MSTs

Application: Network design, Muddy city problem

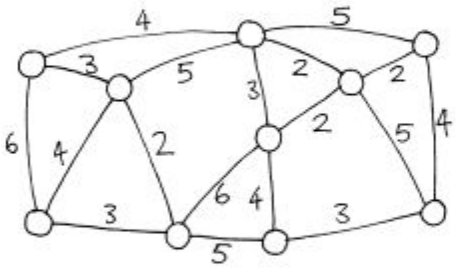
Muddy city problem

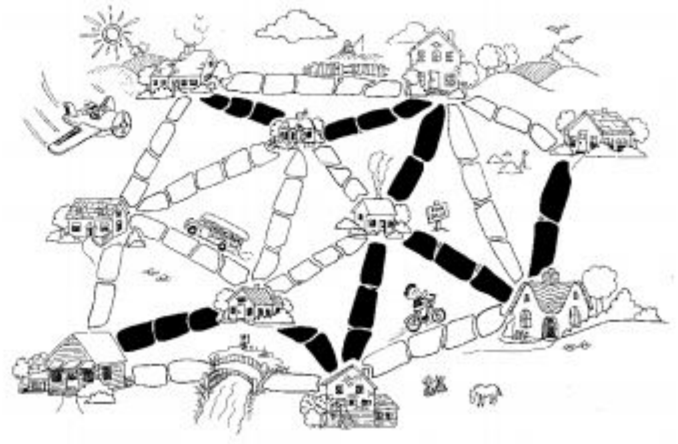
- A city with no paved road
- The mayor of the city decided to pave some of the streets with the following two conditions:
    1. Enough streets must be paved so that it is possible for everyone to travel from their house to anyone else's house only along paved roads, and
    2. The paving should cost as little as possible.



Muddy city problem

Solution: Minimum spanning tree





Growing a MST

Problem:
Given a connected, undirected graph $G = (V, E)$ with a weight function $w:E \rightarrow R$, find a minimum spanning tree for G

A greedy strategy:
Grow the minimum spanning tree one edge at a time

- Kruskal's algorithm
- Prim's algorithm

Growing a MST

```
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
```

Kruskal's algorithm: Finds a safe edge by finding, of all the edges that connect any two trees in the forest, an edge of least weight.

Prim's algorithm: Adds to the tree A a light edge that connects A to an isolated vertex

Disjoint set

A group of sets where no item can be in more than one set.

A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.

Example:

$S= \{\{a\}, \{b\}, \{c,d\}, \{e, f, g, h\}\}$ is a disjoint-set.

Basic disjoint set operations

Make_set (x)

Creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

Union (x, y)

Unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets.

Find_set(x)

Returns a pointer to the representative of the (unique) set containing x.

Kruskal’s algorithm (using disjoint sets)

MST-KRUSKAL(G, w)

$A = \emptyset$
for each vertex $v \in G.V$
    MAKE-SET(v)
sort the edges of $G.E$ into nondecreasing order by weight w
for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
    if FIND-SET$(u) \neq$ FIND-SET(v)
        $A = A \cup \{(u, v)\}$
        UNION(u, v)
return A

Disjoint set operations

Make-set(x) :

Creates a new set whose only member is x

Union(x, y):

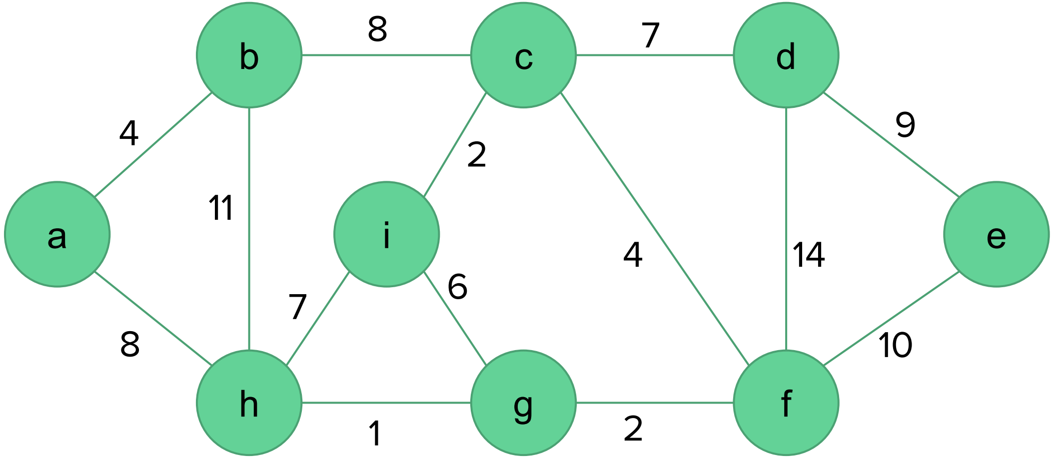
Unites the dynamic sets that contain x and y into a new set that is the union of these two sets

Find-Set(x):

Returns a pointer to the representative of the set containing x

Example

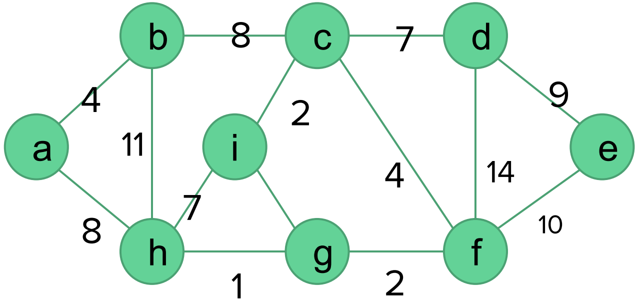
Find a minimum spanning tree of the following graph



Example

MST-KRUSKAL(G, w)

1 $A = \emptyset$
2 **for** each vertex $v \in G.V$
3   MAKE-SET(v)
4 sort the edges of $G.E$ into nondecreasing order by weight w
5 **for** each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
6   **if** FIND-SET$(u) \neq$ FIND-SET(v)
7         $A = A \cup \{(u, v)\}$
8   UNION(u, v)
9 **return** A

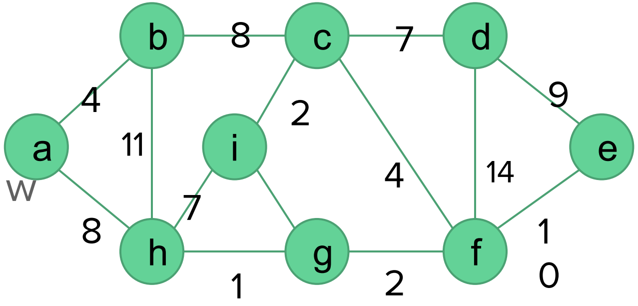


L1: A = {}

L2: {a} {b} {c} {d} {e} {f} {g} {h} {i}

Example

L4: sort the edges of G:E into nondecreasing order by weight w




```
 1  (h,g)
 2  (g,f)
 2  (i,c)
 4  (a,b)
 4  (c,f)
 6  (i,g)
 7  (h,i)
 7  (c,d)
 8  (b,c)
 8  (a,h)
 9  (d,e)
10  (e,f)
11  (b,h)
14  (d,f)
```

Example

w	Edge (u,v)	Disjoint sets	A
1 2 2 4 4 6 7 7 8 8 9 10 11 14	(h,g) (g,f) (i,c) (a,b) (c,f) (i,g) (h,i) (c,d) (b,c) (a,h) (d,e) (e,f) (b,h) (d,f)	{a} {b} {c} {d} {e} {f} **{g,h}** {i}	{(h,g)}

```
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)
```

w	Edge (u,v)	Disjoint sets	A
1	(h,g)	{a} {b} {c} {d} {e} {f} **{g,h}** {i}	{ **(h,g)** }
2	(g,f)		
2	(i,c)		
4	(a,b)		
4	(c,f)		
6	(i,g)		
7	(h,i)		
7	(c,d)		
8	(b,c)		
8	(a,h)		
9	(d,e)		
10	(e,f)		
11	(b,h)		
14	(d,f)		

```
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)
```

w	Edge (u,v)	Disjoint sets	A
1	(h,g)	{a} {b} {c} {d} {e} {f} {g,h} {i}	{(h,g)}
2	(g,f)	{a} {b} {c} {d} {e} **{f,g,h}** {i}	{(h,g), **(g,f)**}
2	(i,c)		
4	(a,b)		
4	(c,f)		
6	(i,g)		
7	(h,i)		
7	(c,d)		
8	(b,c)		
8	(a,h)		
9	(d,e)		
10	(e,f)		
11	(b,h)		
14	(d,f)		

```
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)
```

w	Edge (u,v)	Disjoint sets	A
1	(h,g)	{a} {b} {c} {d} {e} {f} {g,h} {i}	{(h,g)}
2	(g,f)	{a} {b} {c} {d} {e} {f,g,h} {i}	{(h,g),(g,f)}
2	(i,c)	{a} {b} **{c,i}** {d} {e} {f,g,h}	{(h,g),(g,f),**(c,i)**}
4	(a,b)		
4	(c,f)		
6	(i,g)		
7	(h,i)		
7	(c,d)		
8	(b,c)		
8	(a,h)		
9	(d,e)		
10	(e,f)		
11	(b,h)		
14	(d,f)		

for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
  if FIND-SET$(u) \neq$ FIND-SET(v)
    $A = A \cup \{(u, v)\}$
    UNION(u, v)

w	Edge (u,v)	Disjoint sets	A
1	(h,g)	{a} {b} {c} {d} {e} {f} {g,h} {i}	{(h,g)}
2	(g,f)	{a} {b} {c} {d} {e} {f,g,h} {i}	{(h,g),(g,f)}
2	(i,c)	{a} {b} {c,i} {d} {e} {f,g,h}	{(h,g),(g,f),(c,i)}
4	(a,b)	**{a,b}** {c,i} {d} {e} {f,g,h}	{(h,g),(g,f),(c,i), **(a,b)**}
4	(c,f)	{a,b} **{c,f,g,h,i}** {d} {e}	{(h,g),(g,f),(c,i),(a,b), **(c,f)**}
6	(i,g)	Find-set(i) == Find-set(g)	
7	(h,i)		
7	(c,d)		
8	(b,c)		
8	(a,h)		
9	(d,e)		
10	(e,f)		
11	(b,h)		
14	(d,f)		

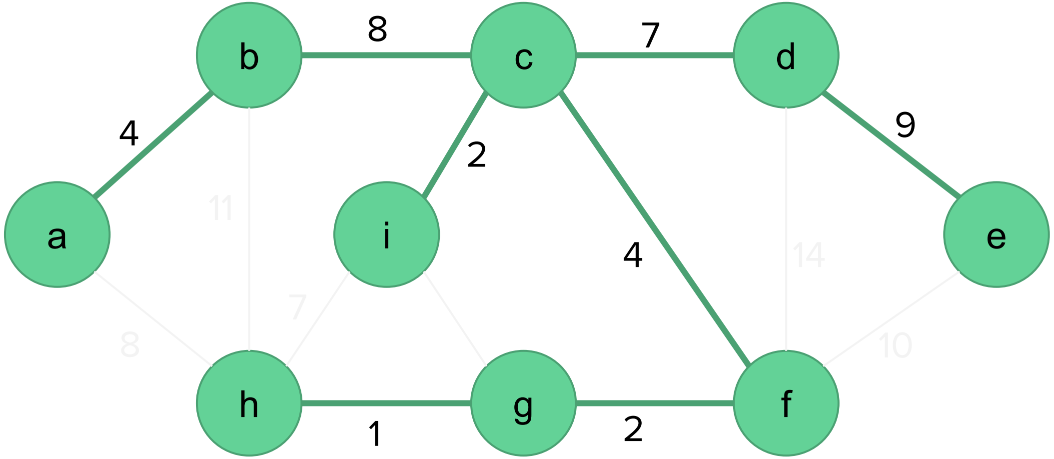
```
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
```

w	Edge (u,v)	Disjoint sets	A
1	(h,g)	{a} {b} {c} {d} {e} {f} {g,h} {i}	{(h,g)}
2	(g,f)	{a} {b} {c} {d} {e} {f,g,h} {i}	{(h,g),(g,f)}
2	(i,c)	{a} {b} {c,i} {d} {e} {f,g,h}	{(h,g),(g,f),(c,i)}
4	(a,b)	{a,b} {c,i} {d} {e} {f,g,h}	{(h,g),(g,f),(c,i),(a,b)}
4	(c,f)	{a,b} {c,f,g,h,i} {d} {e}	{(h,g),(g,f),(c,i),(a,b),(c,f)}
6	(i,g)	Find-set(i) == Find-set(g)	
7	(h,i)	Find-set(h) == Find-set(i)	
7	(c,d)	{a,b} **{c,d,f,g,h,i}** {e}	{(h,g),(g,f),(c,i),(a,b),(c,f),**(c,d)**}
8	(b,c)	**{a,b,c,d,f,g,h,i}** {e}	{(h,g),(g,f),(c,i),(a,b),(c,f),(c,d), **(b,c)** }
8	(a,h)	Find-set(a) == Find-set(h)	
9	(d,e)	**{a,b,c,d,e,f,g,h,i}**	{(h,g),(g,f),(c,i),(a,b),(c,f),(c,d),(b,c),**(d,e)**}
10	(e,f)	Find-set(e) == Find-set(f)	
11	(b,h)	Find-set(b) == Find-set(h)	
14	(d,f)	Find-set(d) == Find-set(f)	

Example

The MST is the tree containing A.

`A= {(h,g),(g,f),(c,i),(a,b),(c,f),(c,d),(b,c),(d,e)}`



Analysis of Kruskal's algorithm

```
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
```

Analysis of Kruskal’s algorithm

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

Line 1: O(1)

Line 2 - 3: O(|V|)

Line 4: Sorting: O(|E| log |E|)

Line 5: O(|E|)

Line 6 - 8: Depends on the implementation of Find-Set and Union operations. We assume that they are very fast (O(1))

Overall complexity = O(1 + V + E log E + E)
= O (E log E)

Prim’s algorithm

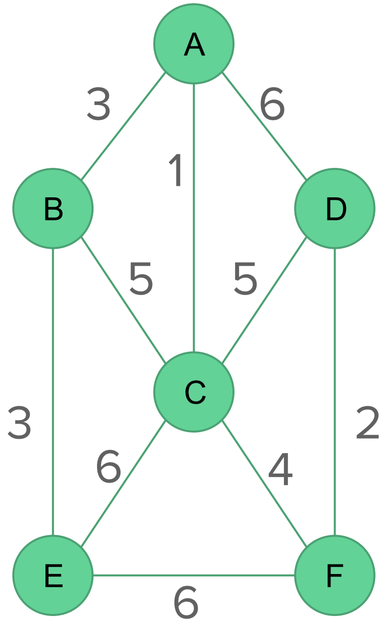
Grows a single tree and adds a light edge (edge with the lowest weight) in each iteration

Steps:

1. Start by picking any vertex to be the root of the tree.
2. While the tree does not contain all vertices in the graph, find shortest edge leaving the tree and add it to the tree

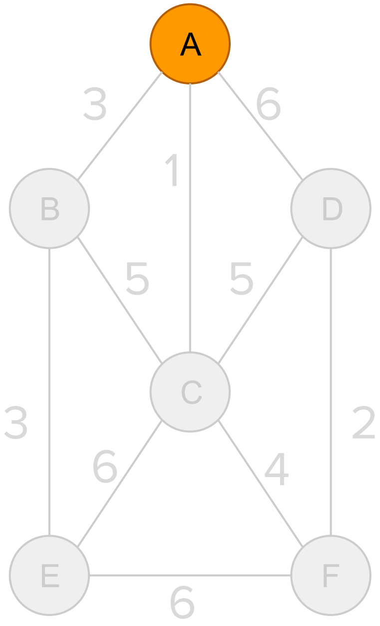
Prim’s algorithm

Example: Find a minimum spanning tree of the following graph



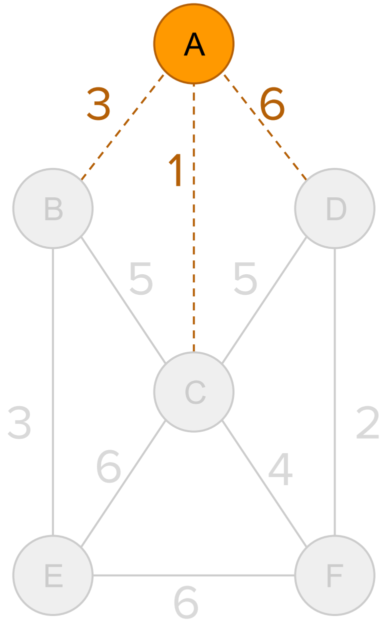
Prim’s algorithm

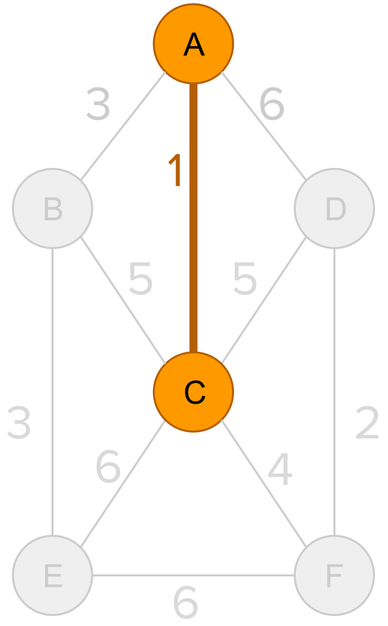
Step 1: Pick any vertex to be the root of the tree. Let’s say A will be the root



