ECS 32B - Graphs

Aaron Kaloti

UC Davis - Summer Session #2 2020

Overview

Nothing from this slide deck will be on exam #3.

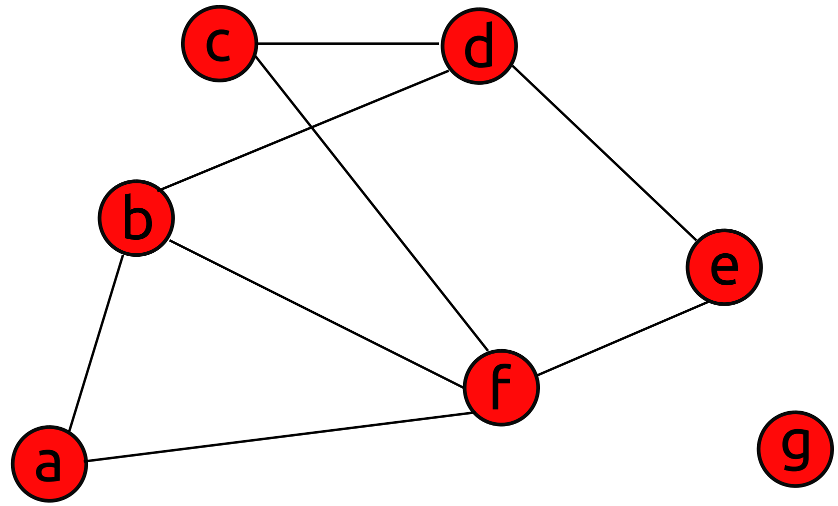
- Definition of a graph.
- Common representations of graphs.
- Graph traversals:
  - Breadth-first search.
  - Depth-first search.
- Single-source shortest-path (SSSP) problem.
  - Dijkstra's algorithm.
- Minimum spanning tree (MST) problem.
  - Prim's algorithm.
  - Kruskal's algorithm.

Defining a Graph

Terminology

Basic Sets

- Denoted $G = (V, E)$.
  - V: set of nodes/vertices.
    - $n = |V|$.
  - E: set of edges.
    - $m = |E|$.



Paths and Cycles

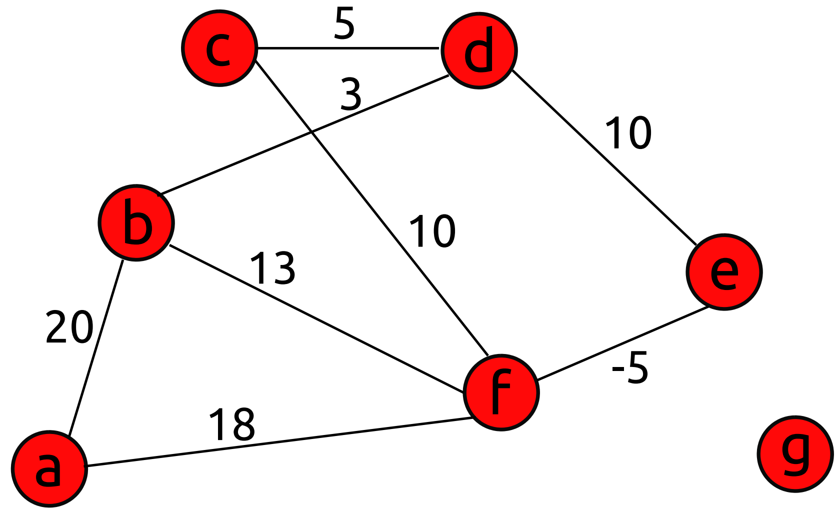
- **Path**: sequence of nodes, edge between every two consecutive nodes in sequence.
  - Example: (a, b, d, e) and (f, b, a, f, e) are paths. (f, b, c) is not.
- **Simple path**: path with distinct vertices.
  - Example: (a, b, d, e) is a simple path. (f, b, a, f, e) is not.
- **Cycle**: path in which all nodes are distinct, except for the first and last nodes, which must be the same.
  - Example: (c, f, e, d, c) is a cycle. (f, b, a, f, e) is not.
  - **Acyclic graph**: graph that has no cycles

Defining a Graph

Terminology

Weighted Graph

- In weighted graph, each edge e has a weight c_e.

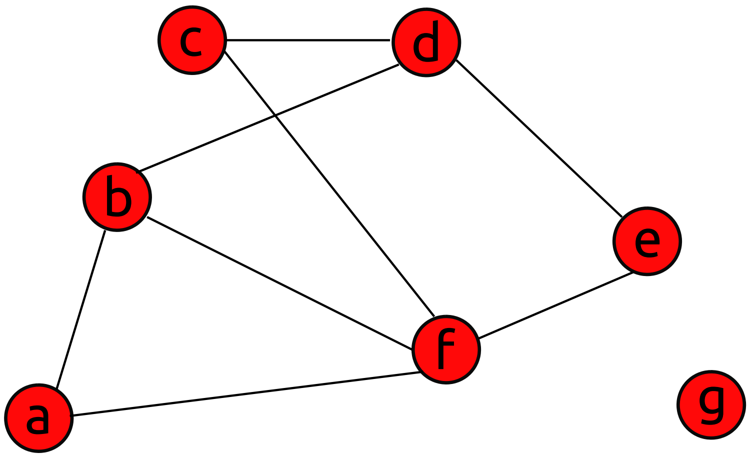


Defining a Graph

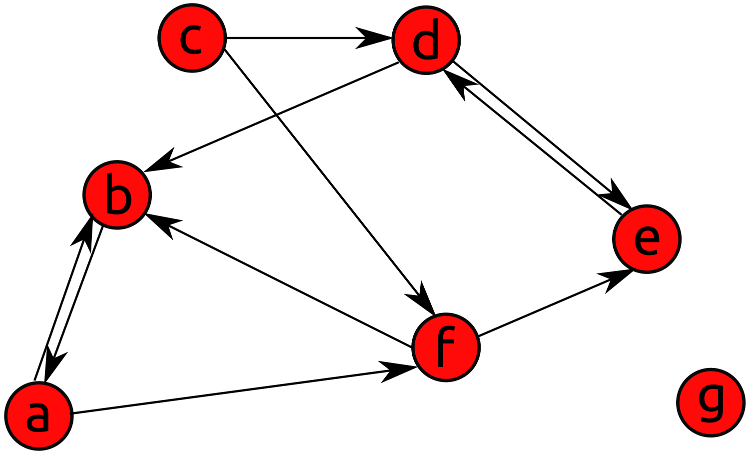
Terminology

Direction

- **Undirected** graph: each edge $e = \{u, v\}$ has no direction.
  - u and v are the **ends** of e



- **Directed** graph (**digraph**): each edge $e' = (u, v)$ is an ordered pair
  - u is **tail** of e'
  - v is **head** of e'
  - e' **leaves** u and **enters** v
  - Affects paths/cycles:
    - (a, b, d, e) no longer a path. (f, b, a, f, e) still is.
    - (c, f, e, d, c) no longer a cycle. (a, f, b, a) is.



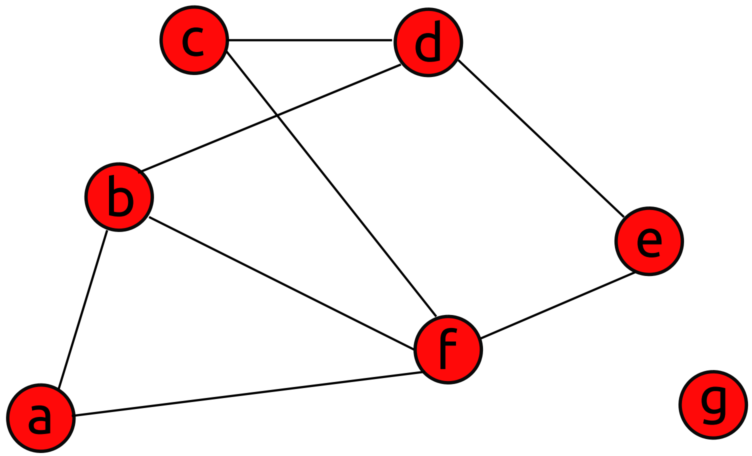
Defining a Graph

Terminology

Incidence and Degree

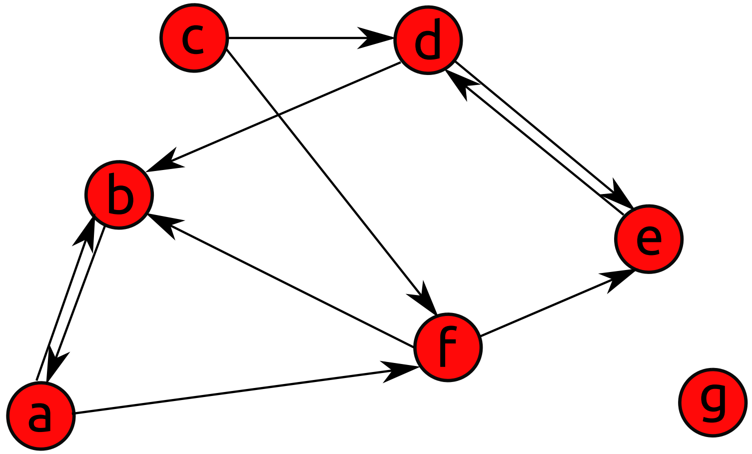
Undirected Graph

- **degree** of $v \in V$, denoted d_v, is number of edges touching v.



Directed Graph

- **in-degree** of $v \in V$ is number of incoming edges.
- **out-degree** of v is number of outgoing edges.

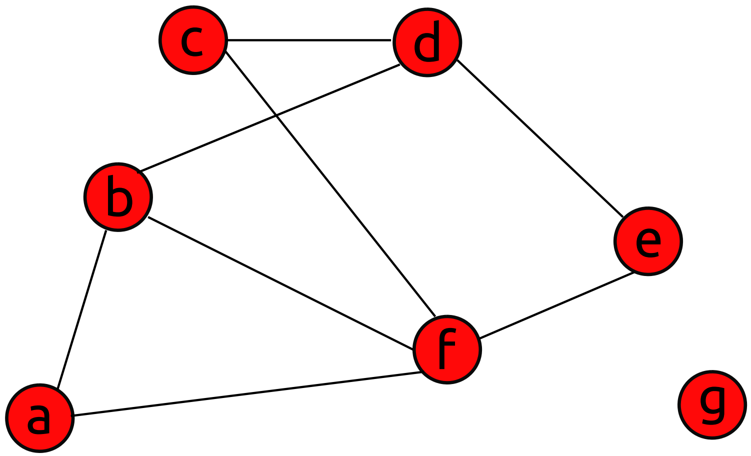


Graph Connectivity

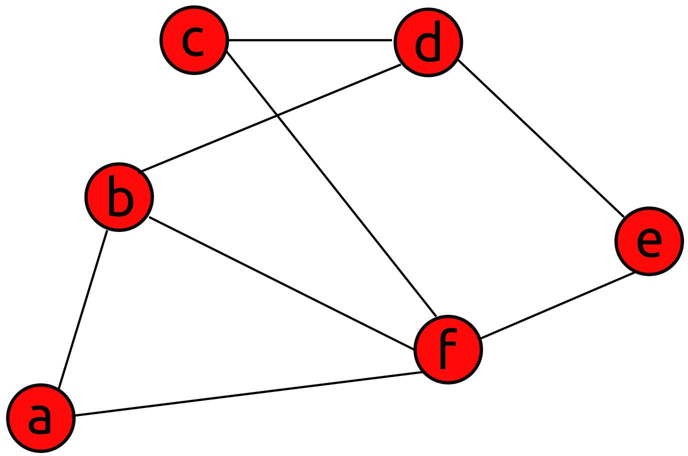
Connected

- *Undirected* graph is **connected** if for every pair of nodes u and v, there's a path from u to v.

Example 1: Not Connected



Example 2: Connected



Graphs: Real-World Examples[1]

Remarks

- For the below, $u, v \in V$.
- Notice when edges are directed vs. not.

Airline Routes

- V: airports.
- $(u, v) \in E$ if flight from u to v.

Social Networks

- V: users on Facebook.
- $\{u, v\} \in E$ if u and v are friends.

Communication Networks

Wired Networks

- V: computers.
- $\{u, v\} \in E$ if is direct physical link connecting u and v.

Wireless Networks

- V: computers.
- $(u, v) \in E$ if v is close enough to u to receive a signal from v.

Dependency Networks

Course Prerequisites

- V: courses.
  - e.g. ECS 122A, ECS 150
- $(u, v) \in E$ if u is a prerequisite for v.
  - e.g. (ECS 36B, ECS 36C) $\in E$.

Subproblem Graph

- V: subproblems.
- $(u, v) \in E$ if u must be completed before v can be completed.

1. Most of these examples are from section 3.1 of *Algorithm Design* by Jon Kleinberg and Éva Tardos.

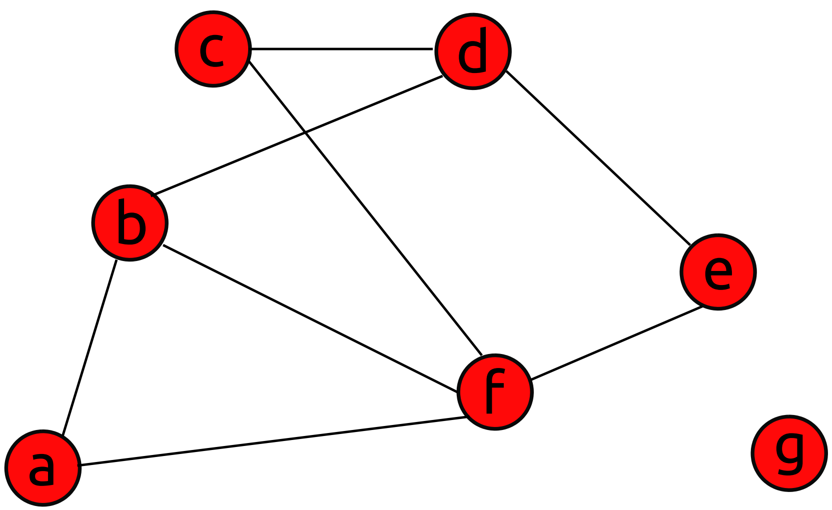
Representing Graphs

Adjacency Matrix (of graph $G = (V, E)$)

Definition

- n-by-n matrix A where $A[u, v] = 1$ (or "true") if either:
  1. G is undirected and $\{u, v\} \in E$
  2. G is directed and $(u, v) \in E$
- G is undirected $\iff$ A is symmetric (i.e. $A[u, v] = A[v, u]$).

Example: Undirected Graph



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

Space Complexity

- $\Theta(n^2)$

Weighted Graph

- Put the weight if the edge exists; `null` otherwise (or -1 or 0, if those are invalid edge weights).

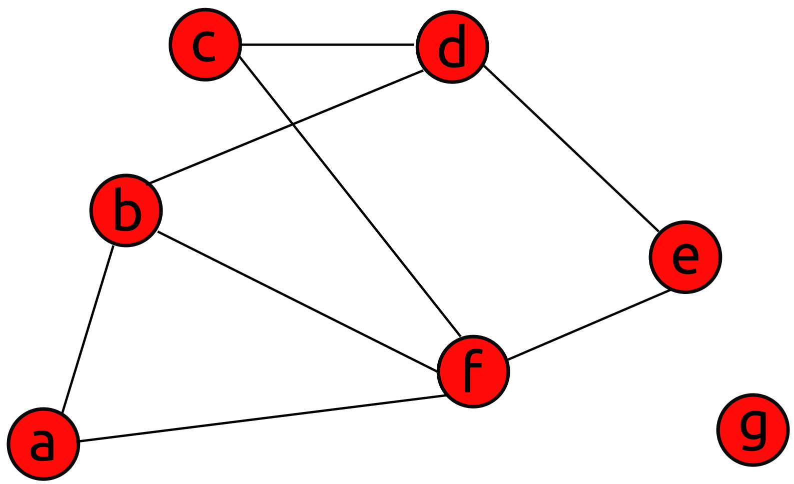
Representing Graphs

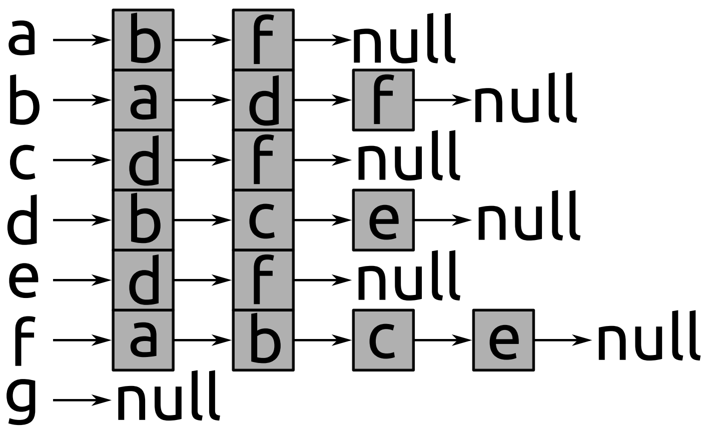
Adjacency List (of graph $G = (V, E)$)

Definition

- Each $v \in V$ has a (linked) list of nodes to which v has edges.

Example: Undirected Graph





Space Complexity[1]

- $\Theta(2m + n) = \Theta(m + n)$

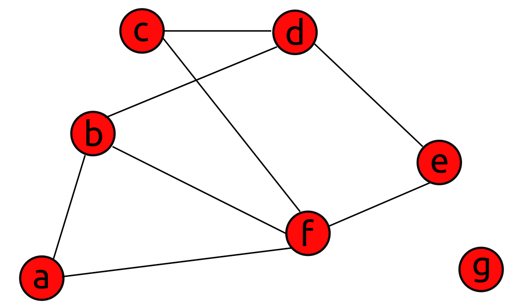
Weighted Graph

- Attach edge weight with each element of adjacency list.

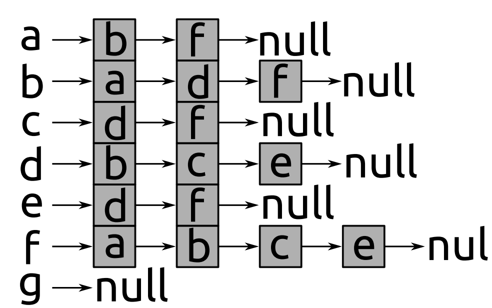
Representing Graphs

Tradeoffs

- Space
  - Dense graph → adjacency matrix.
    - Example: complete graph, transitive closure.
  - Sparse graph → adjacency list.
    - Almost all graphs are sparse.
- Edge lookup.
  - $\Theta(1)$ time with adjacency matrix.
  - $\Theta(n)$ time in worst case with adjacency list (if adjacency list is not Python list).
    - $Theta(\lg n)$ is possible, if adjacency list is Python list.
- Examine ALL edges incident to a given node.
  - $\Theta(n)$ time with adjacency matrix.
  - $O(n)$ time with adjacency list, but much better in practice.
    - Alternatively: $\Theta(max(d_v))$, where d_v is degree of a given $v \in V$.



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

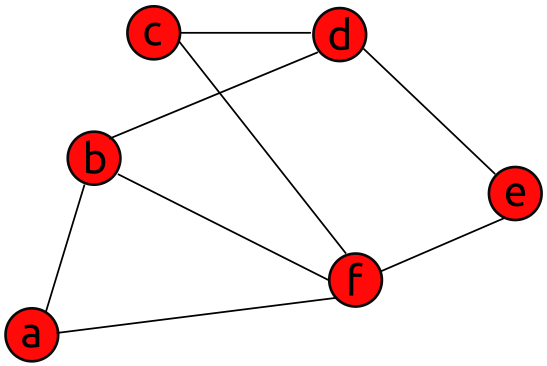


Graph Traversals

Breadth-First Search (BFS)

Description[1]

- Layer-by-layer traversal.
- May be multiple valid BFS orderings.
    - One possible order (if start at a): a, f, b, e, c, d.
    - Another: a, b, f, d, c, e.