Prim’s algorithm

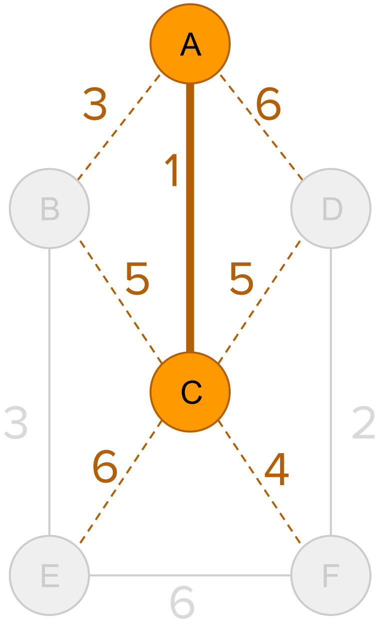
Step 2: Find shortest edge leaving the tree and add it to the tree

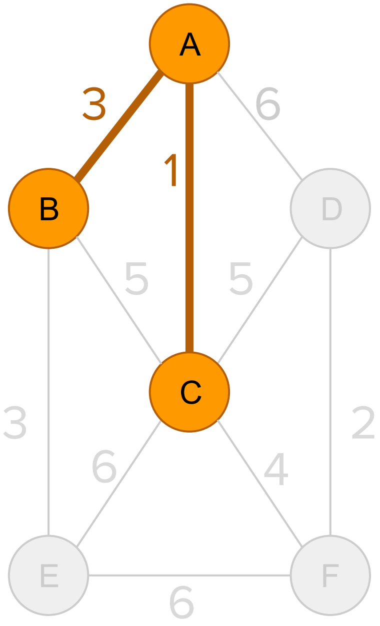




Prim’s algorithm

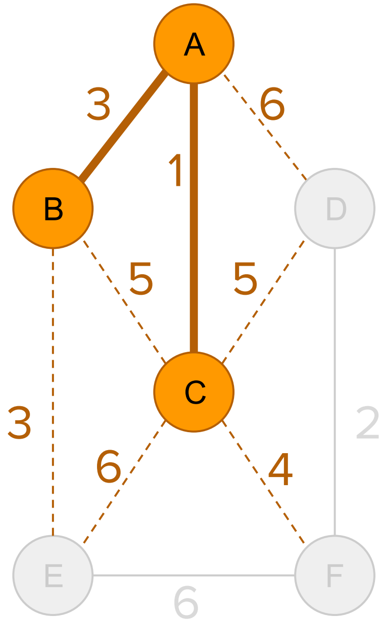
Step 2: Find shortest edge leaving the tree and add it to the tree

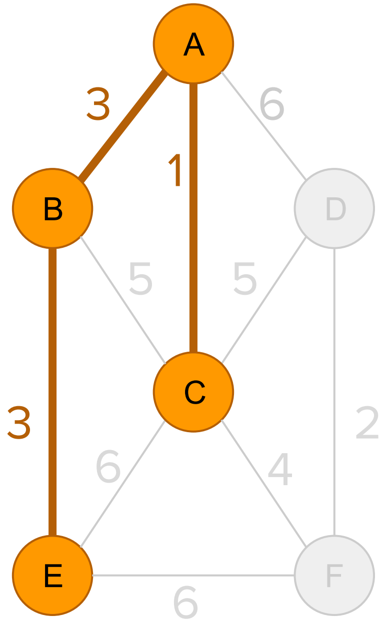




Prim’s algorithm

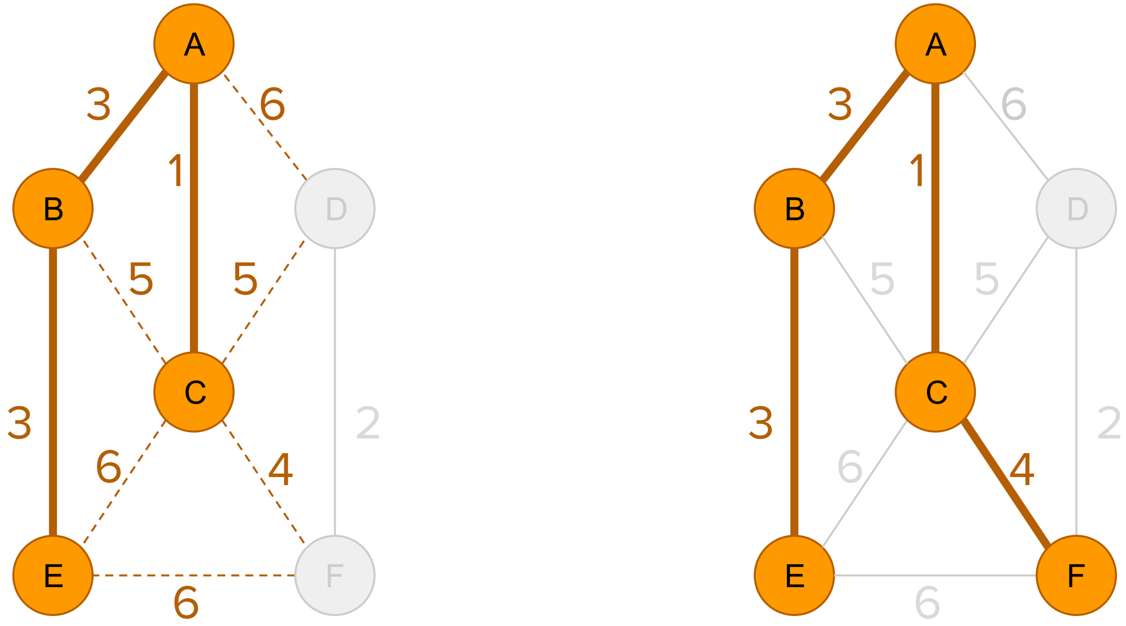
Step 2: Find shortest edge leaving the tree and add it to the tree





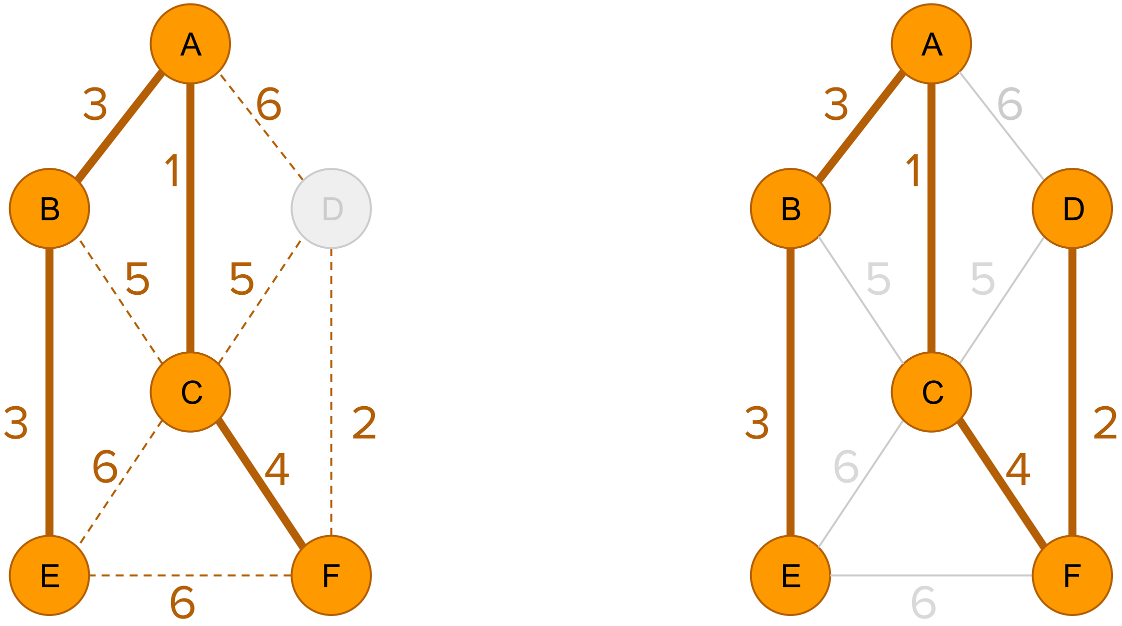
Prim’s algorithm

Step 2: Find shortest edge leaving the tree and add it to the tree



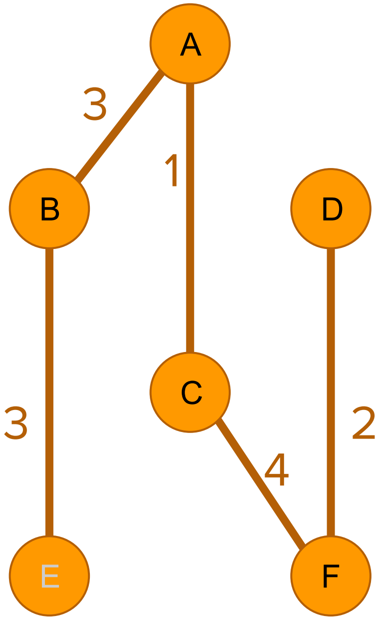
Prim’s algorithm

Step 2: Find shortest edge leaving the tree and add it to the tree



Prim’s algorithm

So the spanning tree is

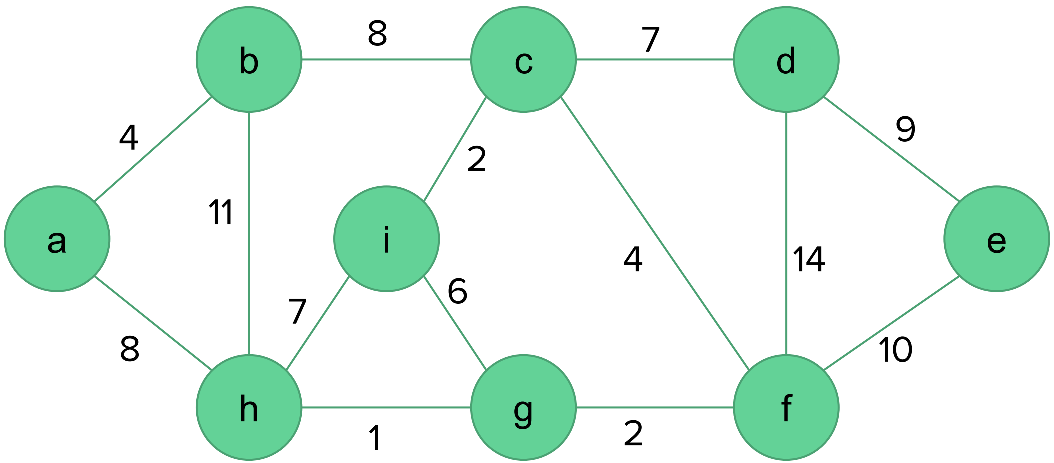


Prim's algorithm

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

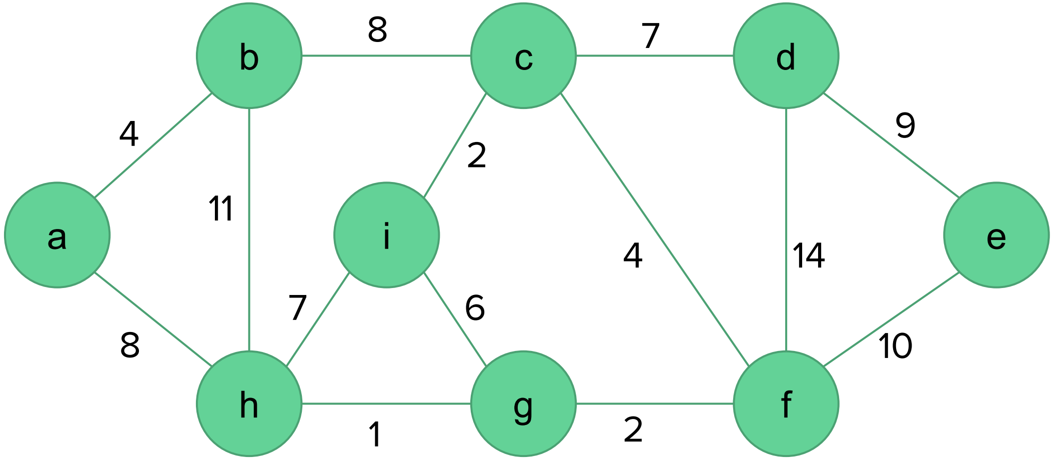
Example

Find a minimum spanning tree of the following graph



Example

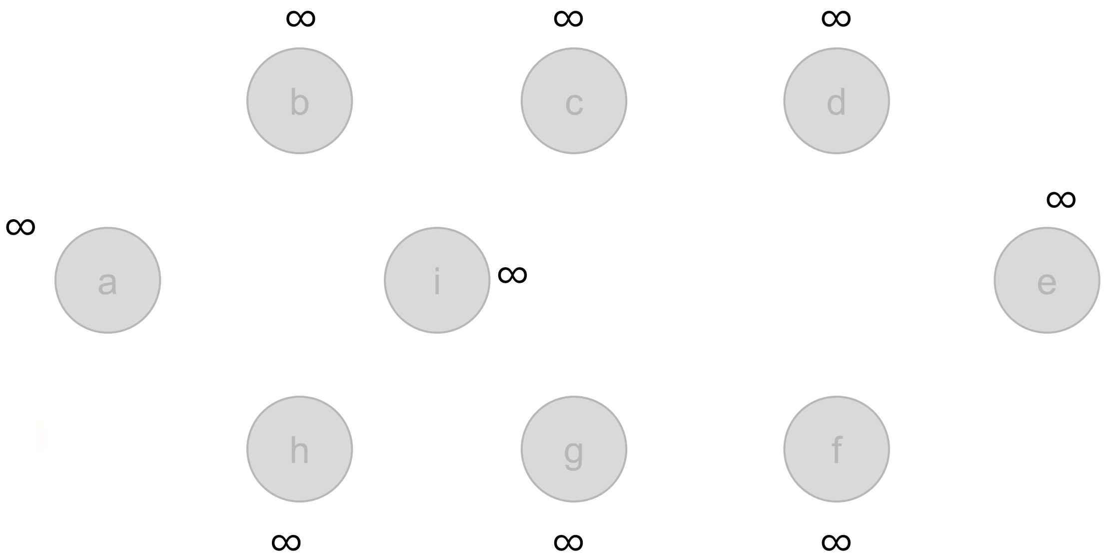
```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)   6
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



Example

MST-PRIM(G, w, r)

```
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
```

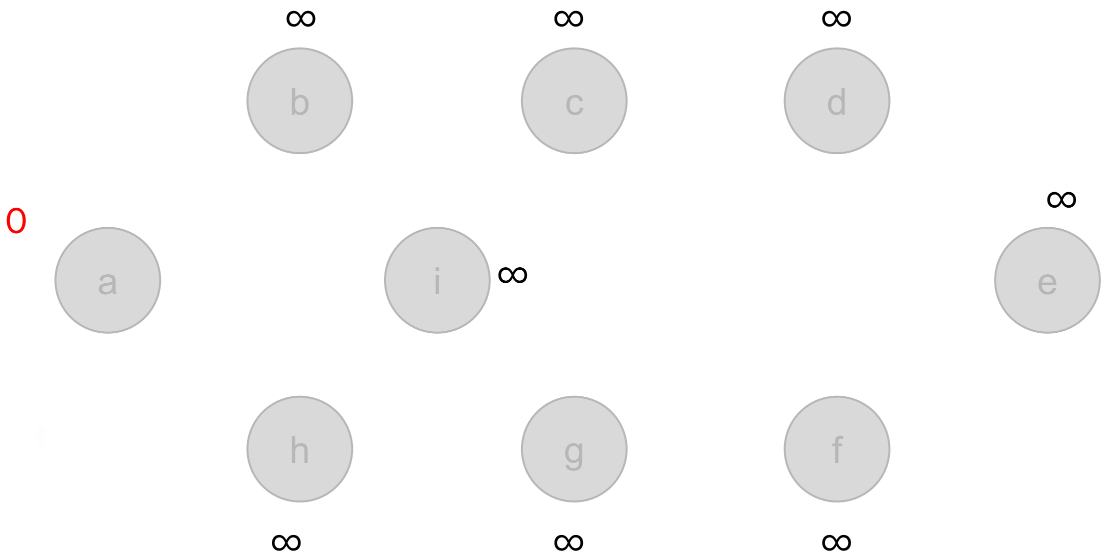


Example

MST-PRIM(G, w, r)

4 $r.key = 0$

5 $Q = G.V$ Q = {a, b, c, d, e, f, g, h, i}

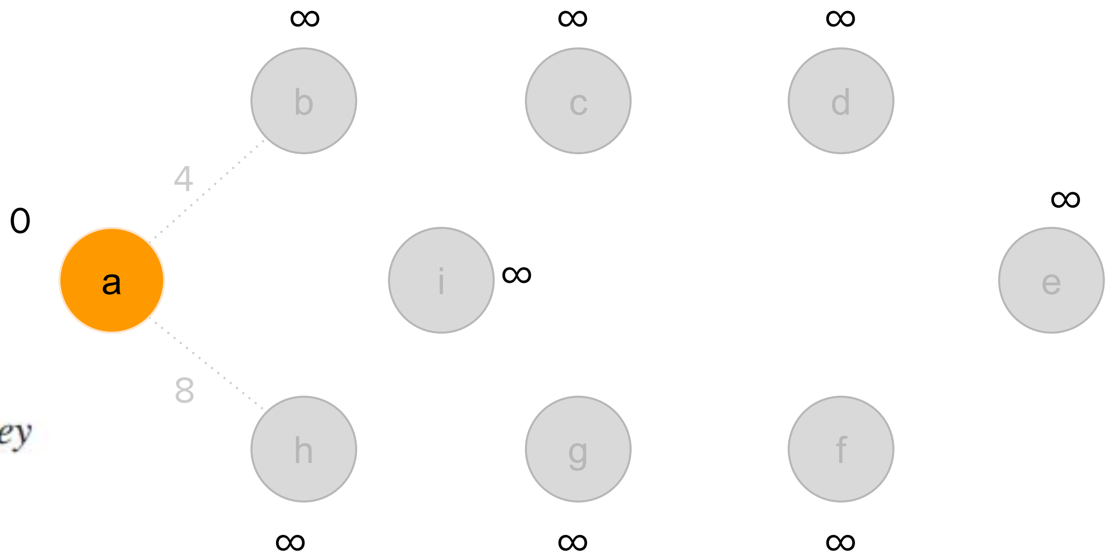


Example

MST-PRIM(G, w, r)

Q = {b, c, d, e, f, g, h, i}