Implementation with One Queue

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

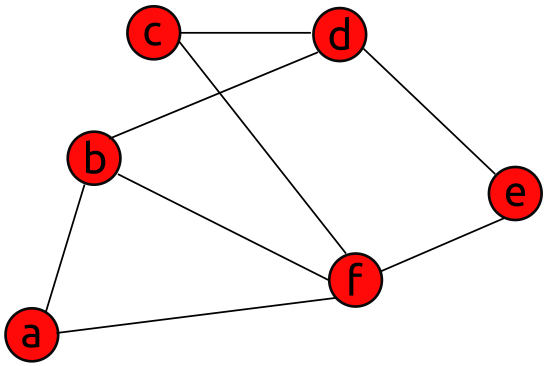
Graph Traversals

Breadth-First Search (BFS)

Example

- Queue: uninitialized
- Discovered:

a	b	c	d	e	f



Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

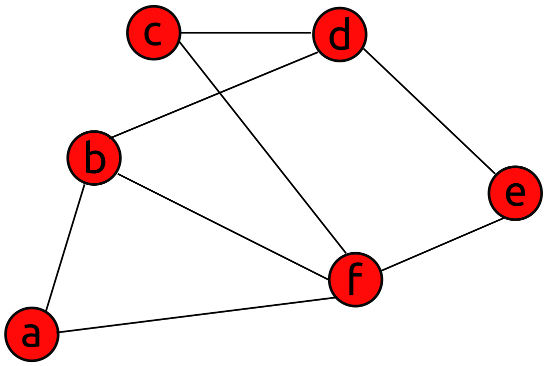
Graph Traversals

Breadth-First Search (BFS)

Example

- Processing order: none
- Queue: uninitialized
- Discovered:

a	*b*	*c*	*d*	*e*	*f*



Modified Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

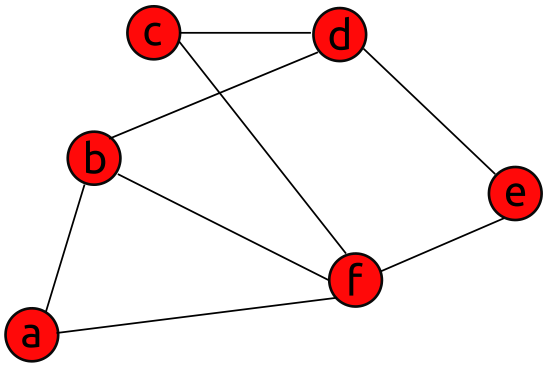
Graph Traversals

Breadth-First Search (BFS)

Example

- Processing order: none
- Queue: a
- Discovered:

a	b	c	d	e	f
T	F	F	F	F	F



Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

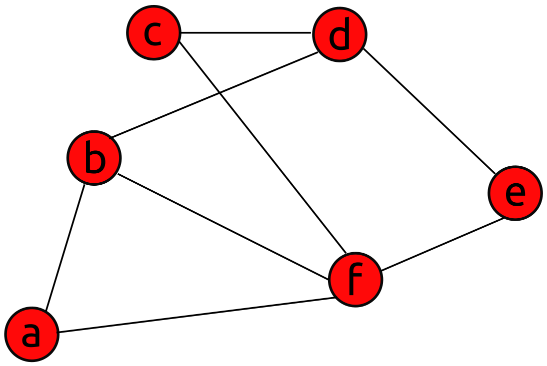
Graph Traversals

Breadth-First Search (BFS)

Example

- Processing order: a
- Queue: empty
- Discovered:

a	b	c	d	e	f
T	F	F	F	F	F



Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

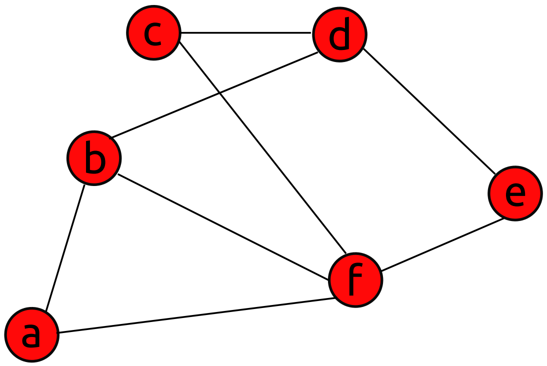
Graph Traversals

Breadth-First Search (BFS)

Example

- Processing order: a
- Queue: b, f
- Discovered:

a	b	c	d	e	f
T	T	F	F	F	T



Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

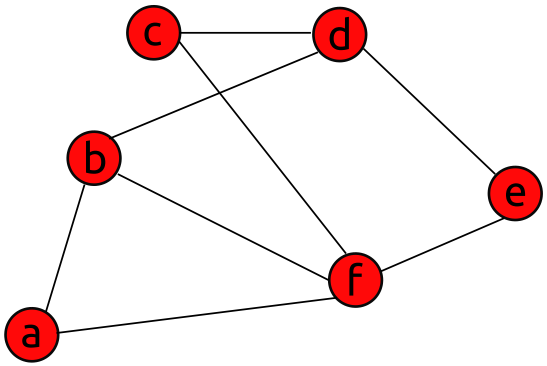
Graph Traversals

Breadth-First Search (BFS)

Example

- Processing order: a, b
- Queue: f
- Discovered:

a	b	c	d	e	f
T	T	F	F	F	T



Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

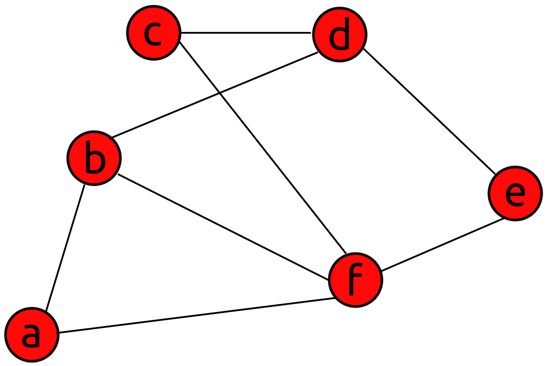
Graph Traversals

Breadth-First Search (BFS)

Example

- Processing order: a, b
- Queue: f, d
- Discovered:

a	b	c	d	e	f
T	T	F	T	F	T



Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

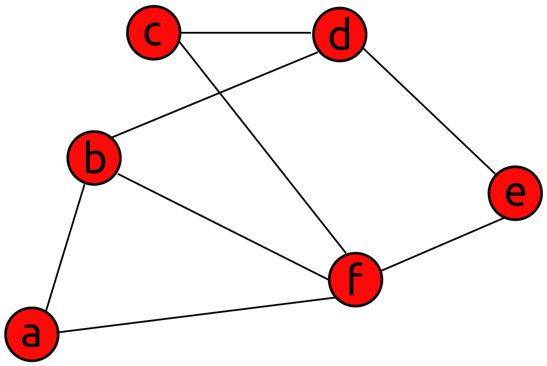
Graph Traversals

Breadth-First Search (BFS)

Example

- Processing order: a, b, f
- Queue: d
- Discovered:

a	b	c	d	e	f
T	T	F	T	F	T



Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

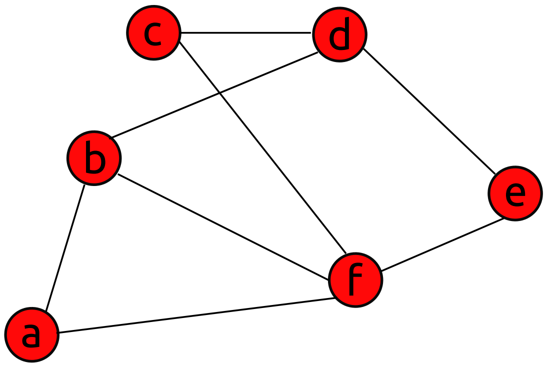
Graph Traversals

Breadth-First Search (BFS)

Example

- Processing order: a, b, f
- Queue: d, c, e
- Discovered:

a	b	c	d	e	f
T	T	T	T	T	T



Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

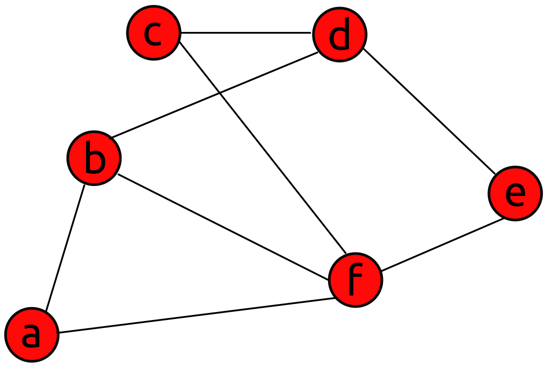
Graph Traversals

Breadth-First Search (BFS)

Example

- Processing order: a, b, f, d
- Queue: c, e
- Discovered:

a	b	c	d	e	f
T	T	T	T	T	T



Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

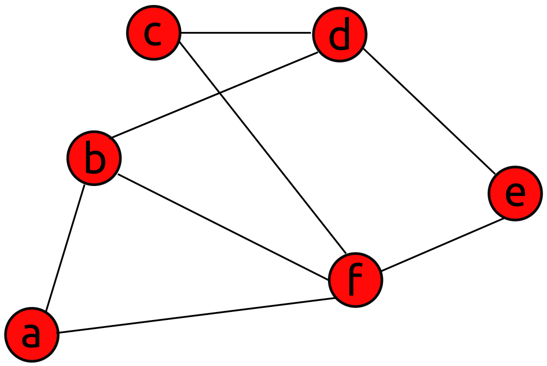
Graph Traversals

Breadth-First Search (BFS)

Example

- Processing order: a, b, f, d, c
- Queue: e
- Discovered:

a	b	c	d	e	f
T	T	T	T	T	T



Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

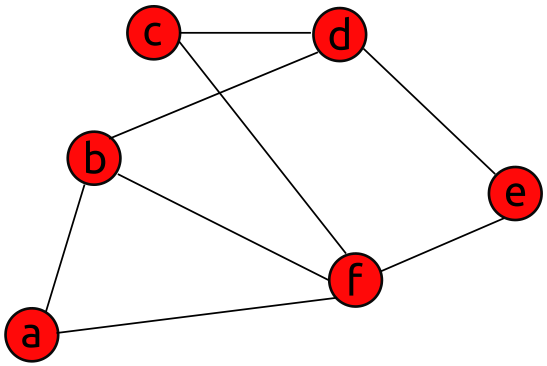
Graph Traversals

Breadth-First Search (BFS)

Example

- Processing order: a, b, f, d, c, e
- Queue: empty
- Discovered:

a	b	c	d	e	f
T	T	T	T	T	T



Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

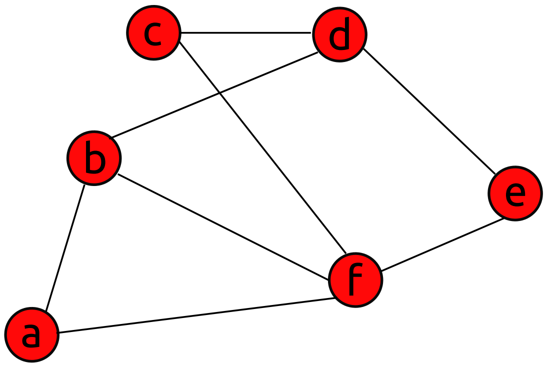
Graph Traversals

Breadth-First Search (BFS)

Example

- **Final order**: a, b, f, d, c, e
- Queue: empty
- Discovered:

a	b	c	d	e	f
T	T	T	T	T	T



Pseudocode

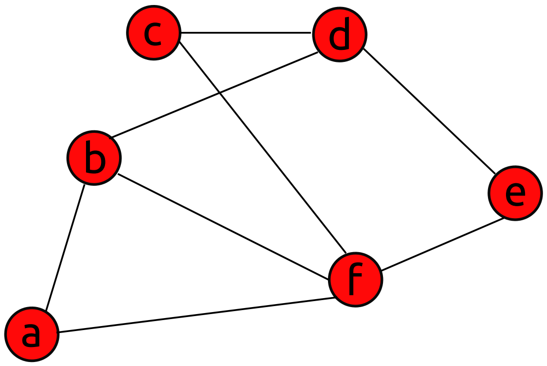
```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```

Graph Traversals

Breadth-First Search (BFS)

Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```



Analysis

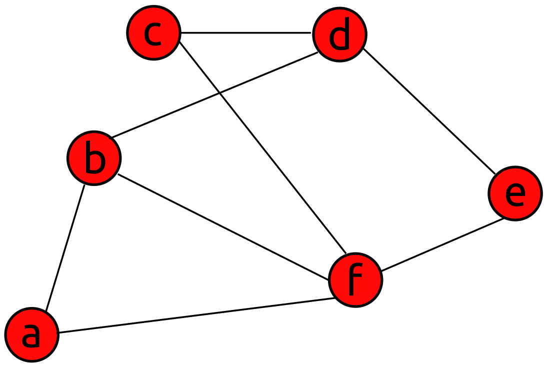
- Time complexity: $\Theta(n^2)$?
    - Outer loop iterates $O(n)$ times.
    - Inner loop iterates $O(n)$ times.
- It's $O(n^2)$, but the tight bound is better.

Graph Traversals

Breadth-First Search (BFS)

Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```



Analysis

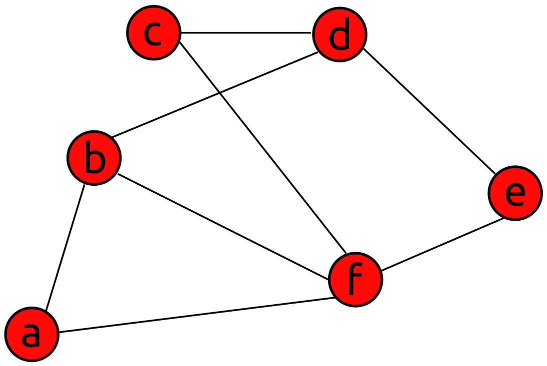
- *Per `while` loop iteration*, how many times does the `for` loop execute *at most*?
- Tempting to say n times.
- If processed vertex v in `while` loop, `for` loop executes d_v times, where d_v is degree of v .

Graph Traversals

Breadth-First Search (BFS)

Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```



Analysis

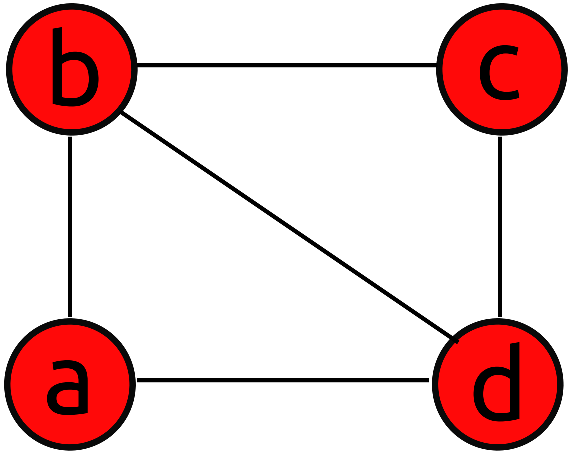
- By the handshaking lemma (degree sum formula): $\sum_{v \in V} d_v = 2m$.

Graph Traversals

Breadth-First Search (BFS)

Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```



Analysis

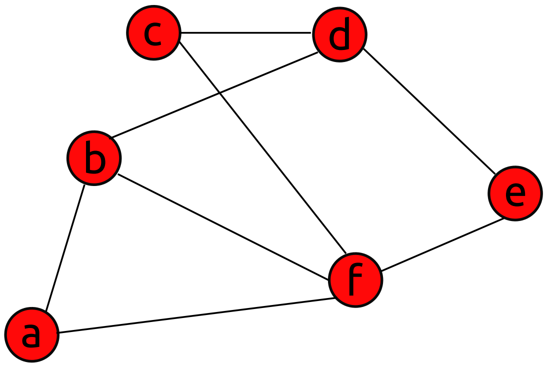
- Recall: $\sum_{v \in V} d_v = d_a + d_b + d_c + d_d = 2 + 3 + 2 + 3 = 10 = 2 \cdot 5 = 2m$, where $m = |E|$.
- Total number of `for` loop iterations *in the entire execution of BFS* is $2m$.

Graph Traversals

Breadth-First Search (BFS)

Pseudocode