```
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



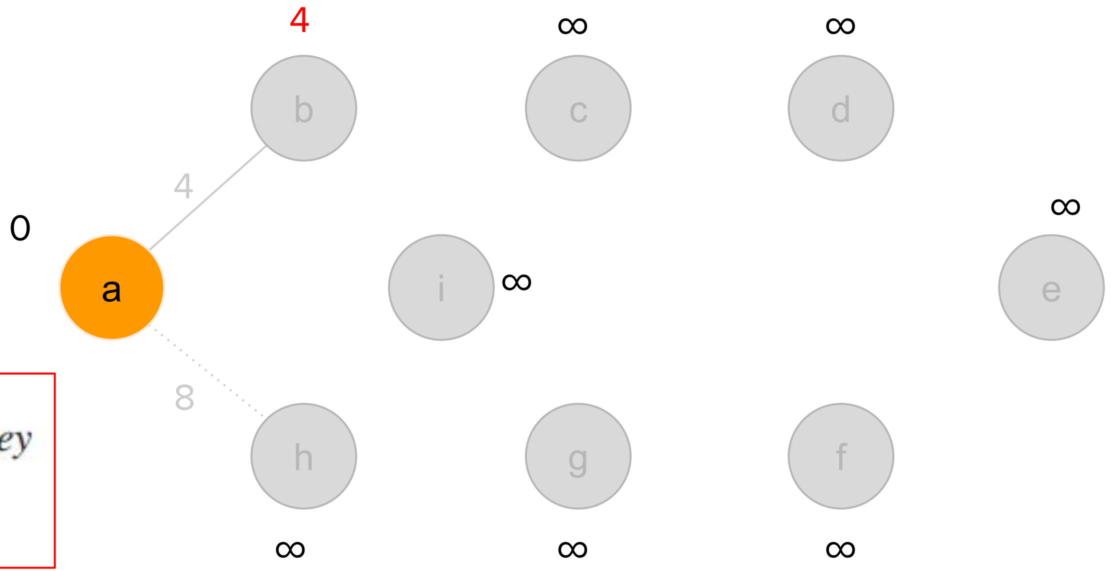
u = a
G.Adj[u] = {b, h}

Example

MST-PRIM(G, w, r)

$Q = \{b, c, d, e, f, g, h, i\}$

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



G.Adj[u] = {b, h}

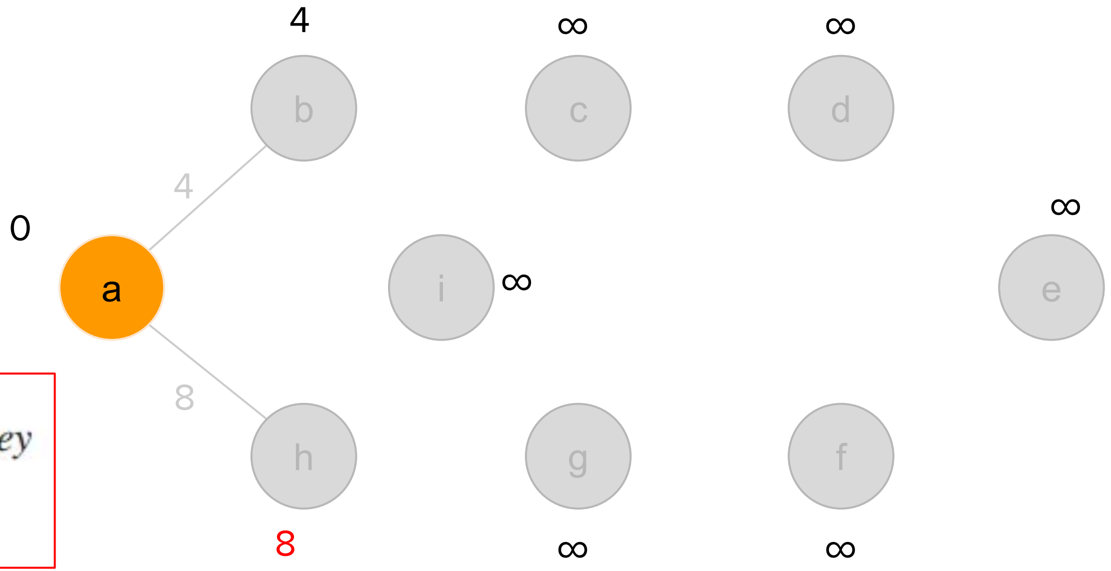
b ∈ Q and w(u,v) < b.key ? True

Example

MST-PRIM(G, w, r)

Q = {b, h, c, d, e, f, g, i}

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



G.Adj[u] = {b, h}

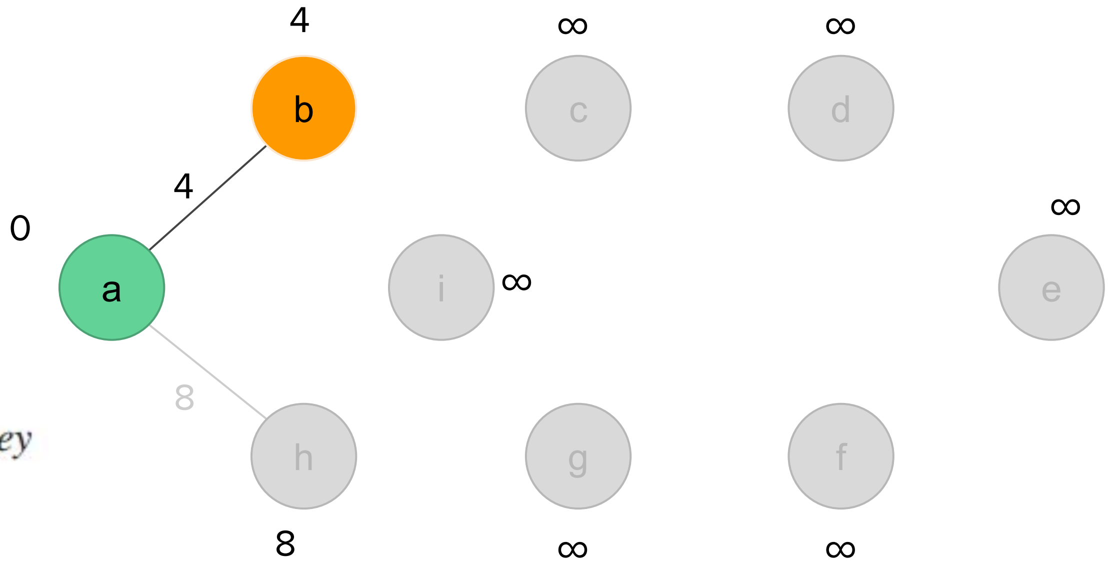
h ∈ Q and w(u,v) < h.key ? True

Example

MST-PRIM(G, w, r)

Q = {h, c, d, e, f, g, i}

```
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



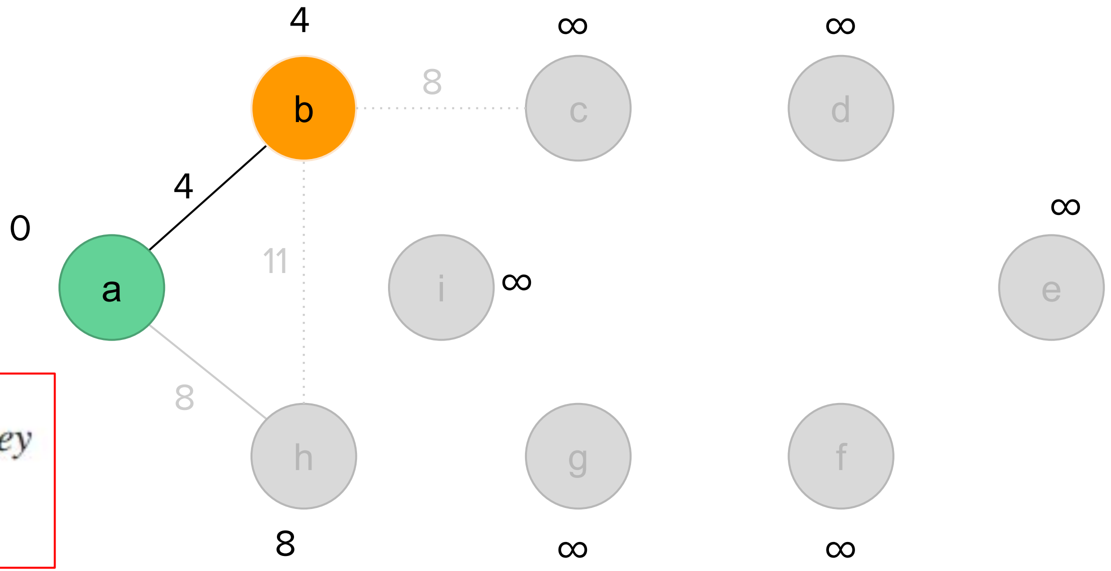
u = b

Example

MST-PRIM(G, w, r)

$Q = \{h, c, d, e, f, g, i\}$

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



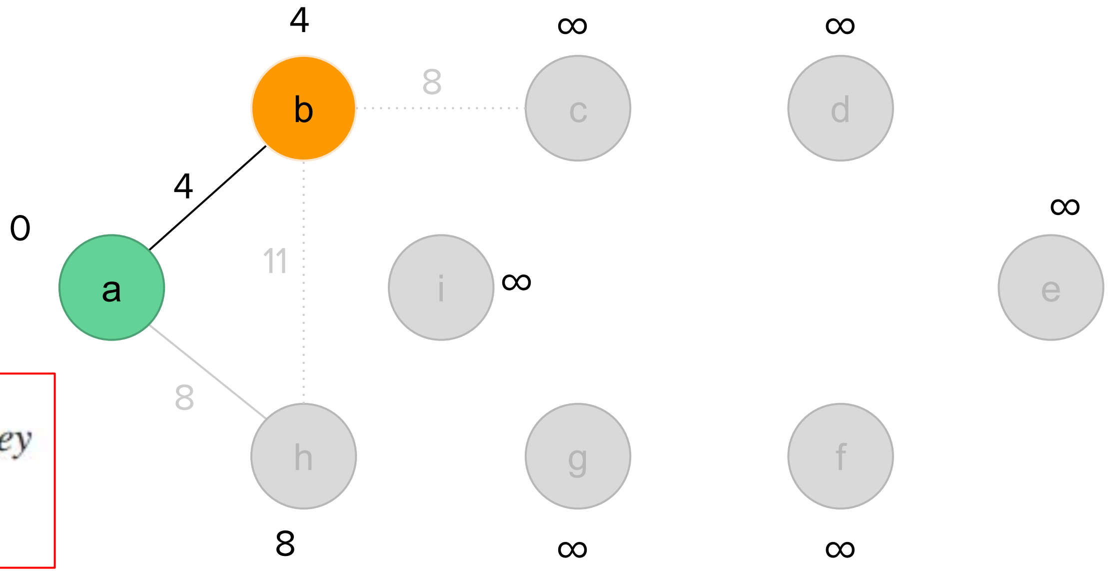
G.Adj[u] = {a, c, h}

Example

MST-PRIM(G, w, r)

$Q = \{h, c, d, e, f, g, i\}$

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



G.Adj[u] = {a, c, h}

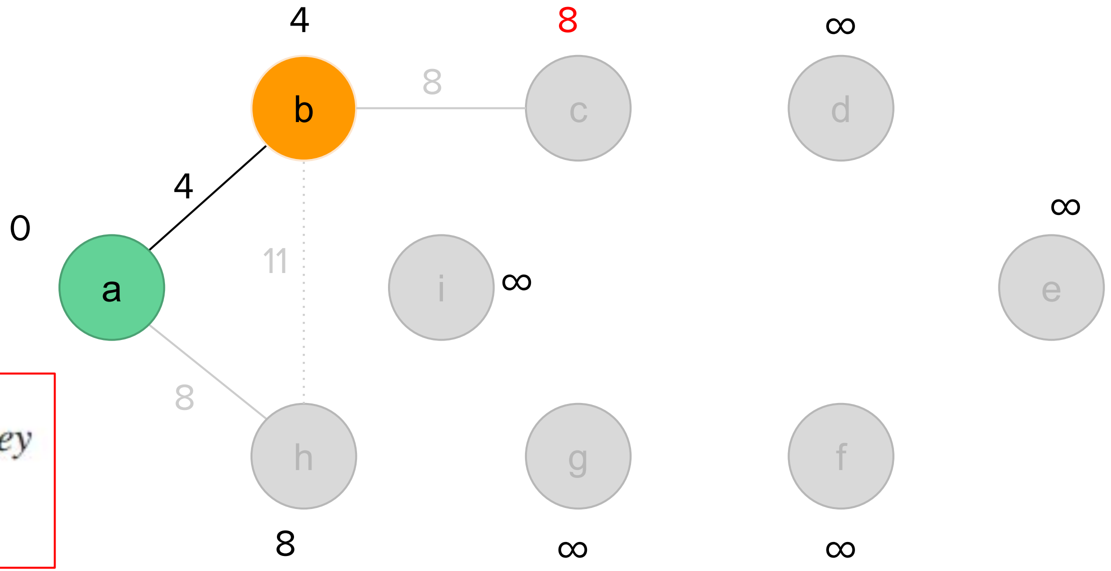
a ∈ Q and w(u,v) < a.key ? False

Example

MST-PRIM(G, w, r)

Q = {h, c, d, e, f, g, i}

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



G.Adj[u] = {a, c, h}

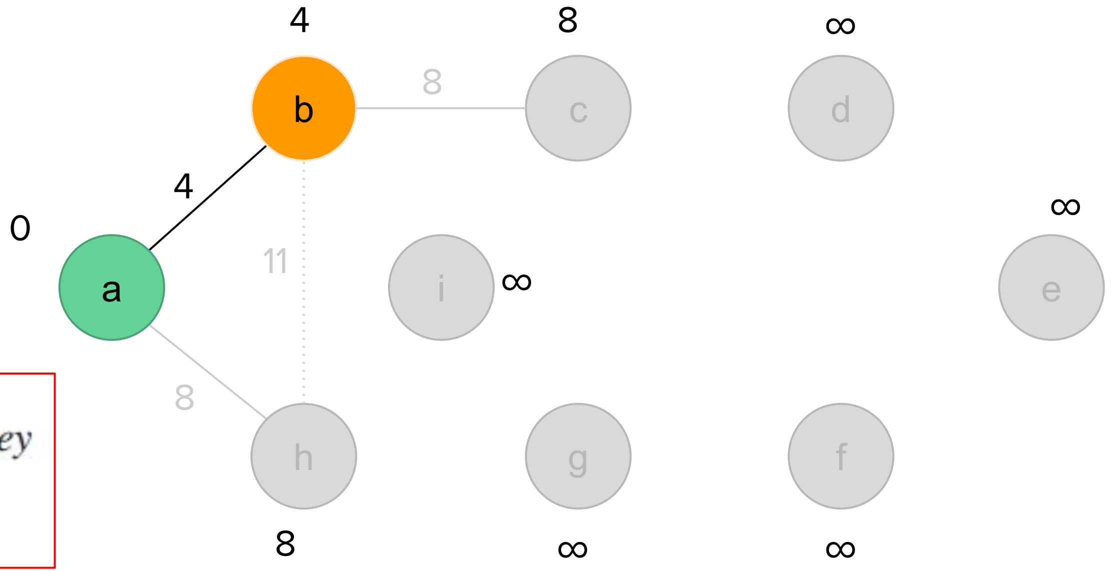
c ∈ Q and w(u,v) < c.key ? True

Example

MST-PRIM(G, w, r)

Q = {h, c, d, e, f, g, i}

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



G.Adj[u] = {a, c, h}

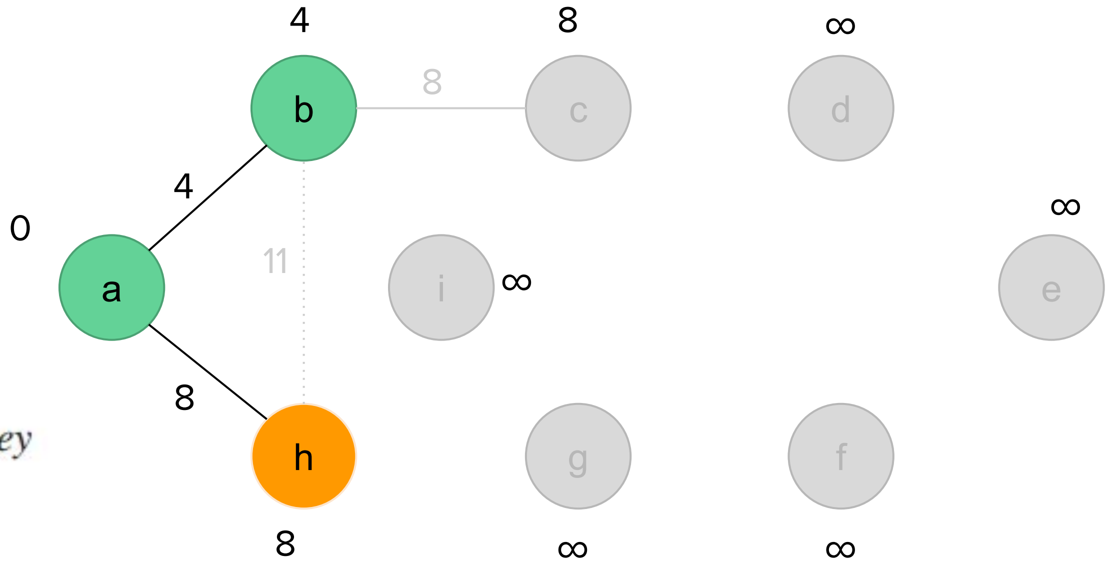
h ∈ Q and w(u,v) < h.key ? False

Example

MST-PRIM(G, w, r)

Q = {c, d, e, f, g, i}

while $Q \neq \emptyset$
    $\boxed{u = \text{EXTRACT-MIN}(Q)}$
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            $v.key = w(u, v)$



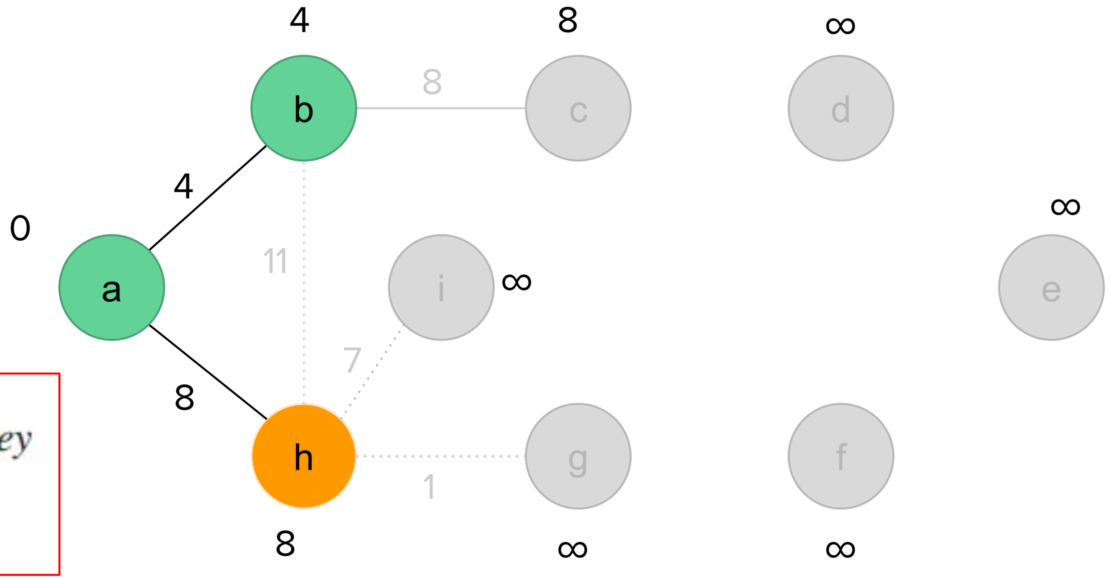
u = h

Example

MST-PRIM(G, w, r)

Q = {c, d, e, f, g, i}