```
For each vertex v
    Set Discovered[v] = false
Set Discovered[a] = true
Initialize queue q with a
While q not empty
    curr_node = q.dequeue()
    Process curr_node
    For each neighbor n of curr_node
        If Discovered[n] is false
            Discovered[n] = true
            q.enqueue(n)
```



Analysis

- Time complexity: $\Theta(m+n)$[1]
  - $\Theta(n)$ queue operations, each $O(1)$ time
  - Inner `for` loop iterates d_v times per outer loop iteration, for total of $\sum_{v\in V} d_v = 2m = \Theta(m)$ inner `for` loop iterations
  - $\Theta(n)$ time to set up `Discovered`.
- Space complexity (auxiliary space): $\Theta(n)$ because `Discovered`

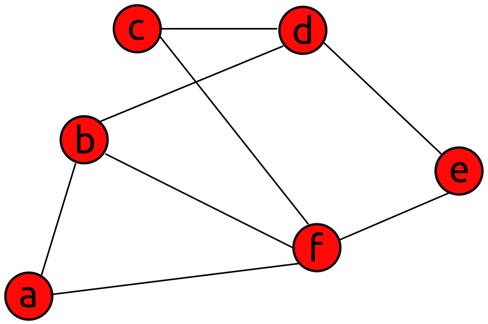
1. Assuming connected graph.

Graph Traversals

Depth-First Search (DFS)

Description

- Go until can't go further.
- May be multiple valid DFS orderings.
    - One possible order (if start at a): a, b, d, c, f, e.
    - Another: a, f, e, d, c, b



Implementation with One Stack

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

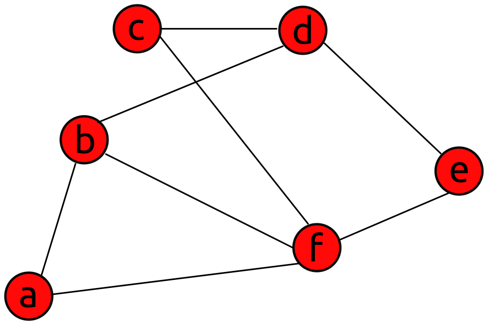
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: none
- Stack: uninitialized

Node	a	b	c	d	e	f
Explored						



Modified Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

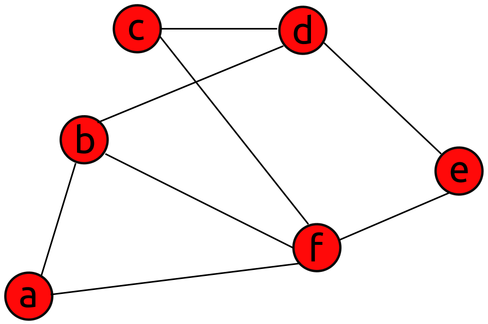
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: none
- Stack: a

Node	a	b	c	d	e	f
Explored	F	F	F	F	F	F



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

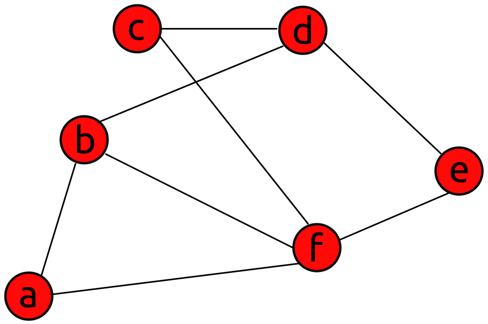
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a
- Stack: none

Node	a	b	c	d	e	f
Explored	T	F	F	F	F	F



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

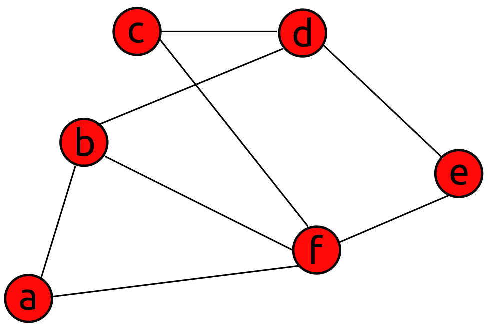
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a
- Stack: b, f

Node	a	b	c	d	e	f
Explored	T	F	F	F	F	F



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

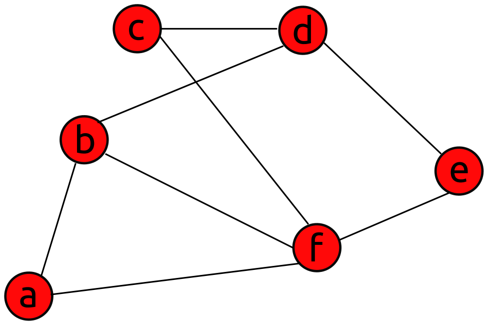
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a, f
- Stack: b

Node	a	b	c	d	e	f
Explored	T	F	F	F	F	T



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

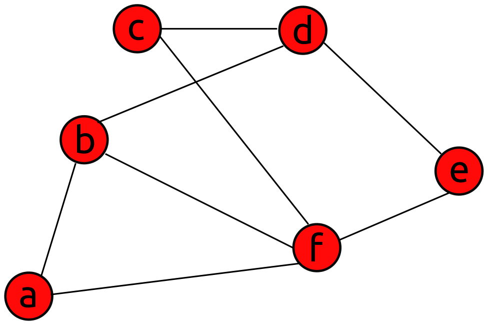
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a, f
- Stack: b, b, c, e

Node	a	b	c	d	e	f
Explored	T	F	F	F	F	T



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

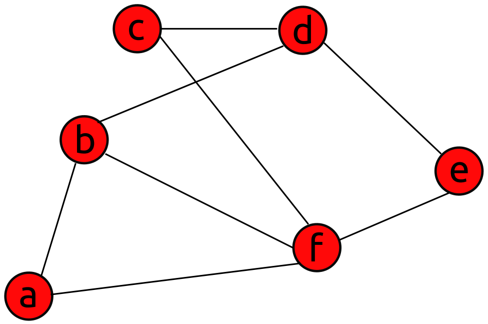
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a, f, e
- Stack: b, b, c

Node	a	b	c	d	e	f
Explored	T	F	F	F	T	T



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

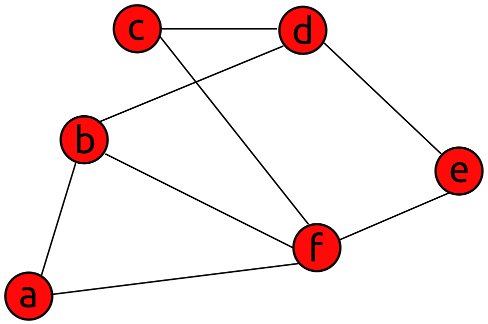
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a, f, e
- Stack: b, b, c, d[1]

Node	a	b	c	d	e	f
Explored	T	F	F	F	T	T



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

1. Technically, based on our pseudocode, f should be pushed into the stack too, but we'll omit f since there's no point.

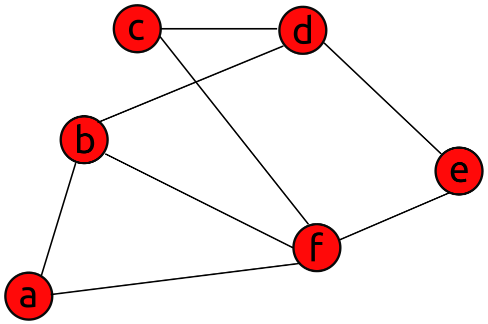
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a, f, e, d
- Stack: b, b, c

Node	a	b	c	d	e	f
Explored	T	F	F	T	T	T



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

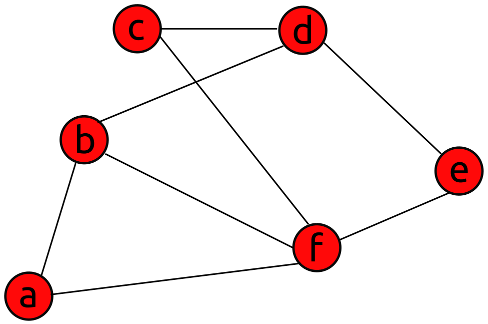
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a, f, e, d
- Stack: b, b, c, b, c

Node	a	b	c	d	e	f
Explored	T	F	F	T	T	T



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

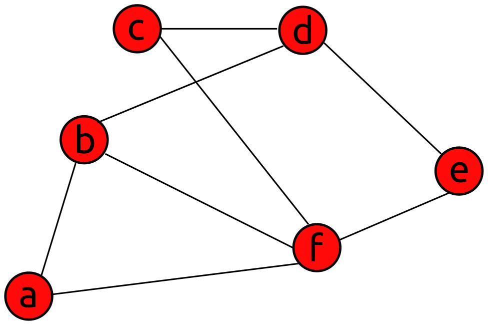
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a, f, e, d, c
- Stack: b, b, c, b

Node	a	b	c	d	e	f
Explored	T	F	T	T	T	T



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

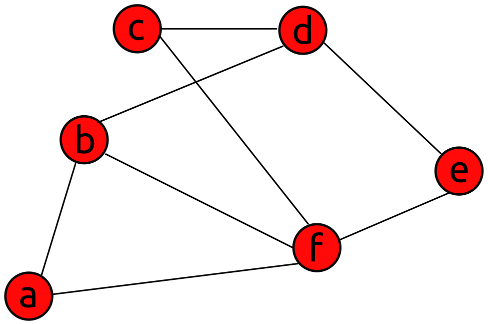
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a, f, e, d, c, b
- Stack: b, b, c

Node	a	b	c	d	e	f
Explored	T	T	T	T	T	T



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

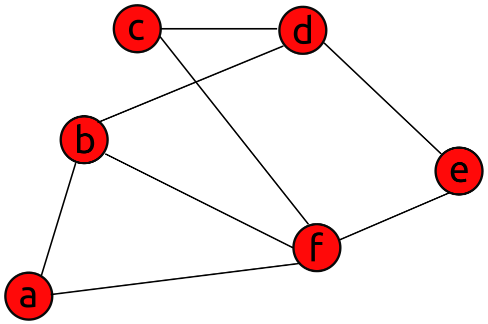
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a, f, e, d, c, b
- Stack: b, b

Node	a	b	c	d	e	f
Explored	T	T	T	T	T	T



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

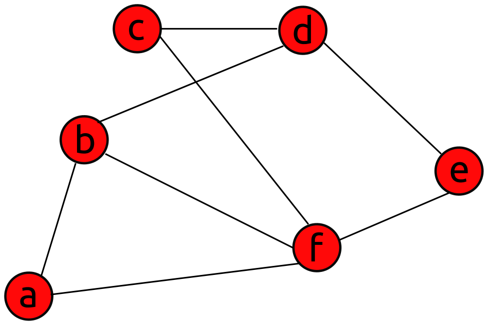
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a, f, e, d, c, b
- Stack: b

Node	a	b	c	d	e	f
Explored	T	T	T	T	T	T



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

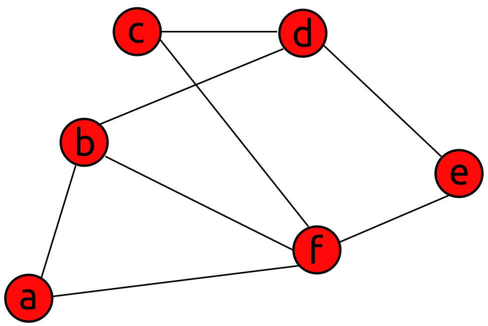
Graph Traversals

Depth-First Search (DFS)

Example

- Processing order: a, f, e, d, c, b
- Stack: empty

Node	a	b	c	d	e	f
Explored	T	T	T	T	T	T



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

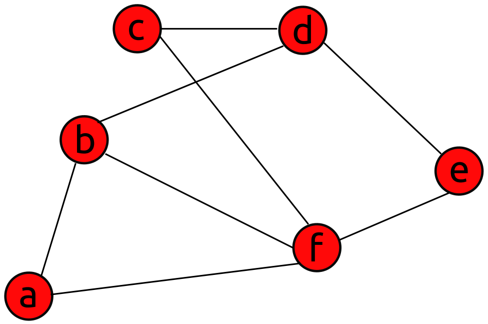
Graph Traversals

Depth-First Search (DFS)

Example

- **Final order**: a, f, e, d, c, b
- Stack: empty

Node	a	b	c	d	e	f
Explored	T	T	T	T	T	T



Pseudocode

```
For each vertex v
    Set Explored[v] = false
Initialize stack s with a
While s not empty
    curr_node = s.pop()
    If Explored[curr_node] is false
        Process curr_node
        Set Explored[curr_node] = true
        For each neighbor n of curr_node
            s.push(n)
```

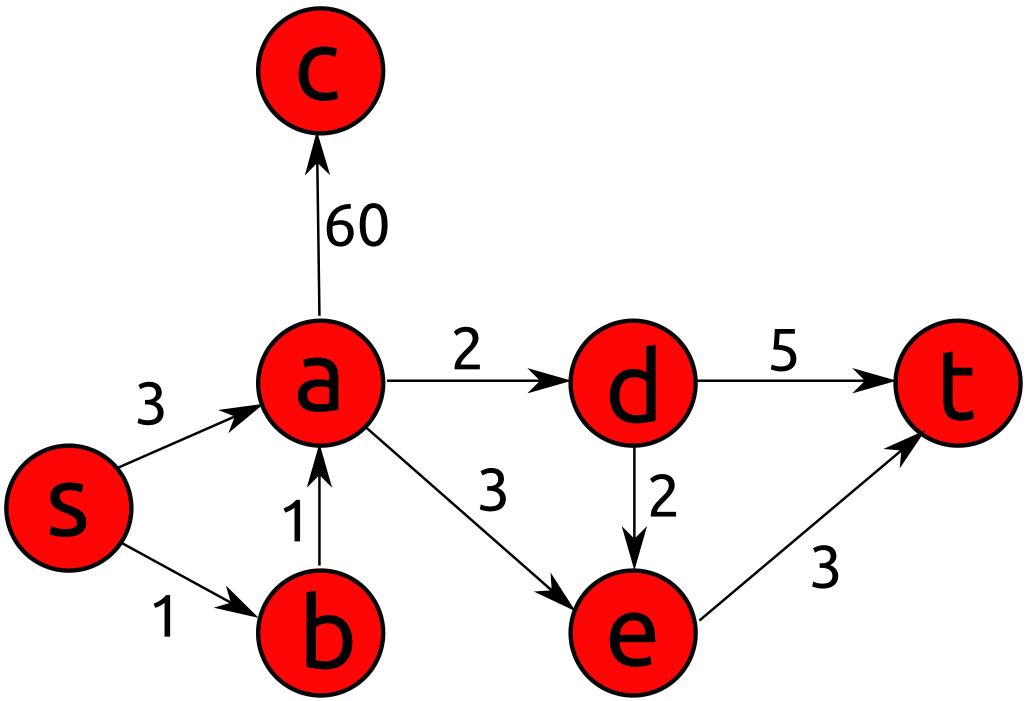
Dijkstra's Algorithm

Single-Source Shortest-Paths (SSSP) Problem

- Input: a *directed*[1], *weighted* graph $G = (V, E)$ and a vertex $s \in V$
- Output: shortest path from s to each of the other nodes

Example

v	Shortest path from s to v	Weight of path
a	s, b, a	2
b	s, b	1
c	s, b, a, c	62
d	s, b, a, d	4
e	s, b, a, e	5
t	s, b, a, e, t	8



1. The graph need not be directed, but the book says the input is a directed graph.

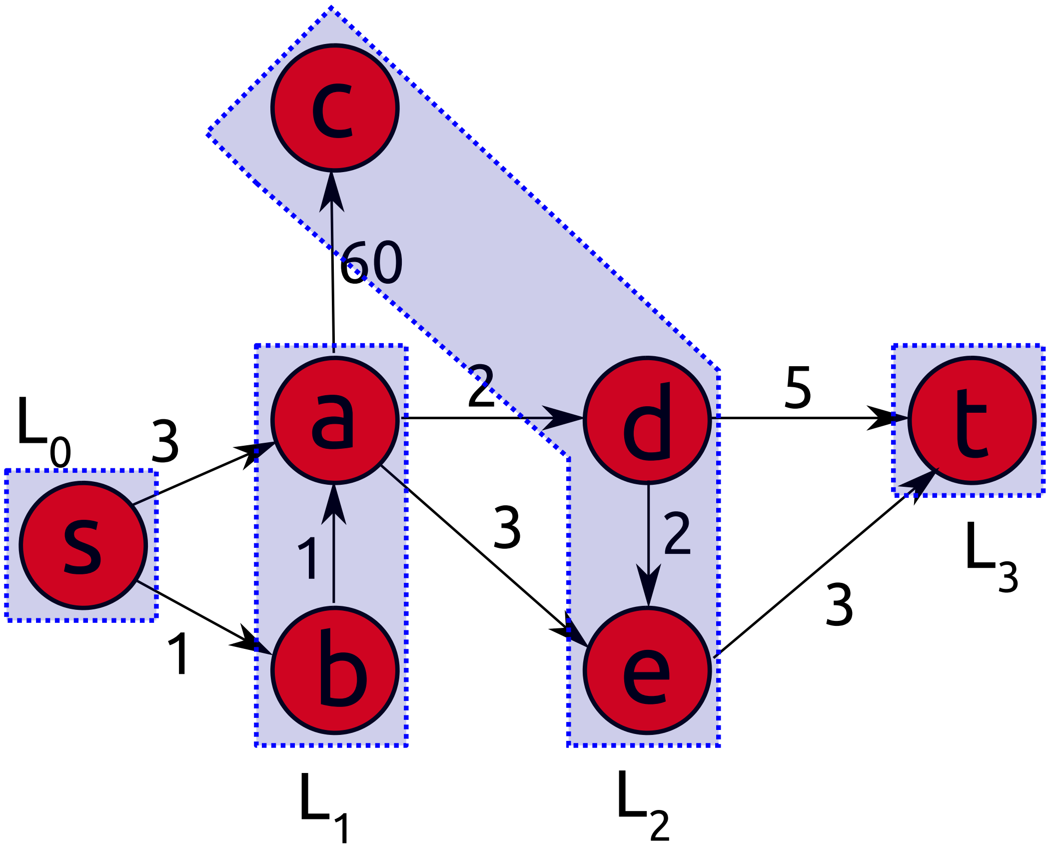
Dijkstra's Algorithm

Single-Source Shortest-Paths (SSSP) Problem

- Input: a *directed*, *weighted* graph $G = (V, E)$ and a vertex $s \in V$
- Output: shortest path from s to each of the other nodes

Layer-Based BFS Fails

- Layer-based BFS only works for SSSP on an *unweighted* graph.



Dijkstra's Algorithm

Description of Algorithm

- Maintains set of "explored" vertices $S \subseteq V$ such that *we know the shortest path* from s to each of the nodes in S.
- Greedily chooses which vertex to add to S next.

Choosing a Vertex to Add

- Choose the vertex with the least costly edge to any vertex in S?

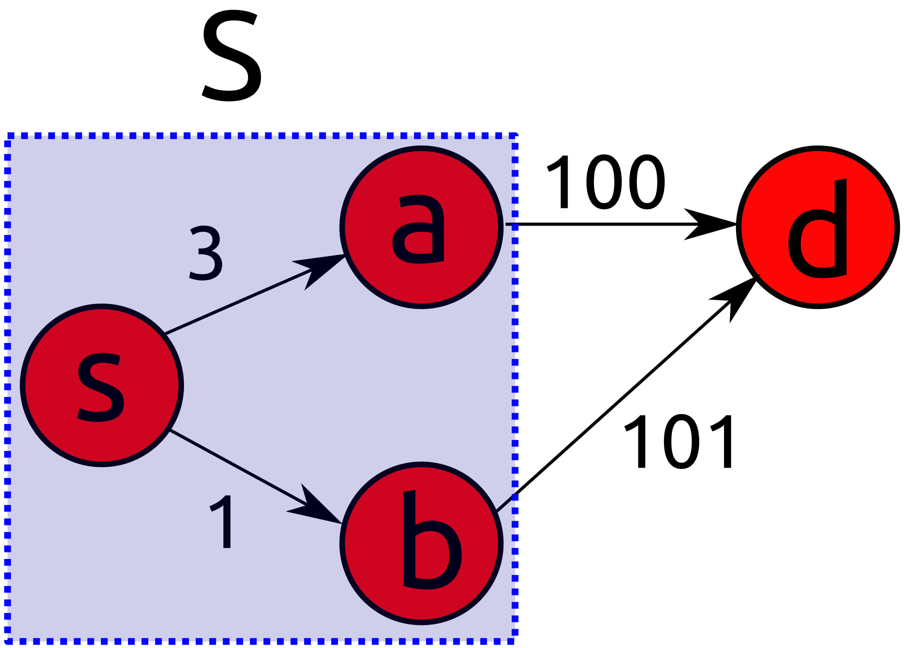
Dijkstra's Algorithm

Description of Algorithm

- Maintains set of "explored" vertices $S \subseteq V$ such that *we know the shortest path* from s to each of the nodes in S.
- Greedily chooses which vertex to add to S next.

Choosing a Vertex to Add

- Choose the vertex with the least costly edge to any vertex in S?
- Doesn't work.



Dijkstra's Algorithm

Description of Algorithm

- Maintains set of "explored" vertices $S \subseteq V$ such that *we know the shortest path* from s to each of the nodes in S.
- Greedily chooses which vertex to add to S next.

Choosing a Vertex to Add

- *Instead*, let $d(u)$ be what the algorithm *currently* thinks is the shortest-path distance from s to u.
    - For each $x \in S \subseteq V$, we know $d(x)$ is final.
- Choose the vertex $v \in V - S$[1] that minimizes $d'(v) = d_u + cost(u, v)$, where $u \in S$ and $(u, v) \in E$.

Termination