```
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



u = h

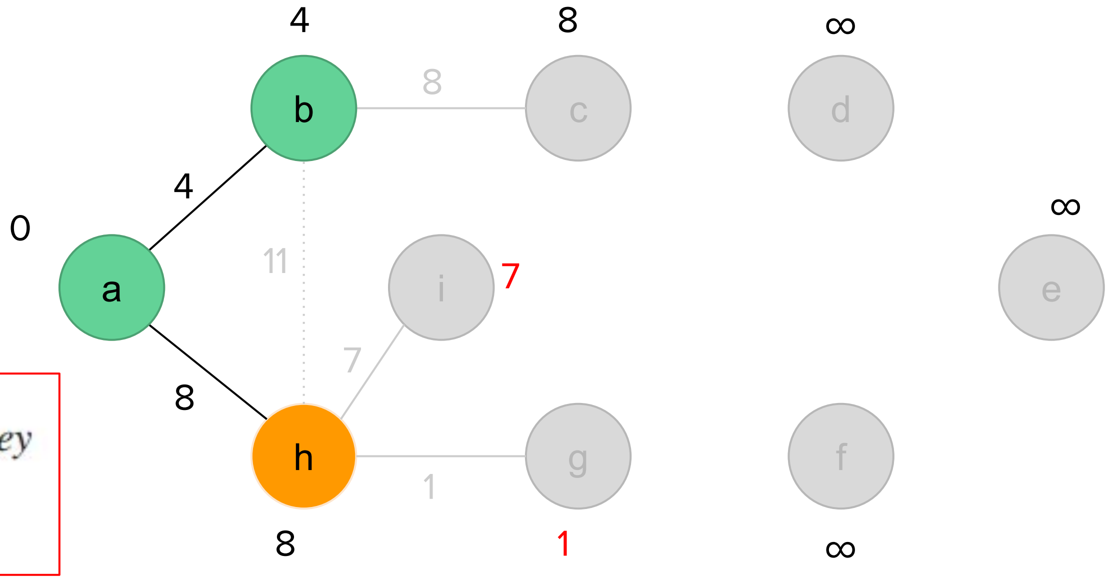
G.Adj[u] = {a, b, g, i}

Example

```
MST-PRIM(G, w, r)

    Q = {g, i, c, d, e, f}

while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



u = h

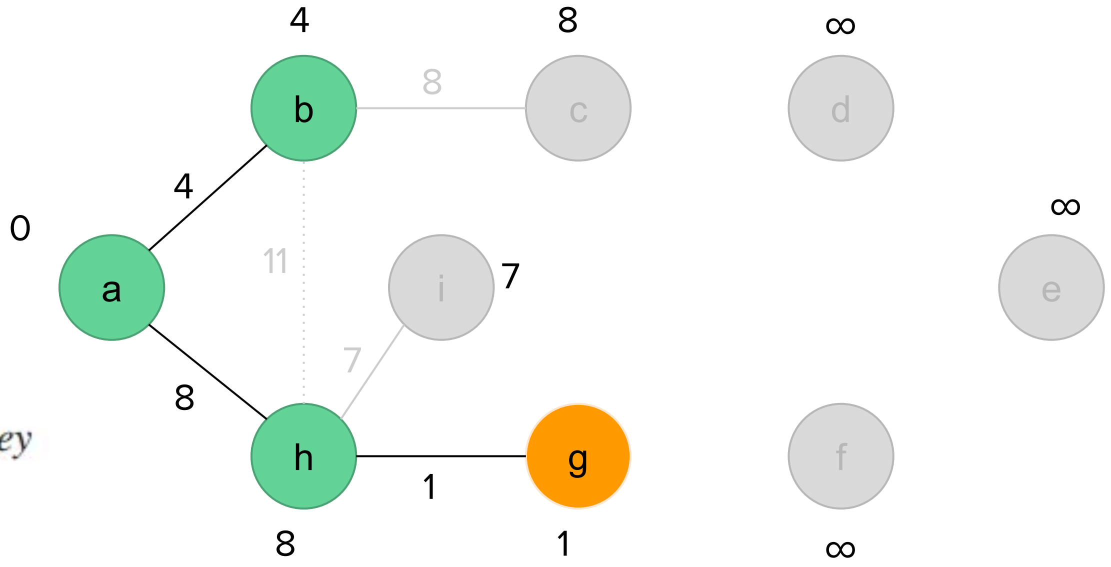
G.Adj[u] = {a, b, g, i}

Example

MST-PRIM(G, w, r)

Q = {i, c, d, e, f}

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



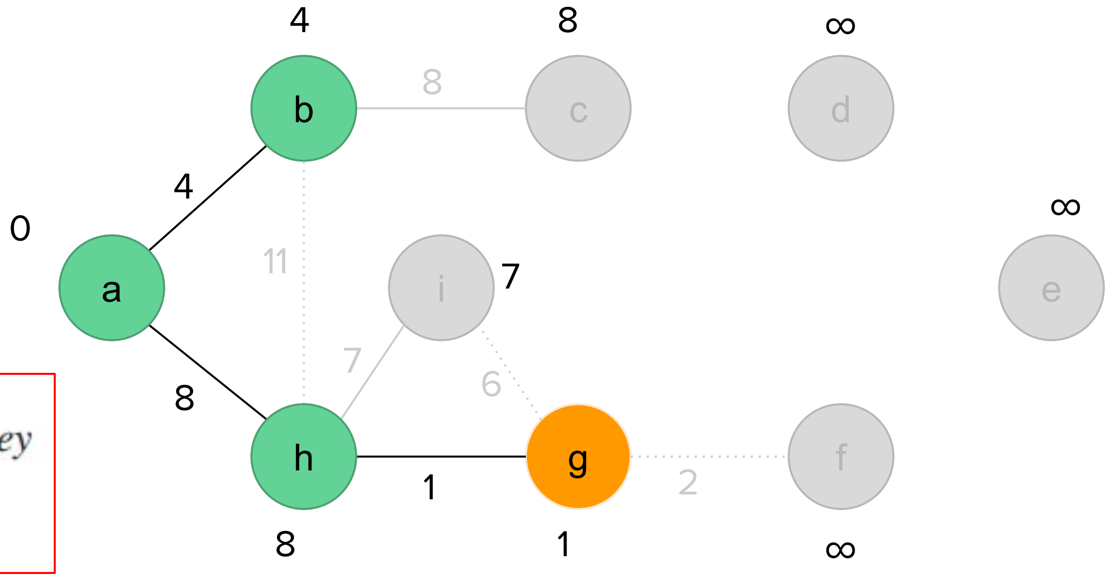
u = g

Example

MST-PRIM(G, w, r)

$Q = \{c, d, e, f, i\}$

```
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



u = g

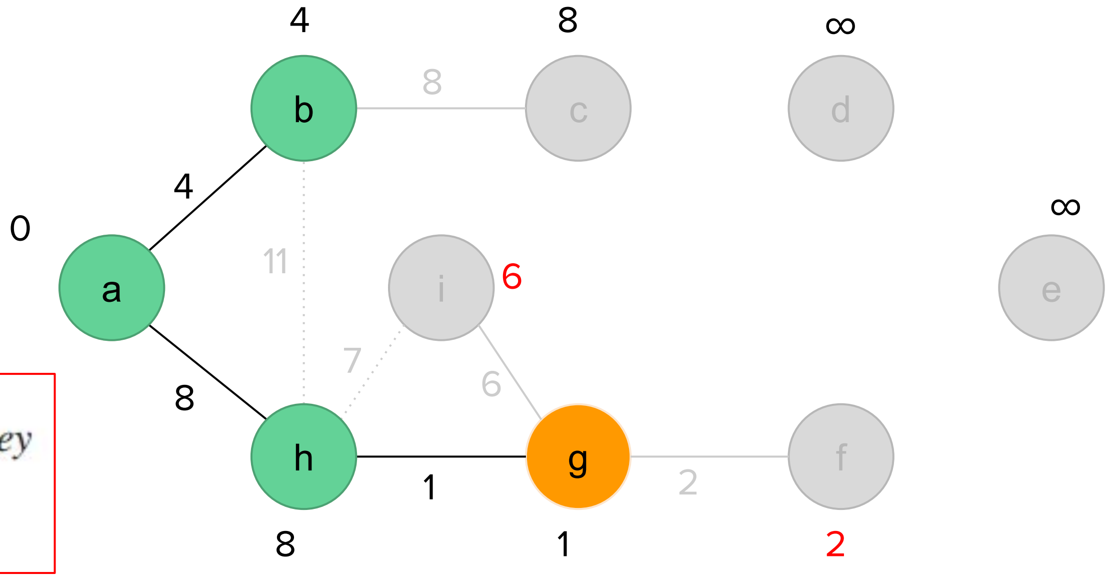
G.Adj[u] = {f, h, i}

Example

MST-PRIM(G, w, r)

Q = {f, i, c, d, e}

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



u = g

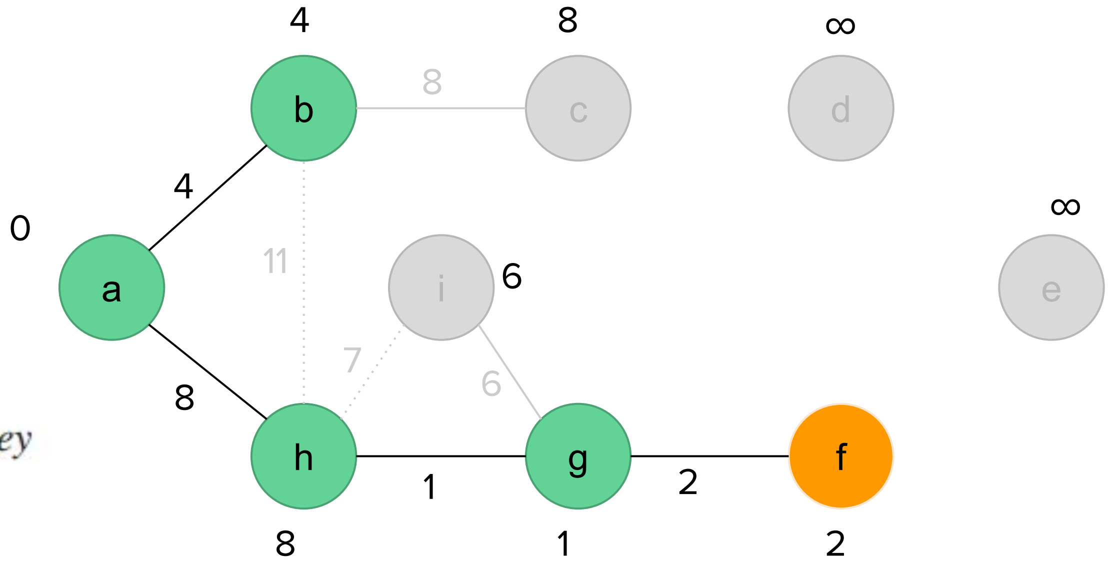
G.Adj[u] = {f, h, i}

Example

MST-PRIM(G, w, r)

$Q = \{i, c, d, e\}$

```
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



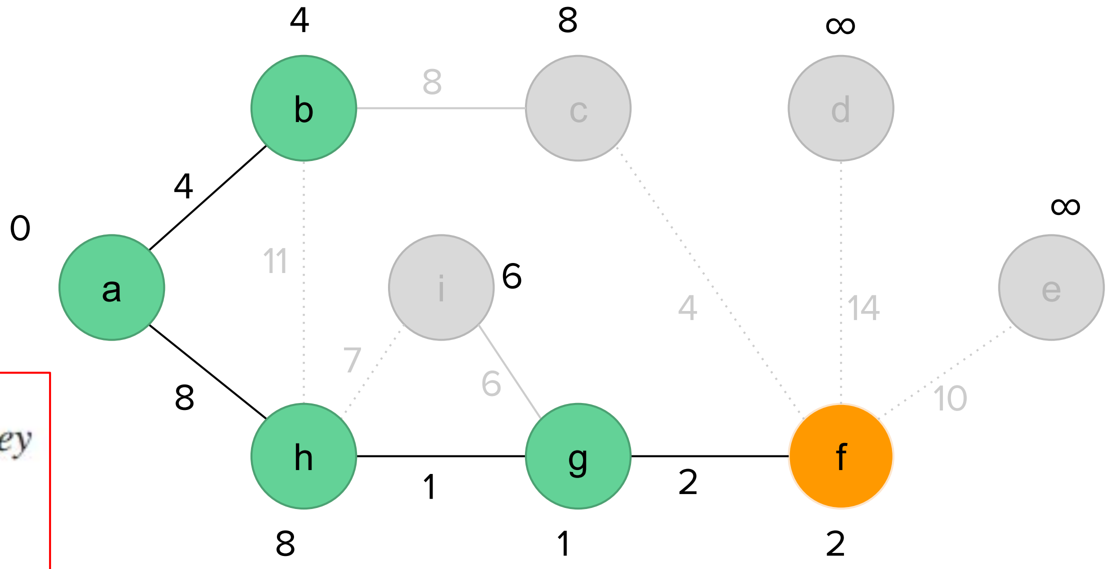
u = f

Example

MST-PRIM(G, w, r)

Q = {i, c, d, e}

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



u = f

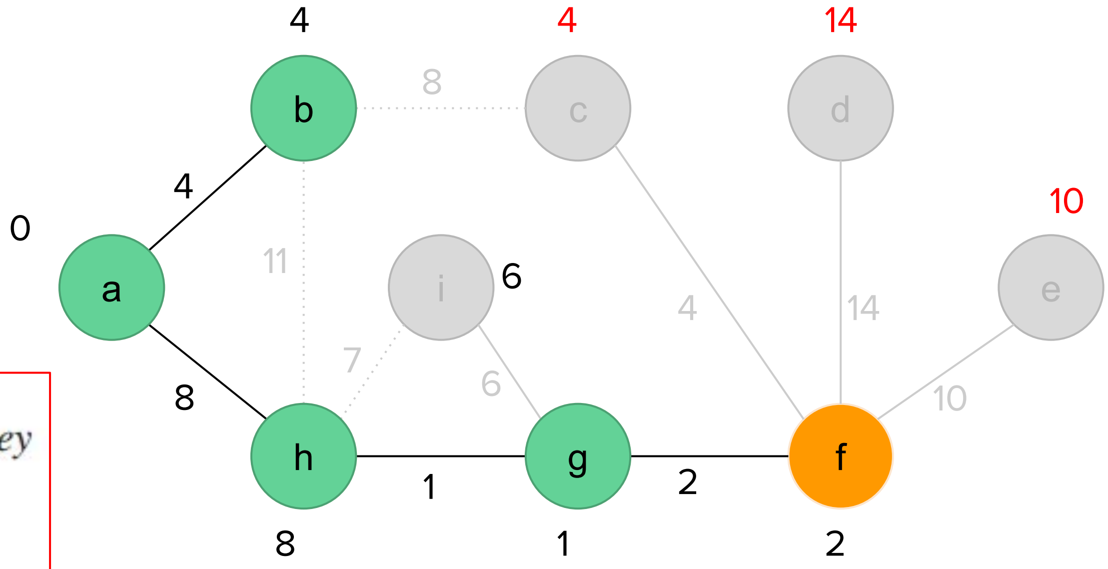
G.Adj[u] = {c, d, e, g}

Example

MST-PRIM(G, w, r)

Q = {c, i, e, d}

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



u = f

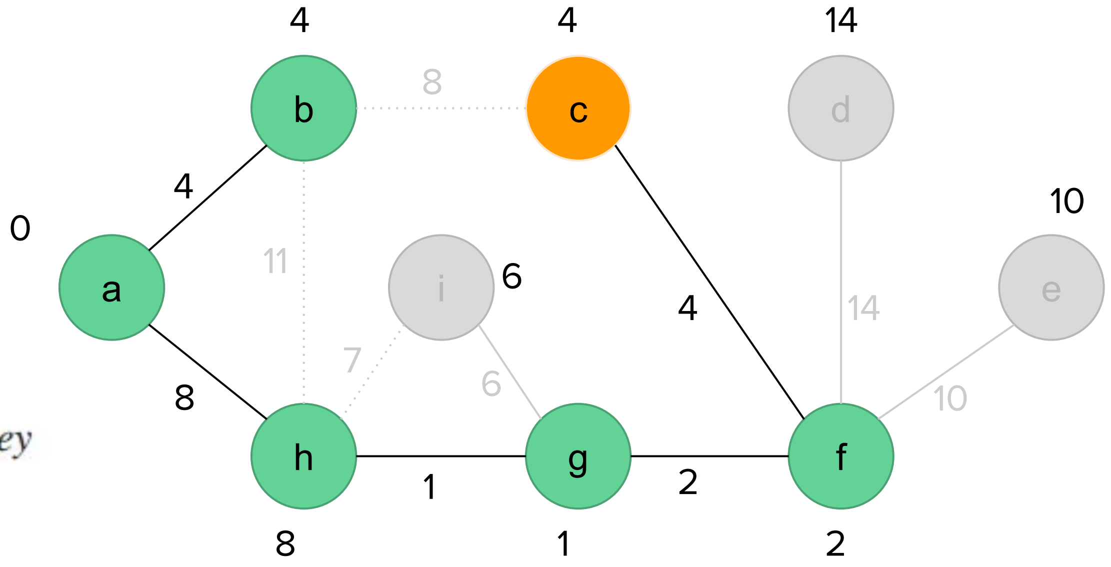
G.Adj[u] = {c, d, e, g}

Example

MST-PRIM(G, w, r)

Q = {i, e, d}

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



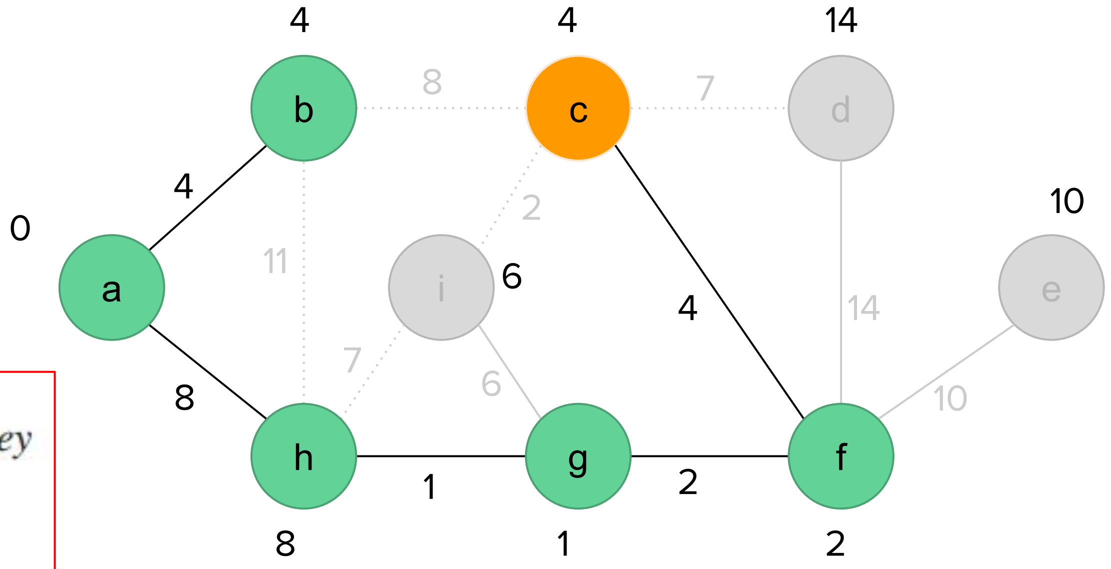
u = c

Example

MST-PRIM(G, w, r)

$Q = \{i, e, d\}$

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



u = c

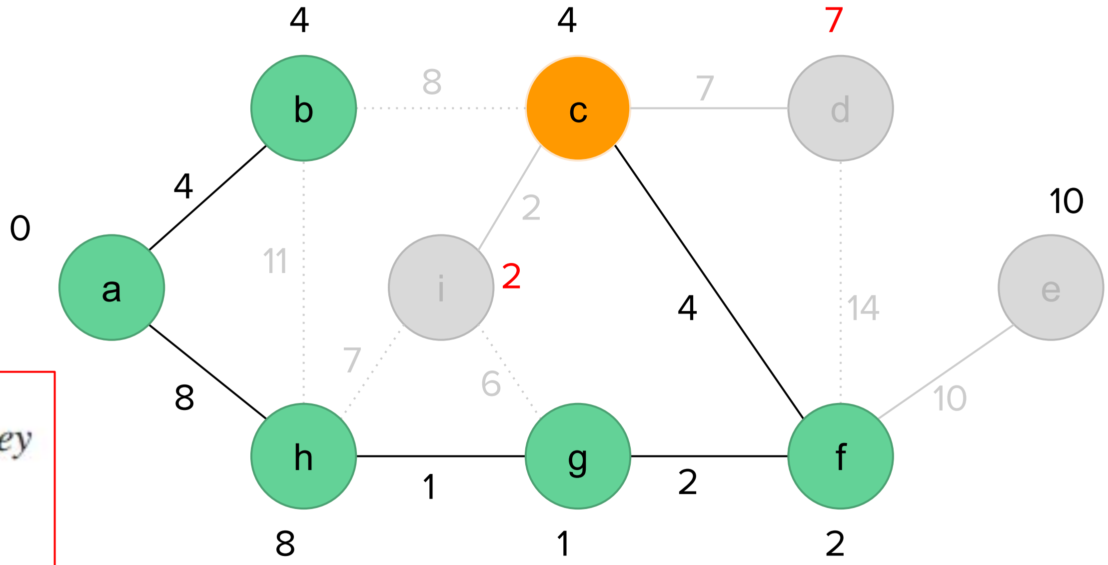
G.Adj[u] = {b, d, f, i}

Example

MST-PRIM(G, w, r)

Q = {i, d, e}

6 **while** $Q \neq \emptyset$
7 $\quad u = \text{EXTRACT-MIN}(Q)$
8 $\quad$ **for** each $v \in G.Adj[u]$
9 $\quad\quad$ **if** $v \in Q$ and $w(u, v) < v.key$
10 $\quad\quad\quad v.\pi = u$
11 $\quad\quad\quad v.key = w(u, v)$



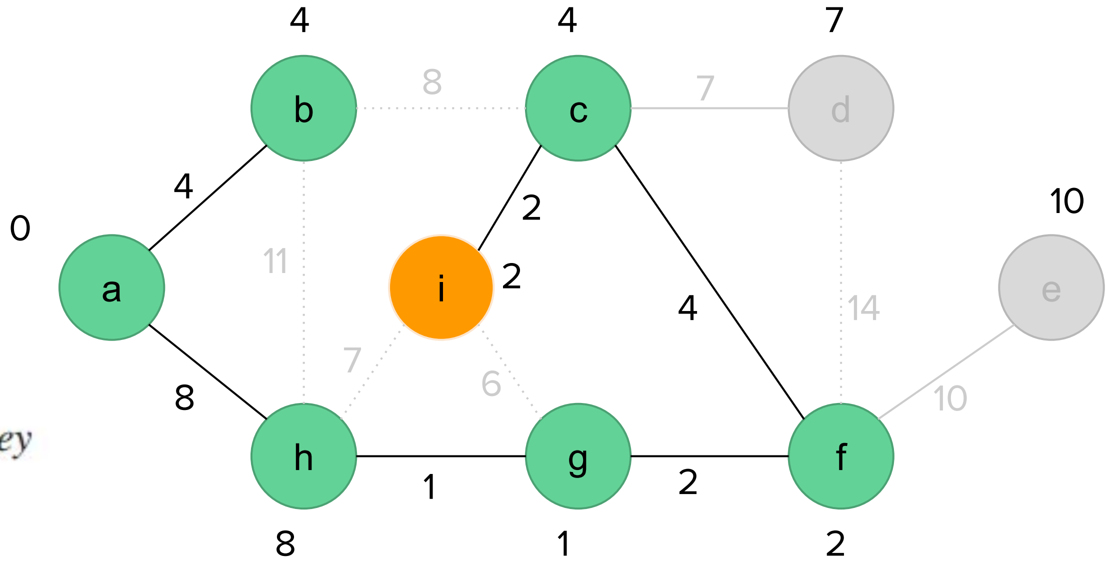
u = c
G.Adj[u] = {b, d, f, i}

Example

MST-PRIM(G, w, r)

$Q = \{d, e\}$

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



u = i

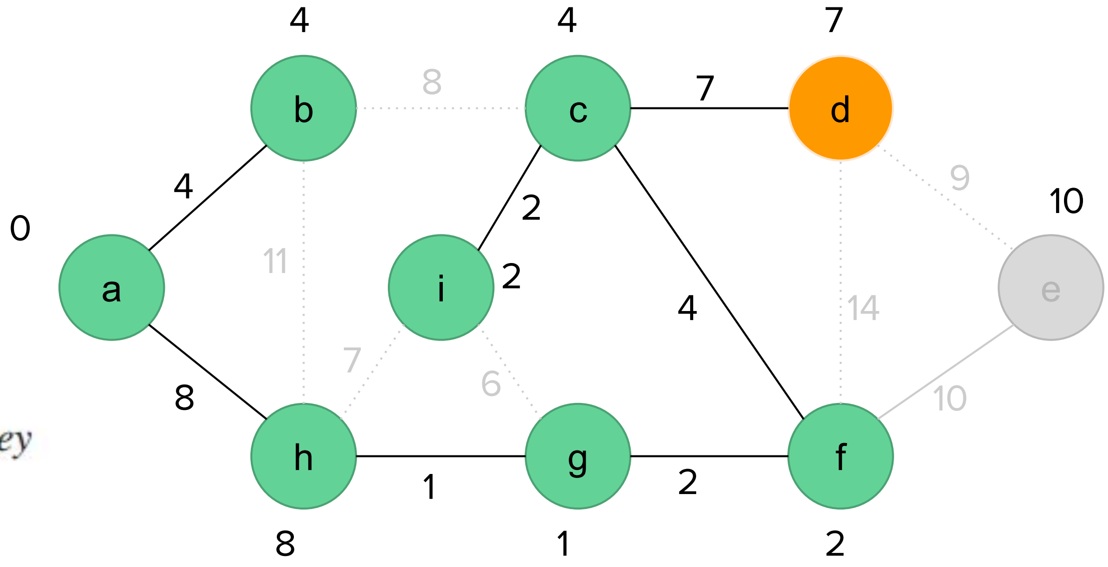
G.Adj[u] = {b, c, g, h}

Example

MST-PRIM(G, w, r)

Q = {e}

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



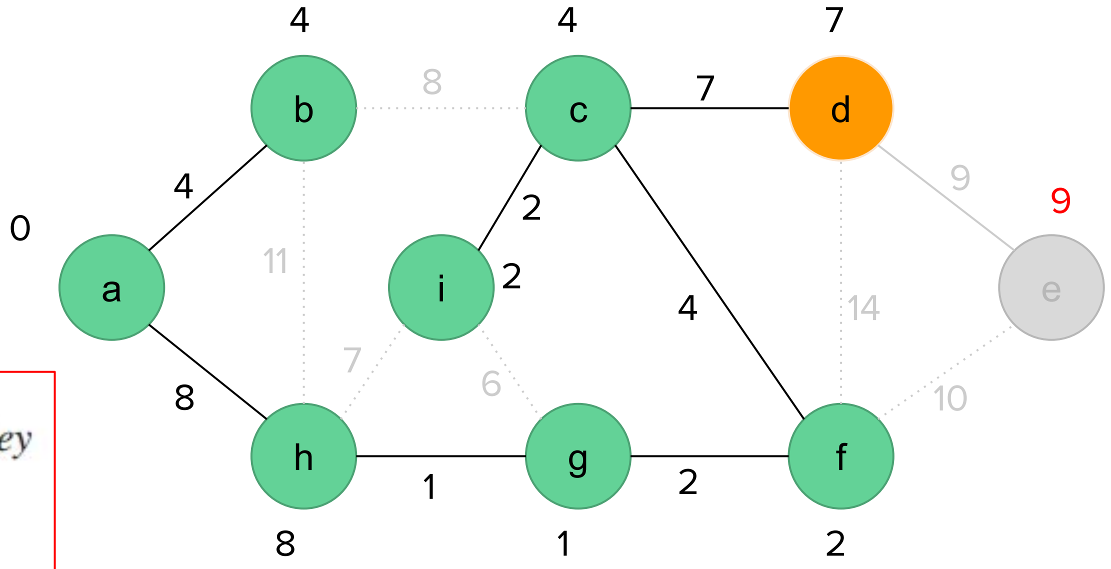
u = d

Example

MST-PRIM(G, w, r)

$Q = \{e\}$

```
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```



u = d

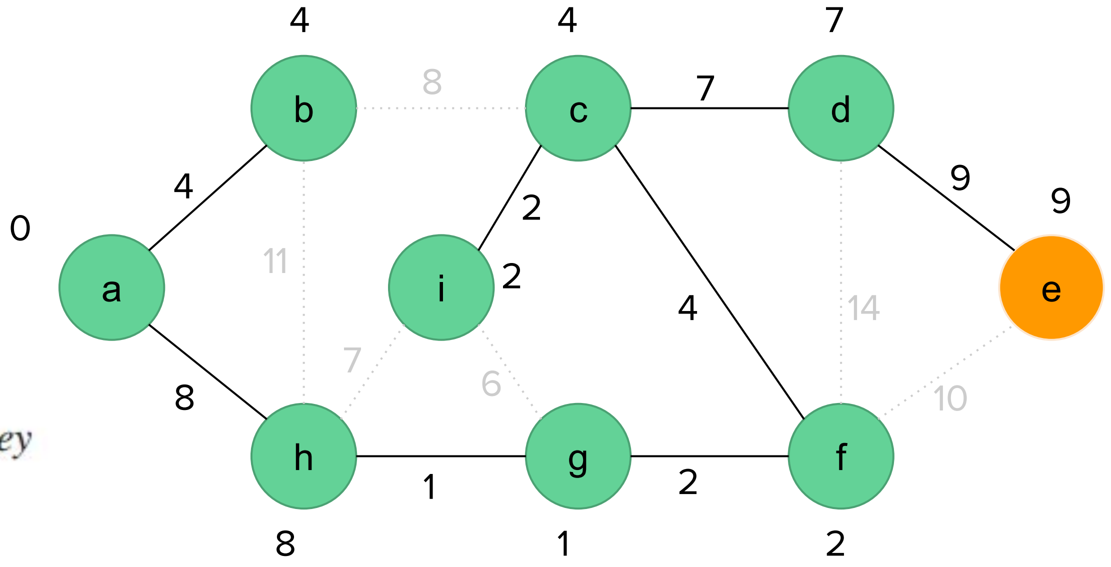
G.Adj[u] = {c, e, f}

Example

MST-PRIM(G, w, r)

Q = {}

```
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

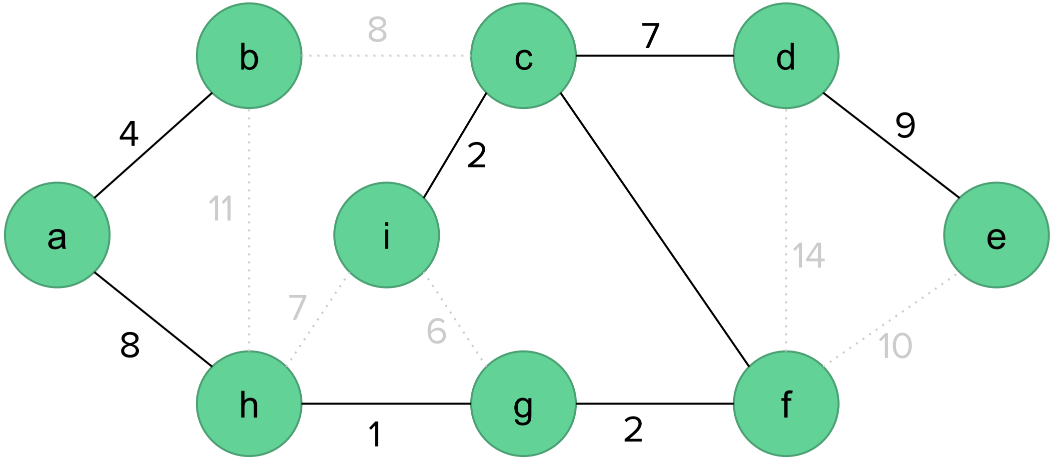


u = e

G.Adj[u] = {d, f}

Example

```
MST-PRIM(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.π = NIL
    r.key = 0
    Q = G.V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each v ∈ G.Adj[u]
            if v ∈ Q and w(u, v) < v.key
                v.π = u
                v.key = w(u, v)
```



Minimum spanning tree

Analysis of Prim's algorithm

```
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)
```

Line 1 - 3: O(|V|)

Line 4: O(1)

Line 5-11: ?

Analysis of Prim's algorithm

```
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)
```

Line 1 - 3: O(|V|)

Line 4: O(1)

Line 5-11: Depends on how we implement the min-priority queue Q

Analysis of Prim's algorithm

- Depends on how we implement the min-priority queue Q
- If we implement Q as a binary min-heap

```
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)
```

Line 1 - 3: O(|V|)

Line 4: O(1)

Line 5: O(|V|)
Line 6: The while loop executes |V| times.

Line 7:
Extract-Min takes O(lg V).
Therefore total time for all calls to Extract-Min is O(V lg V)

Analysis of Prim’s algorithm

- Depends on how we implement the min-priority queue Q
- If we implement Q as a binary min-heap