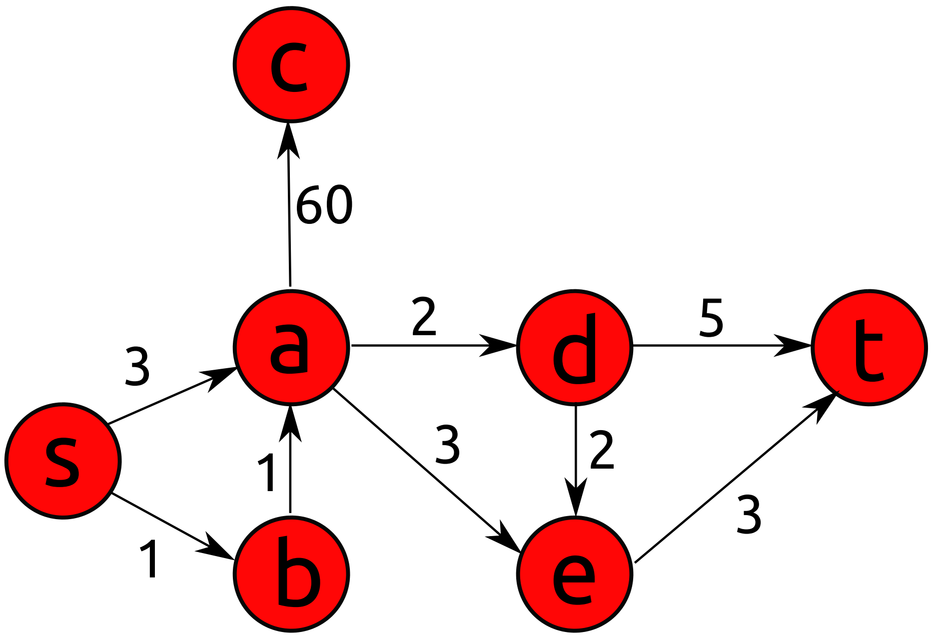
- Repeat until either:
    1. $t \in S$.
        - If only want shortest path from s to t.
    2. $S = V$.
        - If want shortest path from s to all other vertices.

1. Note that $V - S$ and $V \setminus S$ are the same. Both are set subtraction.

Dijkstra's Algorithm

Example

v	$Disc[v]$	$Dist[v]$	$P[v]$
s	False	0	[none]
a	False	∞	NULL
b	False	∞	NULL
c	False	∞	NULL
d	False	∞	NULL
e	False	∞	NULL
t	False	∞	NULL

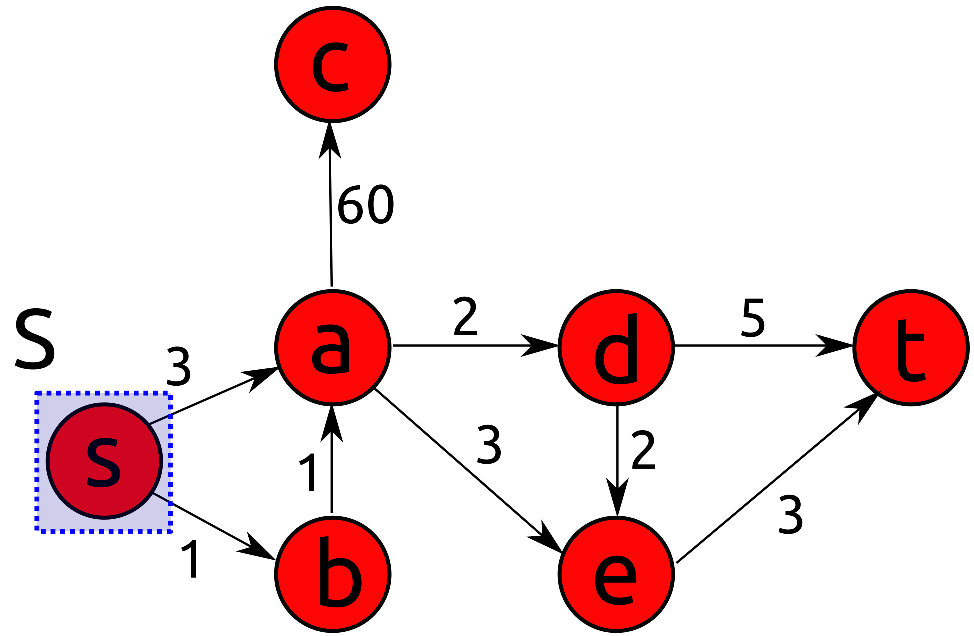


- $Disc[v]$: Has v been processed?
  - If $Disc[v] = True$, then $Dist[v]$ is final.
- $Dist[v]$: What we currently think is the shortest-path distance from s to v.
- $P[v]$: Parent of v, i.e. the node before v in the shortest path from s to v.

Dijkstra's Algorithm

Example

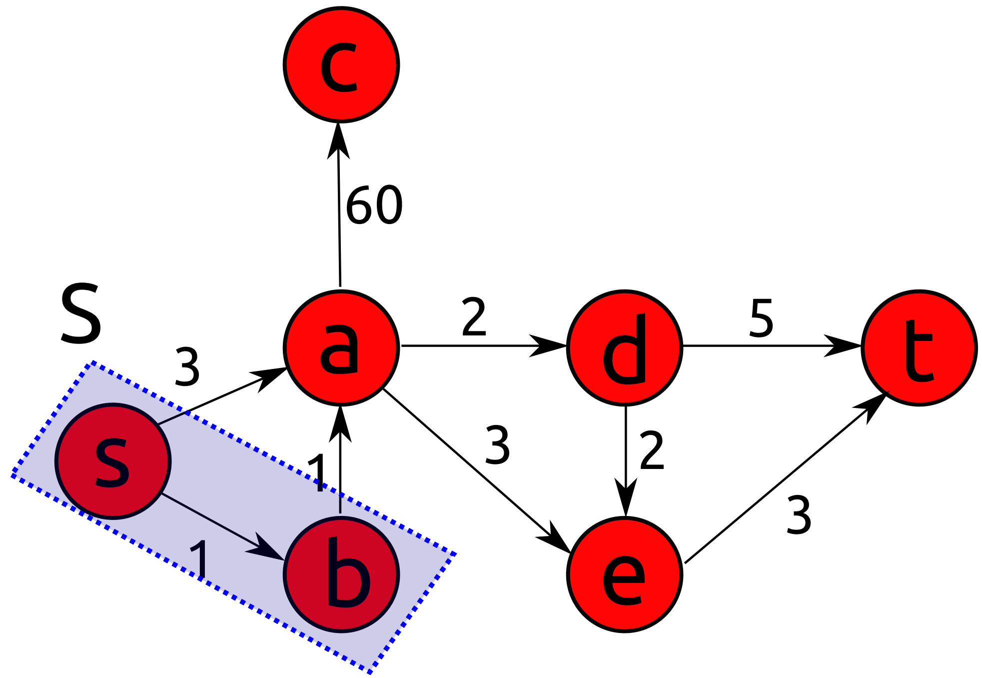
v	$Disc[v]$	$Dist[v]$	$P[v]$
s	True	0	[none]
a	False	3	s
b	False	1	s
c	False	∞	NULL
d	False	∞	NULL
e	False	∞	NULL
t	False	∞	NULL



Dijkstra's Algorithm

Example

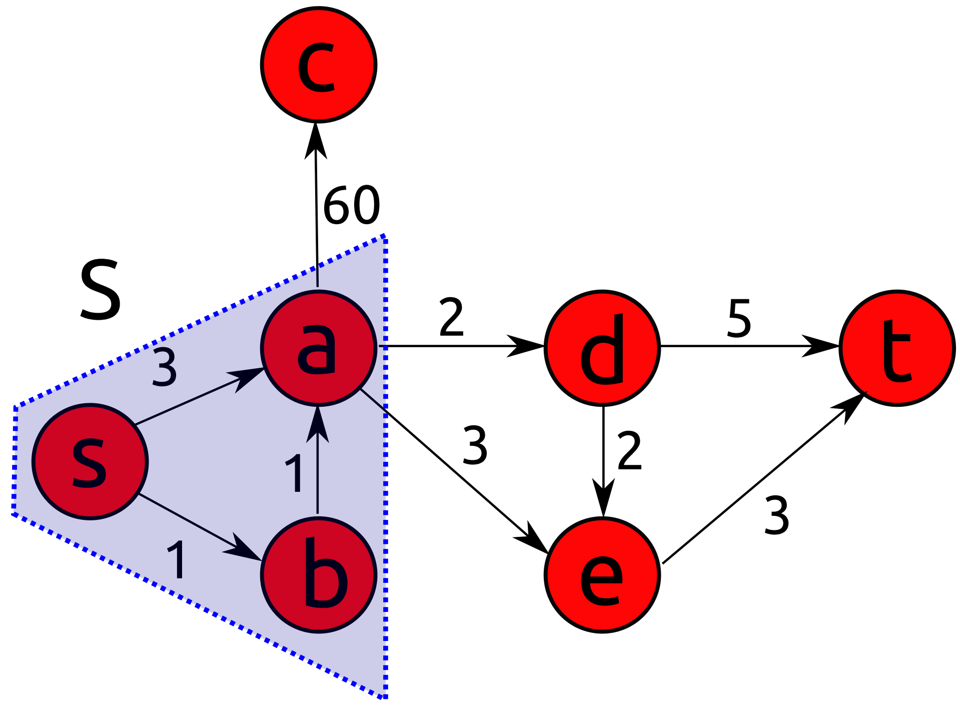
v	$Disc[v]$	$Dist[v]$	$P[v]$
s	True	0	[none]
a	False	2	b
b	True	1	s
c	False	∞	NULL
d	False	∞	NULL
e	False	∞	NULL
t	False	∞	NULL



Dijkstra's Algorithm

Example

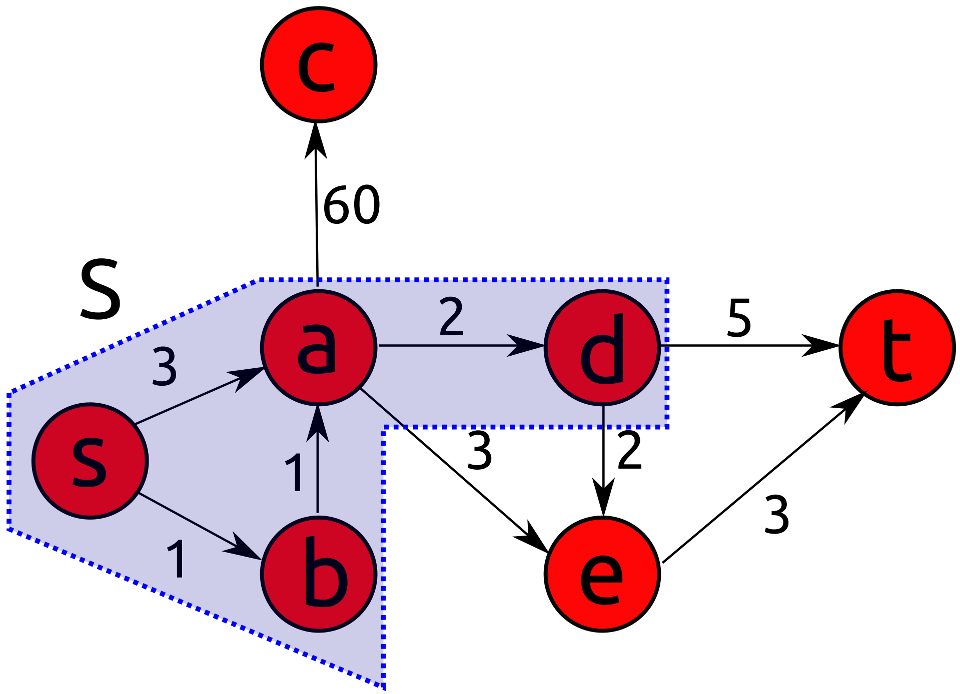
v	$Disc[v]$	$Dist[v]$	$P[v]$
s	True	0	[none]
a	True	2	b
b	True	1	s
c	False	62	a
d	False	4	a
e	False	5	a
t	False	∞	NULL



Dijkstra's Algorithm

Example

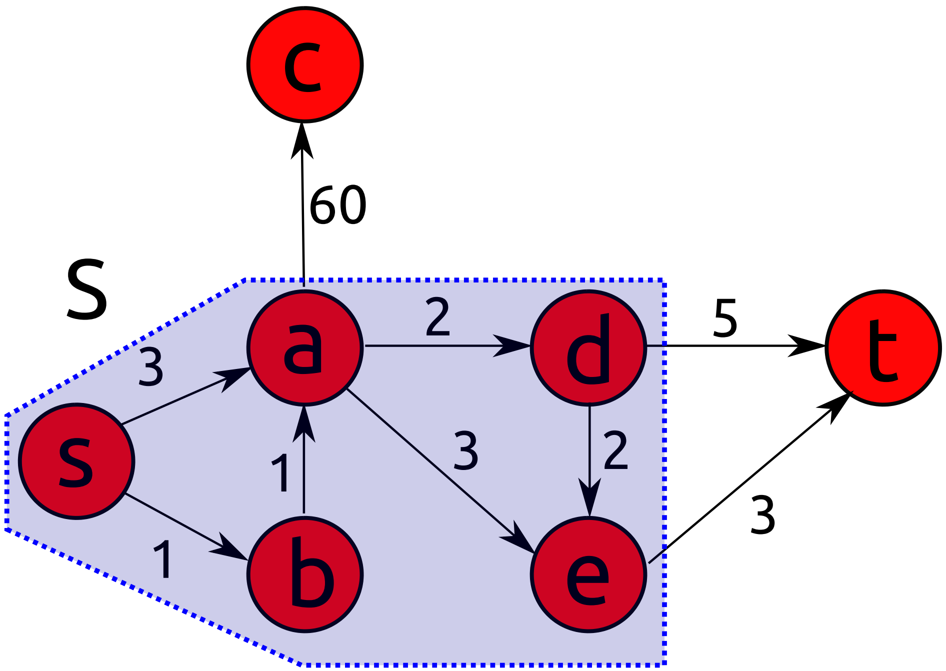
v	$Disc[v]$	$Dist[v]$	$P[v]$
s	True	0	[none]
a	True	2	b
b	True	1	s
c	False	62	a
d	True	4	a
e	False	5	a
t	False	9	d



Dijkstra's Algorithm

Example

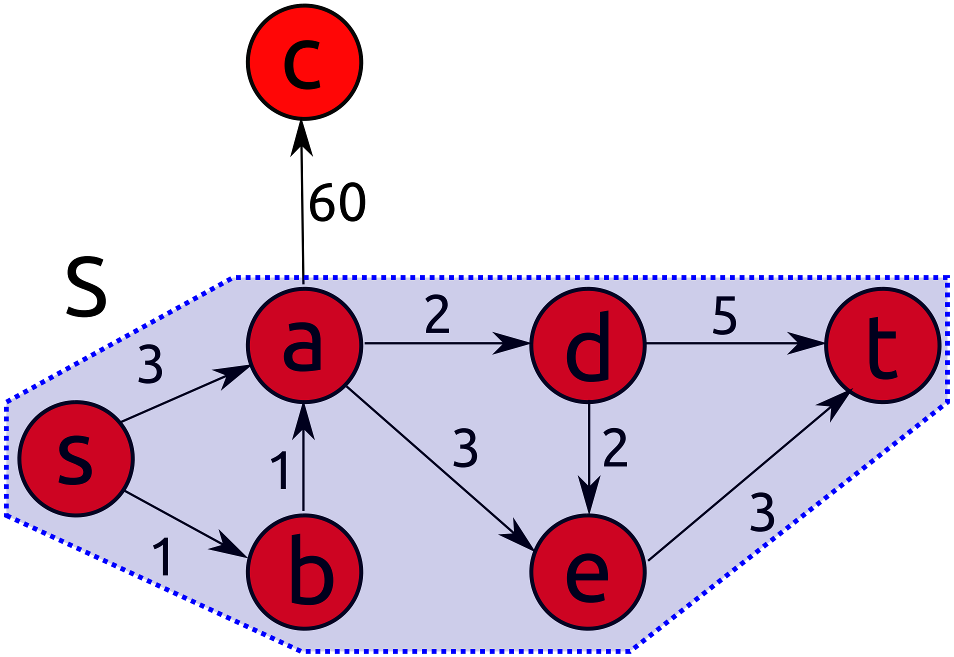
v	$Disc[v]$	$Dist[v]$	$P[v]$
s	True	0	[none]
a	True	2	b
b	True	1	s
c	False	62	a
d	True	4	a
e	True	5	a
t	False	8	e



Dijkstra's Algorithm

Example

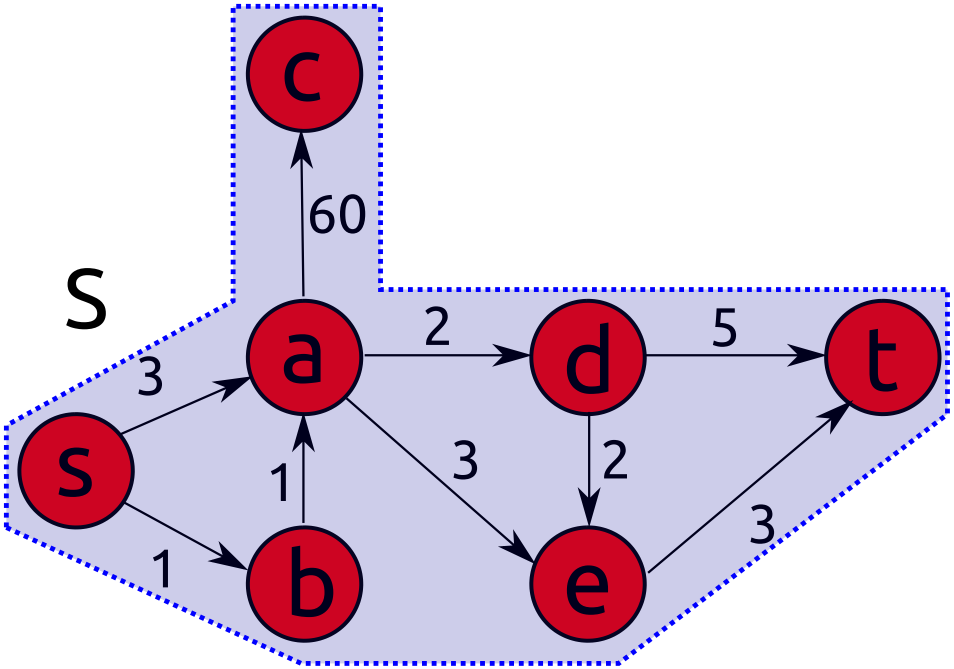
v	$Disc[v]$	$Dist[v]$	$P[v]$
s	True	0	[none]
a	True	2	b
b	True	1	s
c	False	62	a
d	True	4	a
e	True	5	a
t	True	8	e



Dijkstra's Algorithm

Example

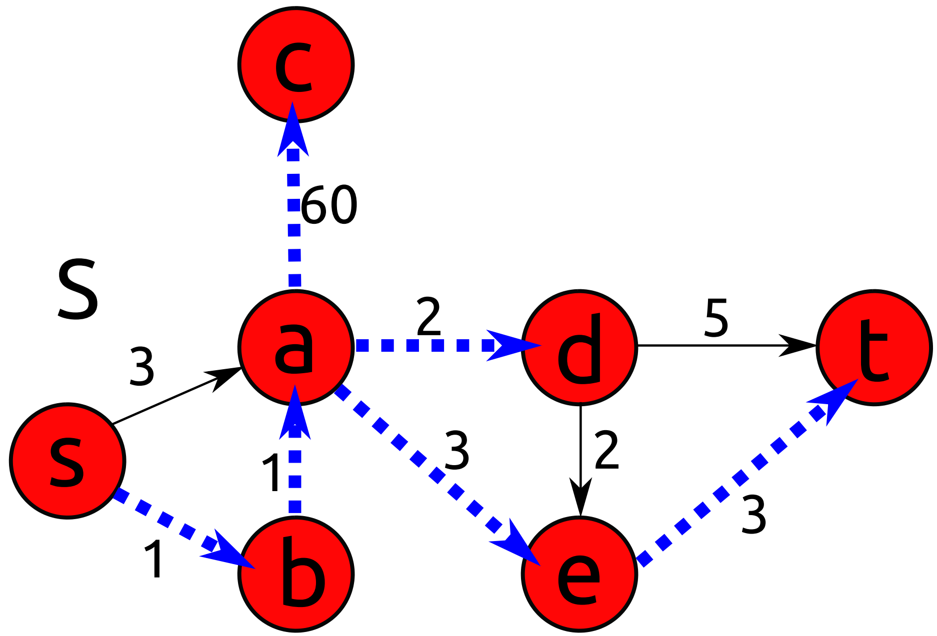
v	$Disc[v]$	$Dist[v]$	$P[v]$
s	True	0	[none]
a	True	2	b
b	True	1	s
c	True	62	a
d	True	4	a
e	True	5	a
t	True	8	e



Dijkstra's Algorithm

Example

v	$Disc[v]$	$Dist[v]$	$P[v]$
s	True	0	[none]
a	True	2	b
b	True	1	s
c	True	62	a
d	True	4	a
e	True	5	a
t	True	8	e



- Look at $P[v]$ to construct the shortest paths from s.

Dijkstra's Algorithm

Implementation and Analysis

Pseudocode Sketch