```
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)
```

Line 1 - 3: O(|V|)

Line 4: O(1)

Line 5: O(|V|)
Line 6: The while loop executes |V| times.

Line 7:
Extract-Min takes O(lg V).
Therefore total time for all calls to Extract-Min is O(V lg V)

Line 8: The for loop executes O(E) since the sum of the lengths of all adjacency lists is 2 |E|

Line 11: It decreases the key of the node in the min-heap, so the heap needs to be adjusted, which a binary min-heap supports in O(lg V) time.

Analysis of Prim's algorithm

- Depends on how we implement the min-priority queue Q
- If we implement Q as a binary min-heap

```
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)
```

Line 1 - 3: O(|V|)

Line 4: O(1)

Line 5: O(|V|)
Line 6: O(|V|).

Line 7: O(V lg V)

Line 8: O(E)

Line 11: O(E lg V)

Total time for Prim's algo = O(V lg V + E lg V) = O(E lg V)

Analysis of Prim's algorithm

- Depends on how we implement the min-priority queue Q
- If we implement Q as a **Fibonacci heap**

```
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)
```

Line 1 - 3: O(|V|)

Line 4-5: O(1)

Line 6: O(|V|).

Line 7: O(V log V)

Line 8: O(E)

Line 11: O(E)

Total time for Prim's algo = O(V lg V + E)

Shortest path algorithms

Shortest path algorithm

Finds the shortest path between two vertices in a graph

Applications:

- Finding the shortest path from one location to another in Google Maps, MapQuest, OpenStreetMap, (KTM Public Route) etc.
- Used by Telephone networks, Cellular networks for routing/connection in communication
- IP routing
- Word ladder problem

Single-source shortest path problem

The problem of finding shortest paths from a source vertex v to all other vertices in the graph.

Optimal substructure of a shortest path

Shortest-path algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.

Dijkstra's algorithm

A solution to the single-source shortest path problem in graph theory.

- **Input**: Weighted graph $G = (V, E)$ and source vertex $v \in V$, such that all edge weights are nonnegative
- **Output**: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Dijkstra’s shortest path algorithm

Steps:

1. Insert the first vertex into the tree
2. From every vertex already in the tree, examine the total path length to all adjacent vertices not in the tree. Selected the edge with the minimum total path weight and insert it into the tree
3. Repeat step 2 until all vertices are in the tree

Dijkstra’s shortest path algorithm

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$ # Q is a min-priority queue

while Q not empty **do**

  $v \leftarrow Q.pop()$

  for all neighbours u of v **do**

    if $d(v) + e(v, u) \leq d(u)$ **then**

      $d(u) \leftarrow d(v) + e(v, u)$

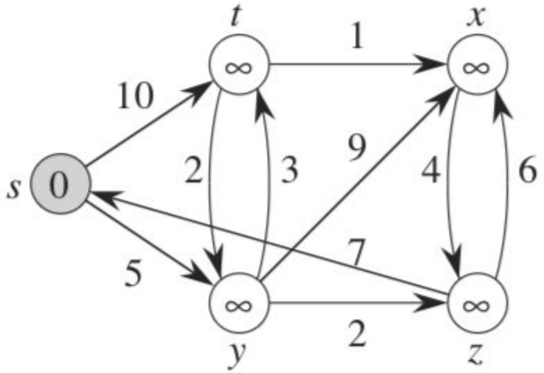
    end if

  end for

end while

Dijkstra’s shortest path algorithm: Example

Find the shortest path from s to all other vertices.

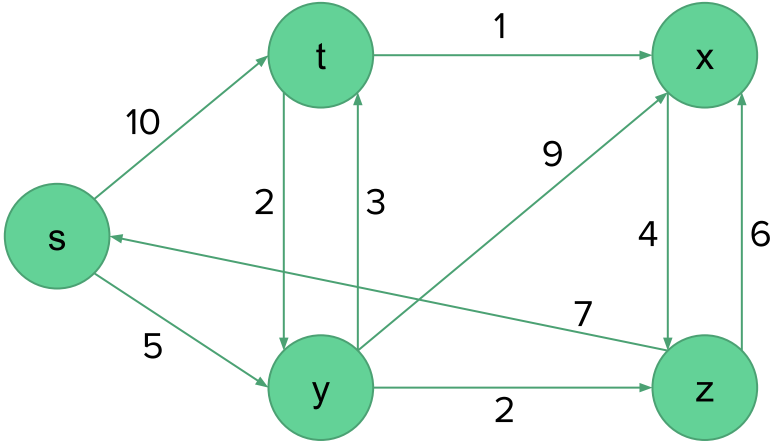


Dijkstra’s shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$
while Q not empty **do**
  $v \leftarrow Q.pop()$
  for all neighbours u of v **do**
    if $d(v) + e(v,u) \leq d(u)$ **then**
      $d(u) \leftarrow d(v) + e(v,u)$
    end if
  end for
end while

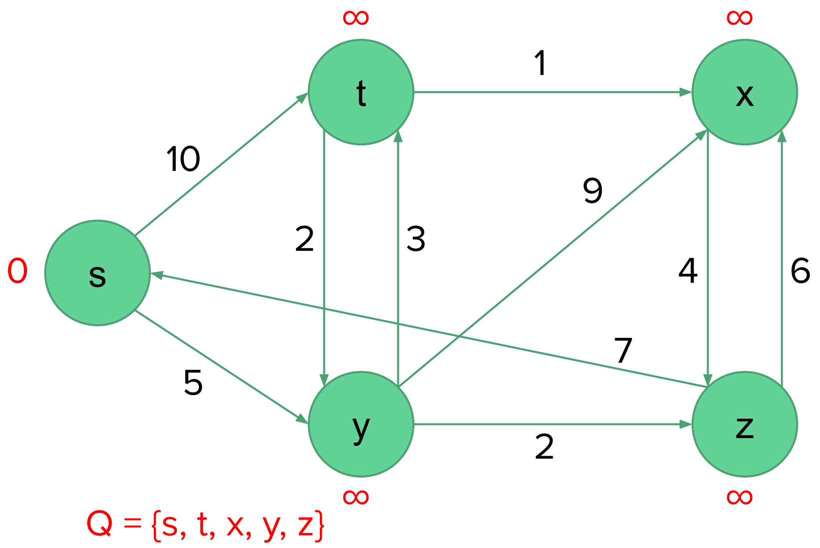


Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$



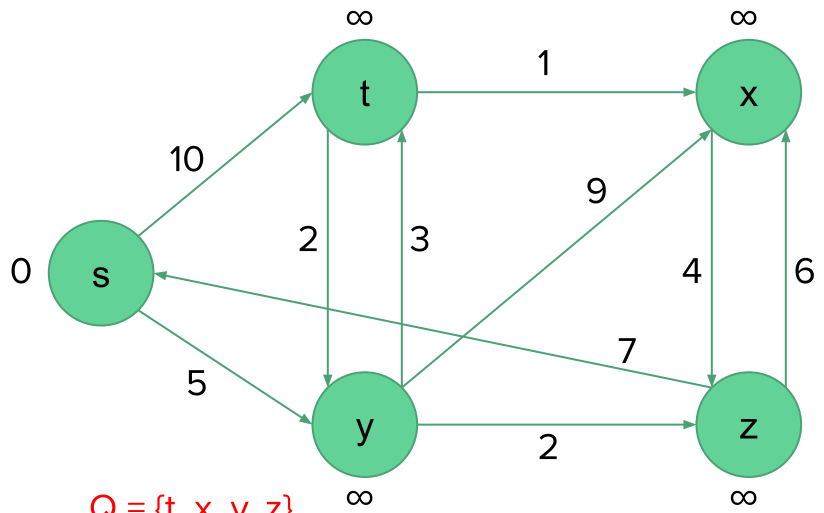
Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**
  $v \leftarrow Q.pop()$
  for all neighbours u of v **do**
    if $d(v) + e(v,u) \leq d(u)$ **then**
      $d(u) \leftarrow d(v) + e(v,u)$
    end if
  end for
end while



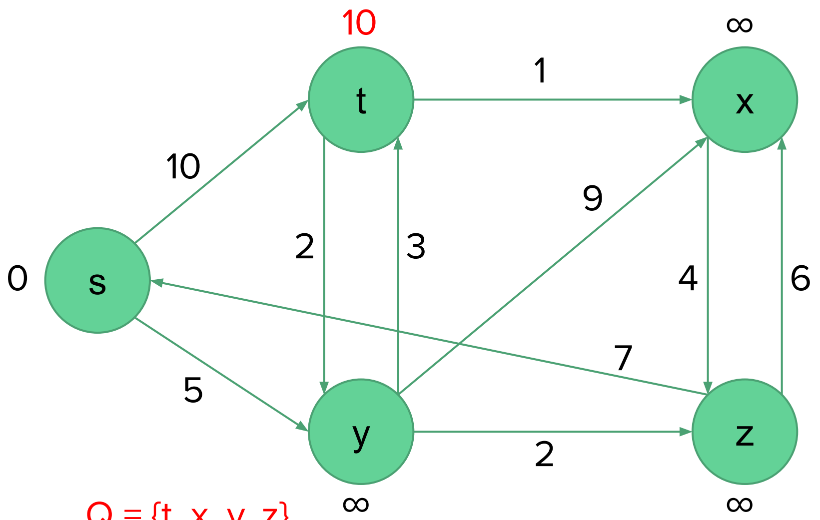
Q = {t, x, y, z}
v = s
Neighbours of v = {t, y}

Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$
while Q not empty **do**
  $v \leftarrow Q.pop()$
  for all neighbours u of v **do**
    if $d(v) + e(v, u) \leq d(u)$ **then**
      $d(u) \leftarrow d(v) + e(v, u)$
    end if
  end for
end while



Q = {t, x, y, z}
v = s
Neighbours of v = {t, y}
d(s) + e(s, t) ≤ d(t) True

Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**

  $v \leftarrow Q.pop()$

  for all neighbours u of v **do**

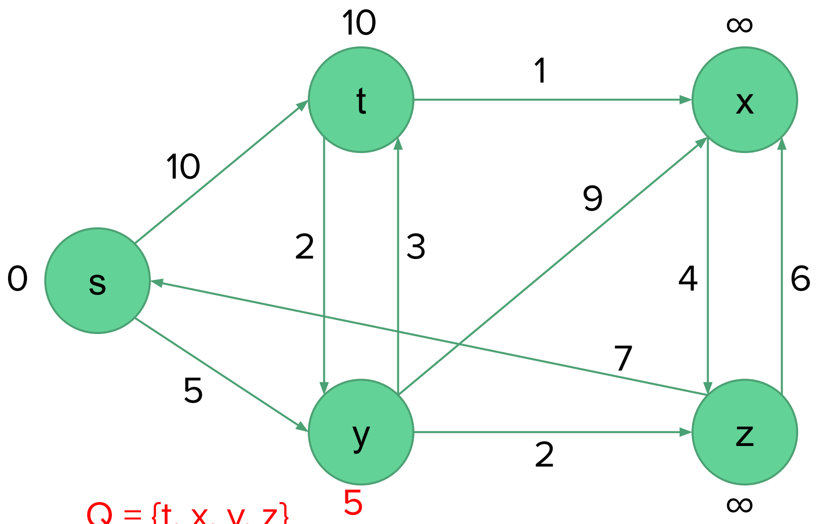
    if $d(v) + e(v, u) \leq d(u)$ **then**

      $d(u) \leftarrow d(v) + e(v, u)$

    end if

  end for

end while



$Q = \{t, x, y, z\}$

$v = s$

Neighbours of $v = \{t, y\}$

$d(s) + e(s, y) \leq d(y)$ True

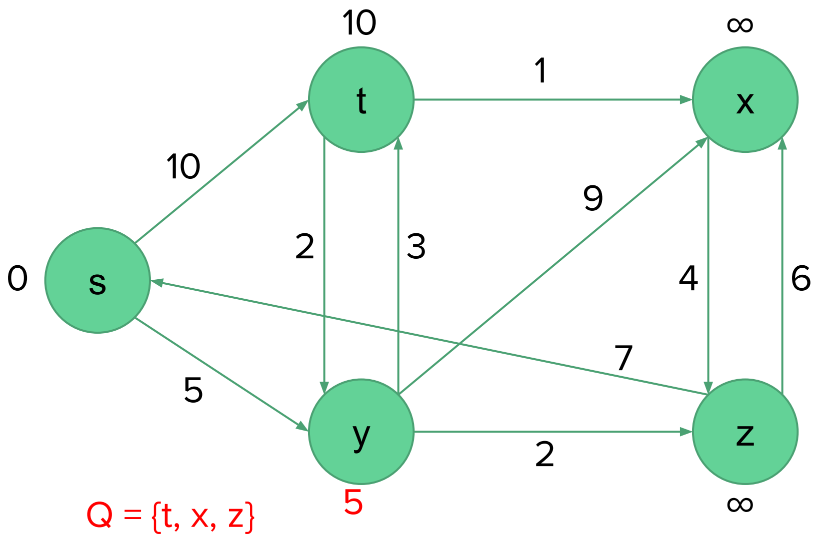
Dijkstra’s shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**
  $v \leftarrow Q.pop()$
  for all neighbours u of v **do**
    if $d(v) + e(v,u) \leq d(u)$ **then**
      $d(u) \leftarrow d(v) + e(v,u)$
    end if
  end for
end while



Q = {t, x, z}
v = y
Neighbours of v = {t, x, z}

Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**

  $v \leftarrow Q.pop()$

  for all neighbours u of v **do**

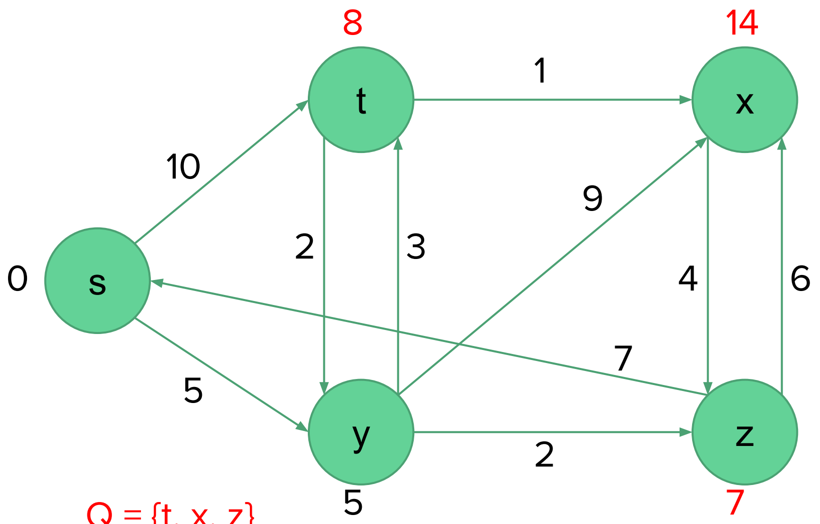
    if $d(v) + e(v,u) \leq d(u)$ **then**

      $d(u) \leftarrow d(v) + e(v,u)$

    end if

  end for

end while



Q = {t, x, z}

v = y

Neighbours of v = {t, x, z}

d(y) + e(y, t) ≤ d(t)   True

d(y) + e(y, x) ≤ d(x)   True

d(y) + e(y, z) ≤ d(z)   True

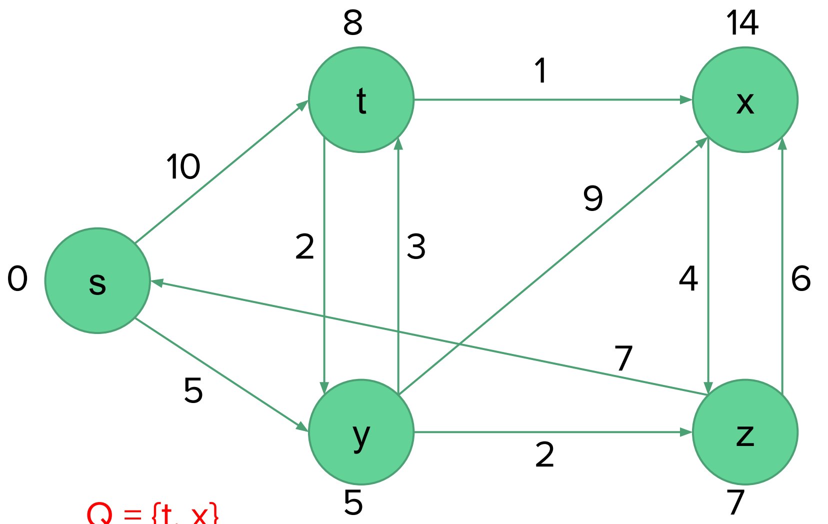
Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**
  $v \leftarrow Q.pop()$
  for all neighbours u of v **do**
    if $d(v) + e(v, u) \leq d(u)$ **then**
      $d(u) \leftarrow d(v) + e(v, u)$
    end if
  end for
end while



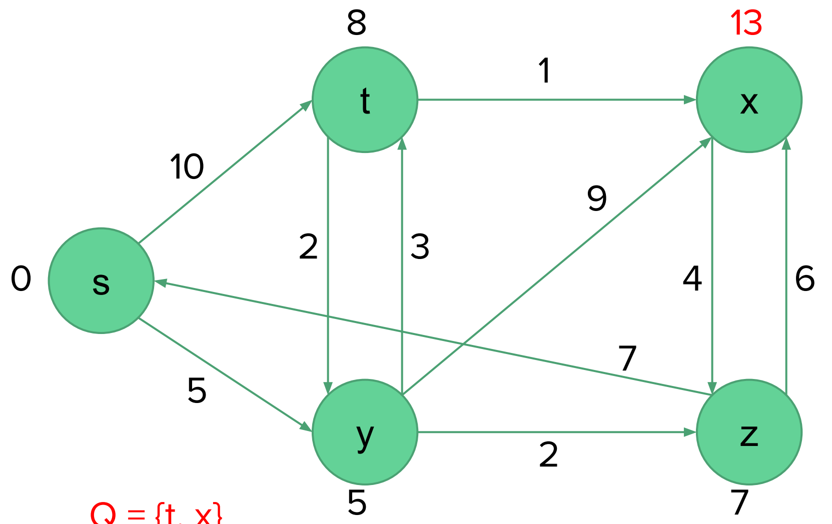
Q = {t, x}
v = z
Neighbours of v = {x, s}

Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$
while Q not empty **do**
  $v \leftarrow Q.pop()$
  for all neighbours u of v **do**
    if $d(v) + e(v, u) \leq d(u)$ **then**
      $d(u) \leftarrow d(v) + e(v, u)$
    end if
  end for
end while



Q = {t, x}
v = z
Neighbours of v = {x, s}
d(z) + e(z, x) ≤ d(x)   True
d(z) + e(z, s) ≤ d(s)   False

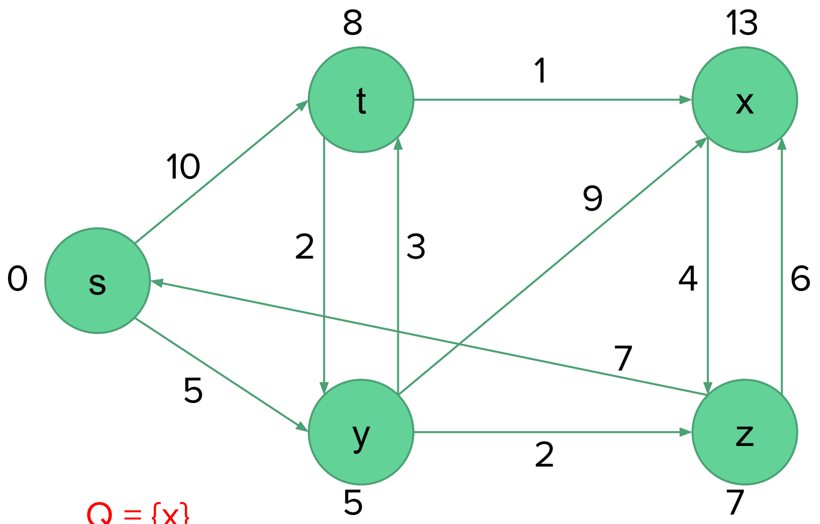
Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**
    $v \leftarrow Q.pop()$
    for all neighbours u of v **do**
        if $d(v) + e(v,u) \leq d(u)$ **then**
            $d(u) \leftarrow d(v) + e(v,u)$
        end if
    end for
end while



Q = {x}
v = t
Neighbours of v = {x, y}

Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**

  $v \leftarrow Q.pop()$

  for all neighbours u of v **do**

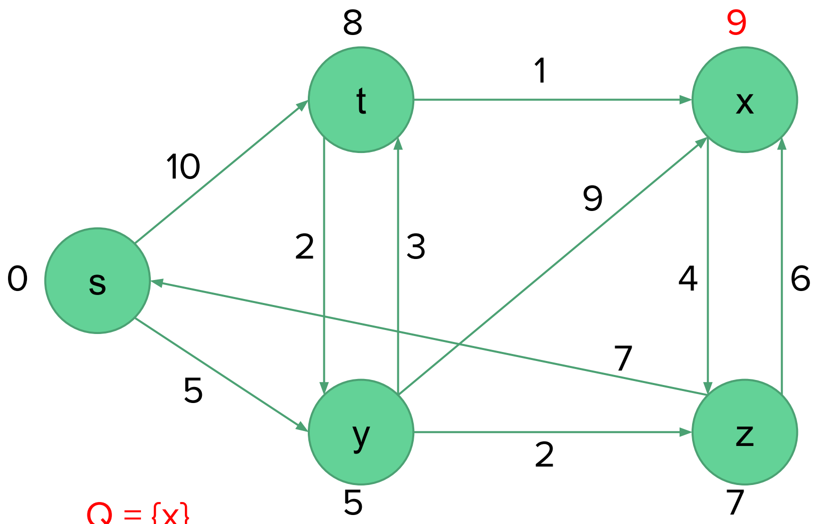
    if $d(v) + e(v,u) \leq d(u)$ **then**

      $d(u) \leftarrow d(v) + e(v,u)$

    end if

  end for

end while



Q = {x}

v = t

Neighbours of v = {x, y}

d(t) + e(t, x) ≤ d(x) True

d(t) + e(t, y) ≤ d(y) False

Running time

- Depends on the implementation
- The simplest implementation is to store vertices in an array or linked list. This will produce a running time of $O(|V|^2 + |E|)$
- For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of $O((|E| + |V|) \log |V|)$

A* search algorithm

A* (pronounced ‘**A-star**’) is a search algorithm that finds the shortest path between some nodes S and T in a graph

It is a **generalization of Dijkstra's algorithm** that cuts down on the size of the subgraph that must be explored using a **heuristic function**

Suppose we want to get to node T, and we are currently at node v. Informally, a heuristic function h(v) is a function that ‘estimates’ how v is away from T

A* search algorithm

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $\boxed{d(v) + h(v)}$

while Q not empty **do**

  $v \leftarrow Q.pop()$

  for all neighbours u of v **do**

    if $d(v) + e(v,u) \leq d(u)$ **then**

      $d(u) \leftarrow d(v) + e(v,u)$

    end if

  end for

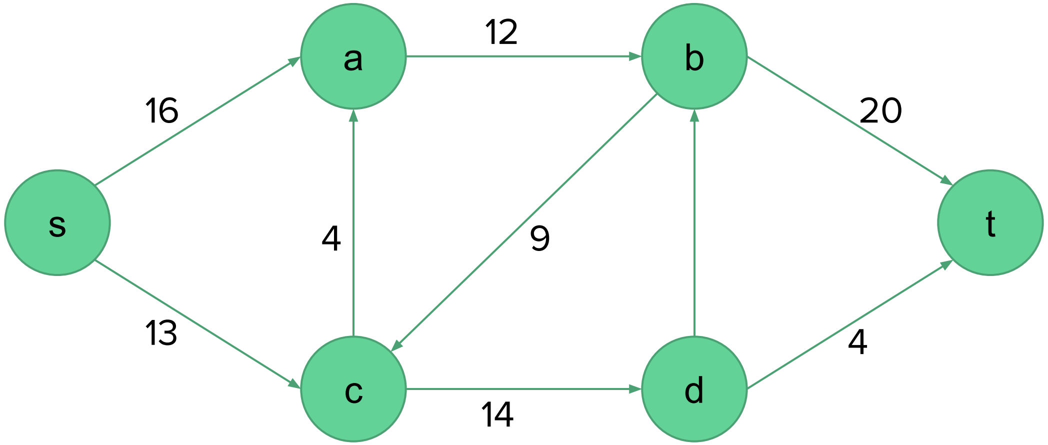
end while

Dijkstra's algorithm is a special case of A*, when we set h(v) = 0 for all v

Flow networks

Flow network

- A directed connected graph G = (V, E), where each edge e is associated with its capacity c(e) > 0.
- If E contains an edge (u, v), then there is no edge (v, u) in the reverse direction
- We distinguish two vertices: a source s and a sink t (s ≠ t)



Flow network

A **flow** in G is a real-valued function f : V x V ➜ R that satisfies the following two properties:

Capacity constraint: Flow on edge e does not exceed its capacity c(e)
That is, for all u, v $\in$ V, we require $0 \leq f(u, v) \leq c(u, v)$

Flow conservation: For every node v ≠ s, t (nodes other than s and t), incoming flow is equal to the outgoing flow.
That is, for all u $\in$ V - {s, t}, we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

Network flow problem (maximum-flow)

Given: A flow network G with source s and sink t.

Problem: Find a flow of maximum value

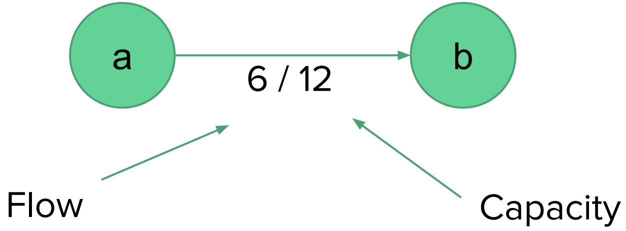
Network flow problem

Alternative formulation: Minimum cut

- Remove some edges from the graph such that after removing the edges, there is no path from s to t
- The cost of removing e is equal to its equal to its capacity c(e)
- The minimum cut problem is to find a cut with total minimum cut

Terminologies

Residual capacity of an edge



Formally, given a flow network G = (V, E) with capacity c and flow f, the residual capacity $c_f(u, v)$, where u, v ∈ V, is defined by

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E , \\ f(v,u) & \text{if } (v,u) \in E , \\ 0 & \text{otherwise} . \end{cases}$$

Terminologies

A **Residual graph/network** contains all the edges that have strictly positive residual capacity.

Formally, given a flow network G = (V, E) and a flow f, the residual network of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

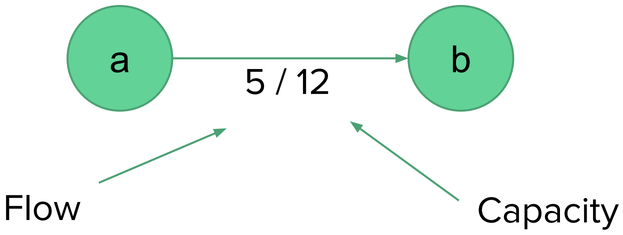
The edges in E_f are either edges in E or their reversals.

Terminologies

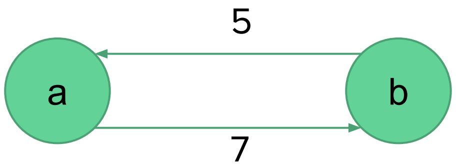
Given a flow network G and a flow f, the **residual network** G_f is constructed by placing

1. Edges with residual capacity $c_f(u, v) = c(u, v) - f(u, v)$
2. Edges with residual capacity $c_f(v, u) = f(u, v)$ (representing a decrease of a flow)

Example:



An edge in a flow network



Corresponding edge in the induced residual network

Terminologies

Augmenting path

Given a flow network $G = (V, E)$ and a flow f, an **augmenting path** p is a simple path from s to t in the residual network G_f.

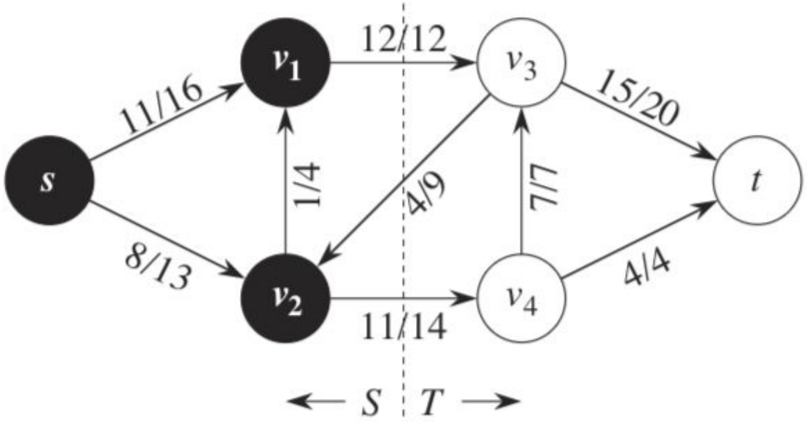
By the definition of the residual network, we may increase the flow on an edge (u,v) of an augmenting path by up to $c_f(u, v)$ without violating the capacity constraint on whichever of (u, v) and (v, u) is in the original flow network G.

If there are augmenting paths, then the flow is not maximum yet.

Terminologies

Cuts of flow networks

A cut (S, T) of flow network G = (V, E) is a partition of V into S and T = V - S such that s ∈ S and t ∈ T.



If f is a flow, then the **net flow** f(S,T) across the cut (S, T) is

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

The **capacity** of the cut (S, T) is

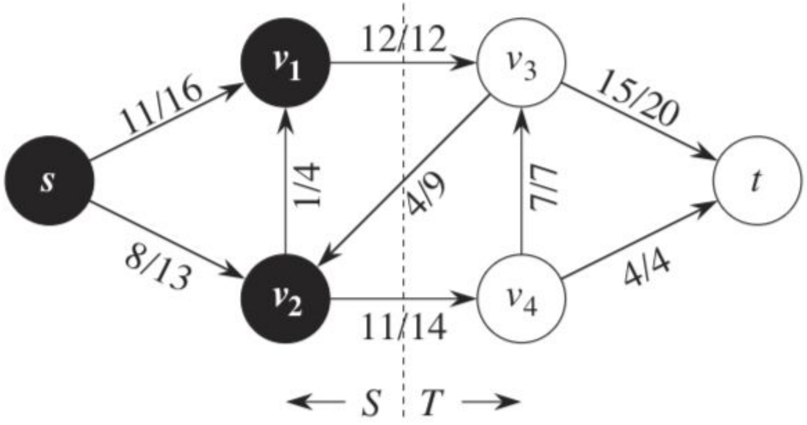
$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

A minimum cut of a network is a cut whose capacity is minimum over all cuts of the network.

Terminologies

Cuts of flow networks

A cut (S, T) of flow network G = (V, E) is a partition of V into S and T = V - S such that s ∈ S and t ∈ T.



Example:
Here the cut is ({s, v1, v2}, {v3, v4, t}).

Net flow = f(v1, v3) + f(v2, v4) - f(v3, v2)
= 12 + 11 - 4
= 19

Capacity = c(v1, v3) + v(2, v4) = 12 + 14 = 26

Max-flow min-cut theorem

Theorem 26.6 (Max-flow min-cut theorem)
If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

Ford-Fulkerson method

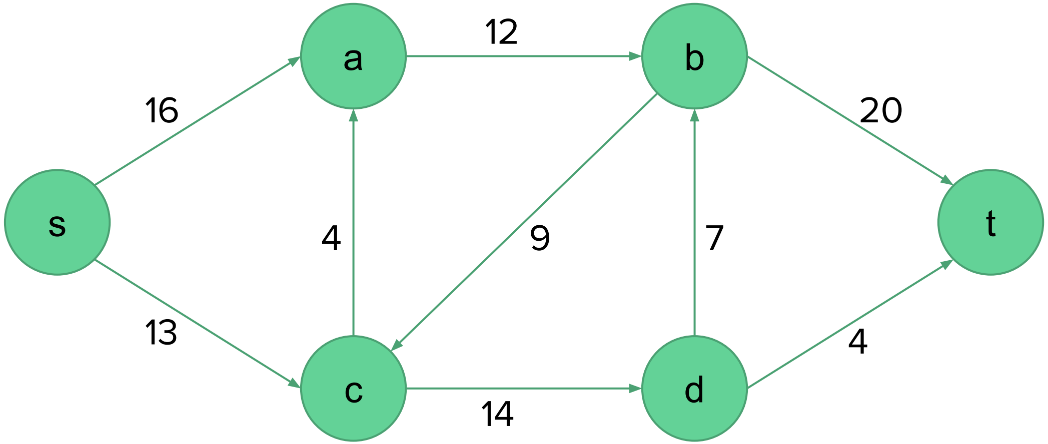
Main idea:

Find valid flow paths until there is none left, and add them up.

Steps:

1. Set $f_{total} = 0$
2. Repeat until there is no path from s to t
   a. Run DFS from s to find a flow path to t
   b. Let f be the minimum capacity value on the path (aka bottleneck capacity)
   c. Add f to f_{total}
   d. For each edge $u \rightarrow v$ on the path
      i. Decrease $c(u \rightarrow v)$ by f
      ii. Increase $c(v \rightarrow u)$ by f

Example

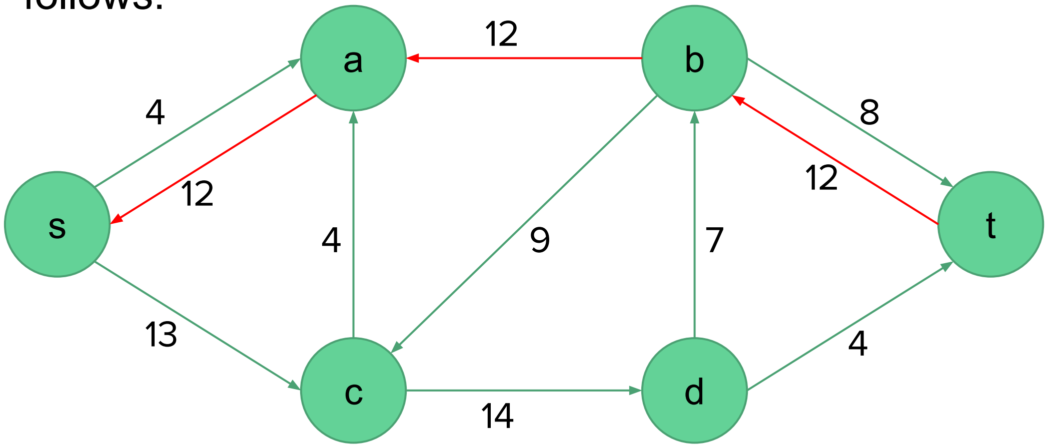


Augmenting paths:

	Min. capacity
s → a → b → t	12
s → c → d → t	4
s → c → a → b → t	4
s → c → d → b → t	7
s → a → b → c → d → t	4

Example

If we choose the augmenting path s → a → b → t, the induced residual graph would be as follows:



Augmenting paths:

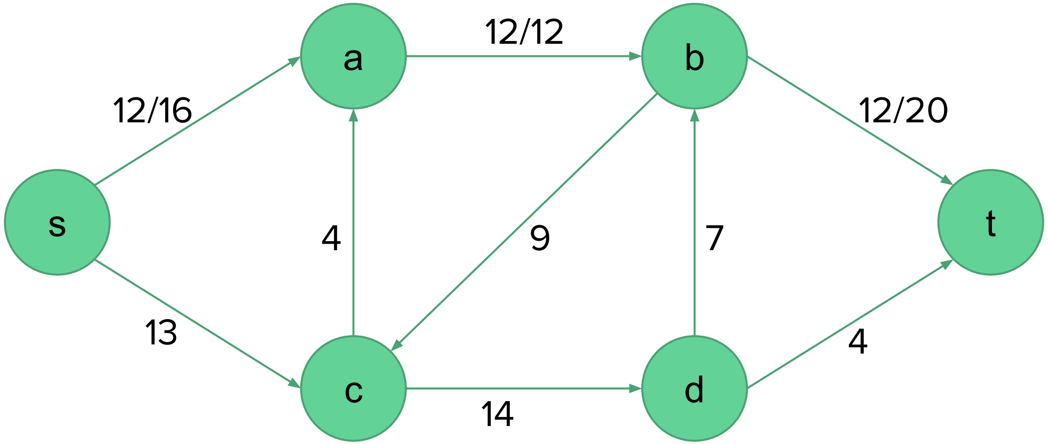
	Min. capacity
s → c → d → t	4
s → c → d → b → t	7

Selected augmenting paths:

	Flow
s → a → b → t	12

Example

And the corresponding flow network would be as follows:



Augmenting paths:

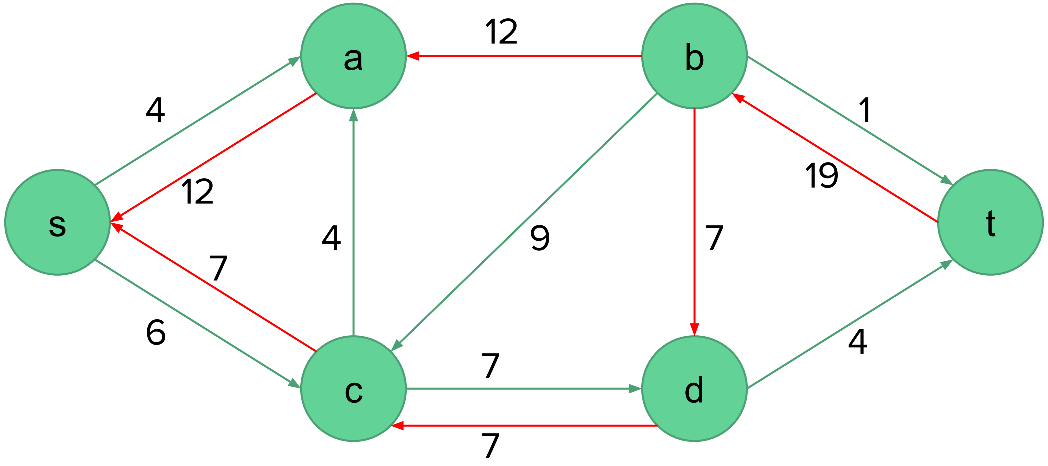
	Min. capacity
s → c → d → t	4
s → c → d → b → t	7

Selected augmenting paths:

	Flow
s → a → b → t	12

Example

If we choose the augmenting path s → c → d → b → t, the residual graph would be as follows:



Augmenting paths:

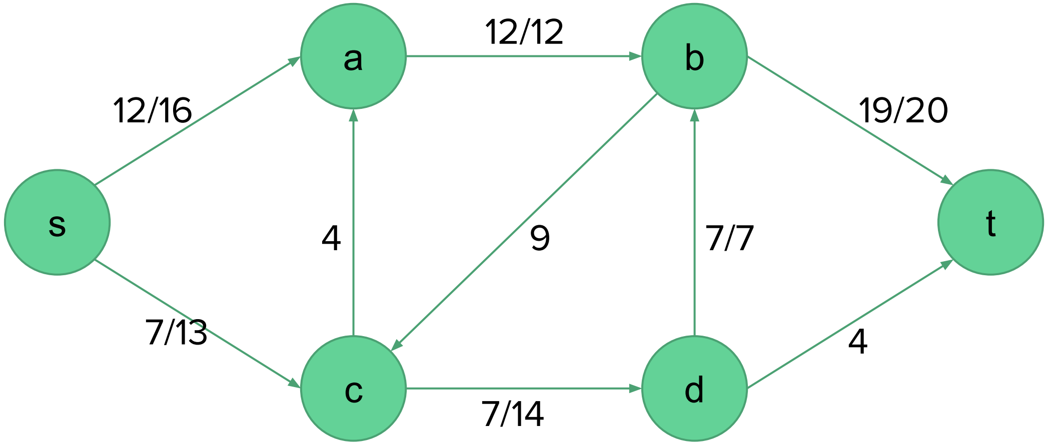
	Min. capacity
s → c → d → t	4

Selected augmenting paths:

	Flow
s → a → b → t	12
s → c → d → b → t	7

Example

And the corresponding flow network would be as follows:



Augmenting paths:

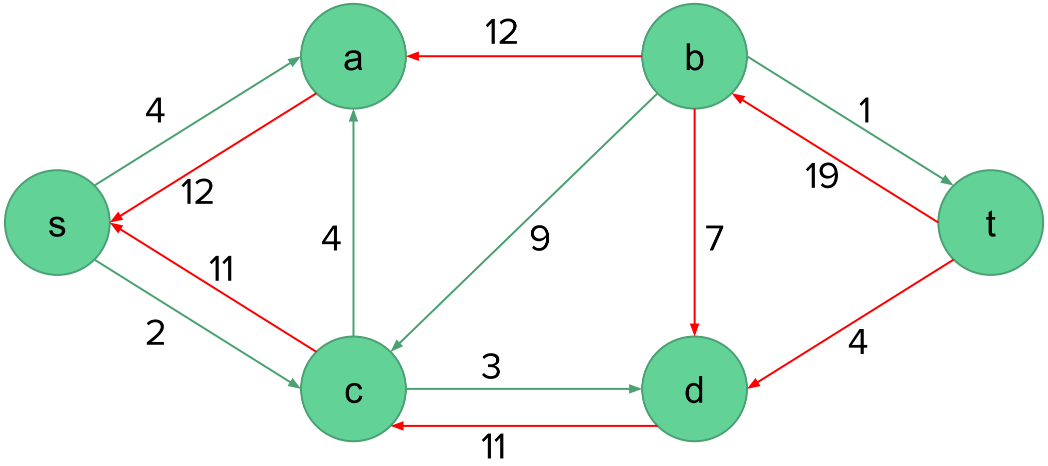
	Min. capacity
s → c → d → t	4

Selected augmenting paths:

	Flow
s → a → b → t	12
s → c → d → b → t	7

Example

If we choose the augmenting path s → c → d → t, the residual graph would be as follows:



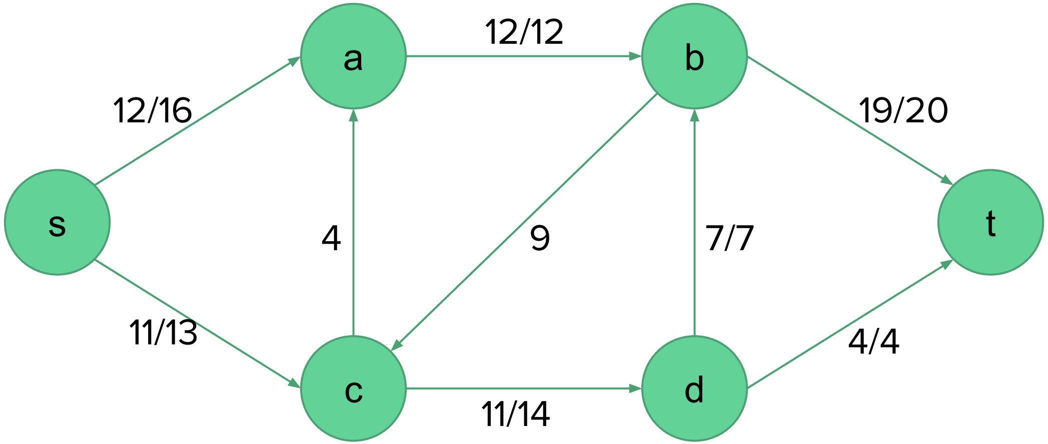
No more augmenting paths

Selected augmenting paths:

	Flow
s → a → b → t	12
s → c → d → b → t	7
s → c → d → t	4

Example

There is no path from s to t, thus the max flow network is as below:



No more augmenting paths

Selected augmenting paths:

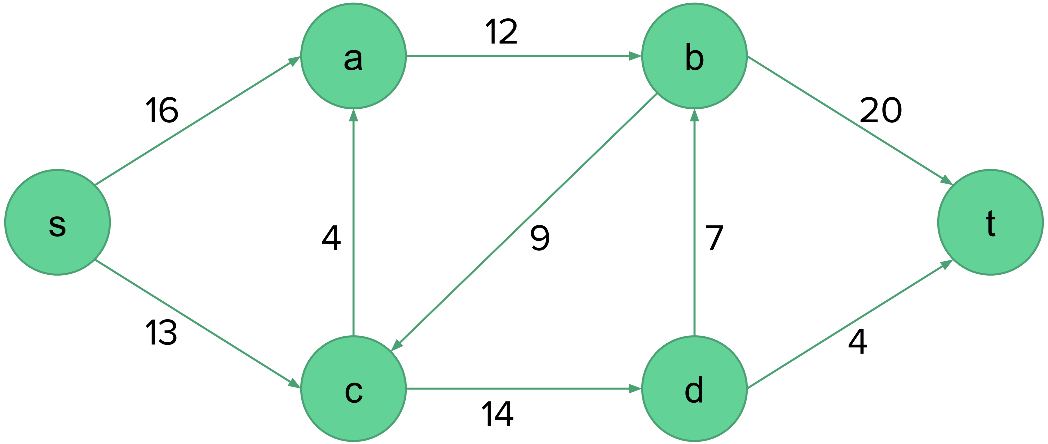
	Flow
s → a → b → t	12
s → c → d → b → t	7
s → c → d → t	4
Total flow	23

Computing minimum cut

- To compute minimum cut, we use the residual graph induced by the max flow
- Mark all nodes reachable from s
  - Call this set of reachable nodes A
- Now separate these nodes from the others
  - Cut edges from A to V-A

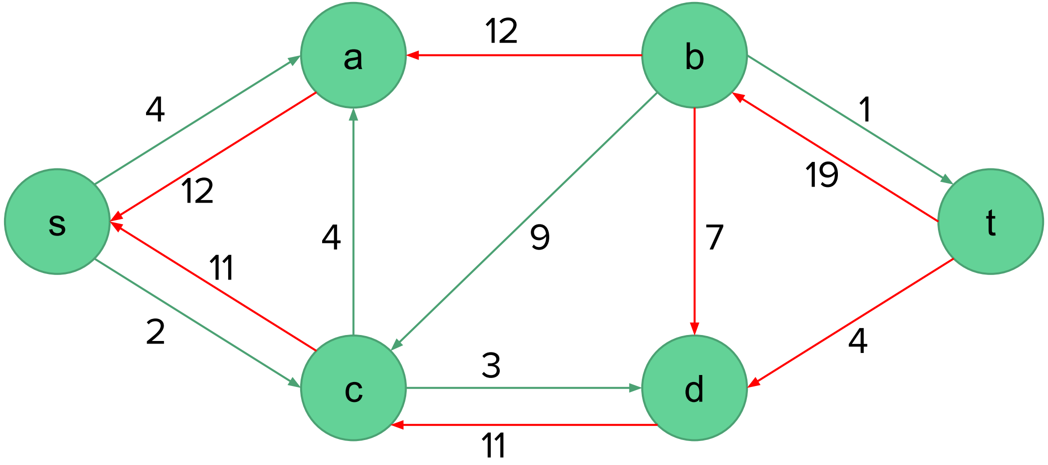
Computing minimum cut

Example: Compute the minimum cut in the following flow network



Computing minimum cut: Example

The residual graph induced by the max flow is as below:

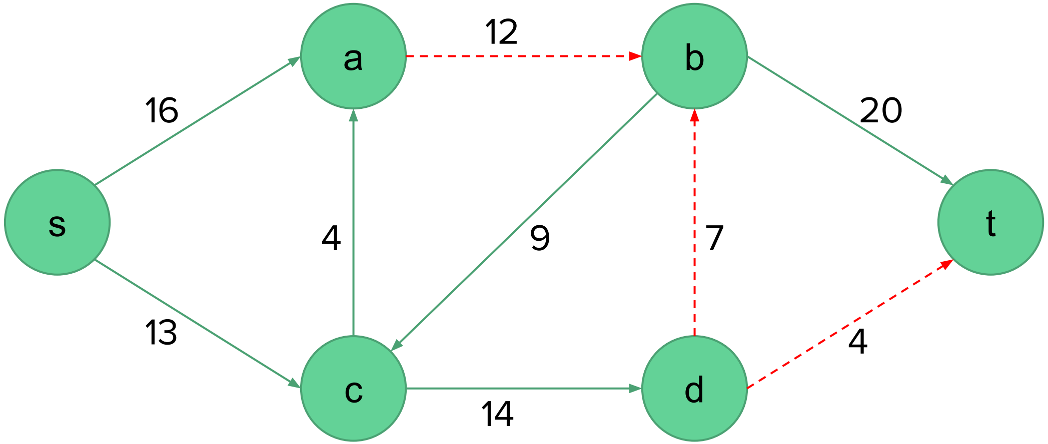


Nodes reachable from s
A = {a, c, d}

V-A = {b, t}

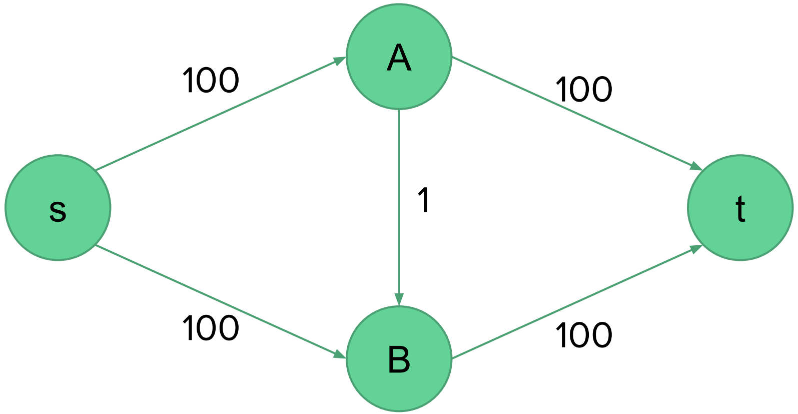
Computing minimum cut: Example

Cut edges from A to V-A



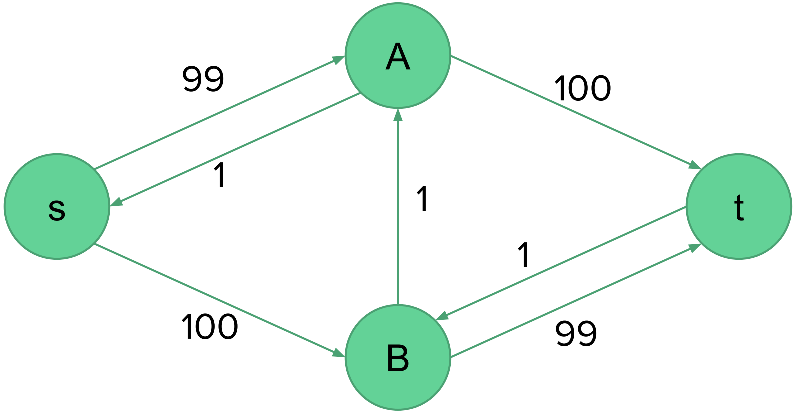
Analysis of Ford-Fulkerson

- Running time depends on how we find augmenting paths
- If we choose it poorly, the algorithm might not even terminate
- For example, consider the following network



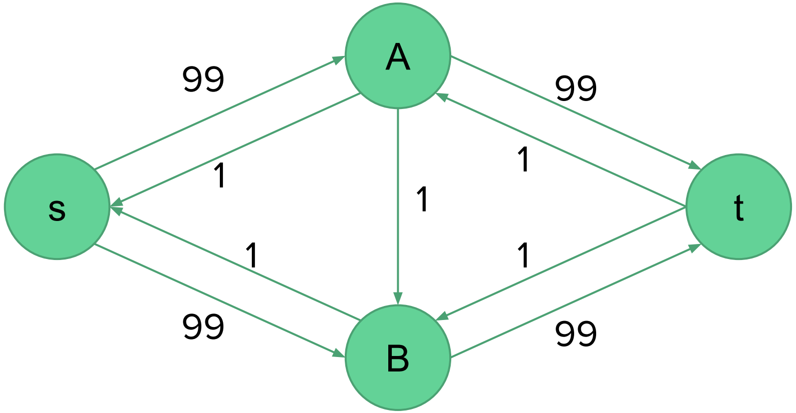
Analysis of Ford-Fulkerson

Augmenting path: s → A → B → t



Analysis of Ford-Fulkerson

Augmenting path: $s \rightarrow B \rightarrow A \rightarrow t$



Analysis of Ford-Fulkerson

If we find the augmenting path by using a BFS/DFS, the algorithm runs in polynomial time.

Analysis of Ford-Fulkerson

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p in the residual network G_f
3      augment flow f along p
4  return f
```

If we perform DFS/BFS to find augmenting paths, it will take O(V+E) time to find an augmenting path.

The while loop of line 2-4 executes for at most | f | times since we may add 1 unit flow in each iteration.

Thus, total time of Ford-Fulkerson algorithm is O (|f| (V + E)).

Applications

- Airline scheduling
    - Managing flight crews by reusing them over multiple flights
- Network connectivity
    - Routing as many packets as possible on a given network
    - Finding min number of connections (edges) whose removal disconnects t from s
- Project selection
    - Choosing a feasible subset of projects to maximize revenue
- Bipartite matching etc.
    - Finding an assignment of jobs to applicants in such that as many applicants as possible get jobs.