```
Init arrays Disc, Dist, and P, each of length |V|
For each v in V:
    Disc[v] = False
    Dist[v] = Infinity
    P[v] = NULL
Dist[s] = 0  // or else wouldn't know where to start
P[s] = None
Num_Discovered = 0
While Num_Discovered < |V|:
    Choose v that has smallest Dist[v] such that Disc[v] is False
    Disc[v] = True
    Num_Discovered = Num_Discovered + 1
    For each neighbor n of v:
        If Dist[v] + cost(v,n) < Dist[n]:
            Dist[n] = Dist[v] + cost(v,n)
            P[n] = v
```

Example

- In the previous example, when $b \in V$ is discovered, a is a neighbor of b. At this *exact* point, $Dist[a] = 3$, $P[a] = s$, and $Dist[b] = 1$. In the if statement, we compare $Dist[b] + cost(v, n) = 1 + 1 = 2$ to $Dist[a] = 3$ and conclude that $Dist[a]$ and $P[a]$ should be updated (to 2 and b, respectively).

Dijkstra's Algorithm

Implementation and Analysis

Pseudocode Sketch[1]

```
Init arrays Disc, Dist, and P, each of length |V|
For each v in V:
    Disc[v] = False
    Dist[v] = Infinity
    P[v] = NULL
Dist[s] = 0  // or else wouldn't know where to start
P[s] = None
Num_Discovered = 0
While Num_Discovered < |V|:
    min_dist = Infinity
    min_v = NULL
    For each v in V:
        If Disc[v] = False and Dist[v] < min_dist:
            min_dist = Dist[v]
            min_v = v
    Disc[v] = True
    Num_Discovered = Num_Discovered + 1
    For each neighbor n of v:
        If Dist[v] + cost(v,n) < Dist[n]:
            Dist[n] = Dist[v] + cost(v,n)
            P[n] = v
```

- Expand out the choosing of $v \in V - S$ (v is the next vertex to discover).

1. The naive Dijkstra's algorithm implementation in section 4.4 of the primary textbook differs in how it "phrases" the selection of the next vertex, somehow resulting in a different time complexity.

Dijkstra's Algorithm

Implementation and Analysis

Pseudocode Sketch

```
Init arrays Disc, Dist, and P, each of length |V|
For each v in V:
    Disc[v] = False
    Dist[v] = Infinity
    P[v] = NULL
Dist[s] = 0  // or else wouldn't know where to start
P[s] = None
Num_Discovered = 0
While Num_Discovered < |V|:
    min_dist = Infinity
    min_v = NULL
    For each v in V:
        If Disc[v] = False and Dist[v] < min_dist:
            min_dist = Dist[v]
            min_v = v
    Disc[v] = True
    Num_Discovered = Num_Discovered + 1
    For each neighbor n of v:
        If Dist[v] + cost(v,n) < Dist[n]:
            Dist[n] = Dist[v] + cost(v,n)
            P[n] = v
```

Time Complexity

- Each while loop iteration processes one node $\Rightarrow$ n iterations total.
- Selecting node with smallest $Dist[v]$ takes $\Theta(n)$ time (must check all vertices).
- For a selected vertex v, updating the $Dist[n]$ of all of the neighbors takes d_v time.
- **Total**: $\Theta(n^2 + m) = \Theta(n^2)$.

Dijkstra's Algorithm

Implementation and Analysis: Using a Priority Queue

Description

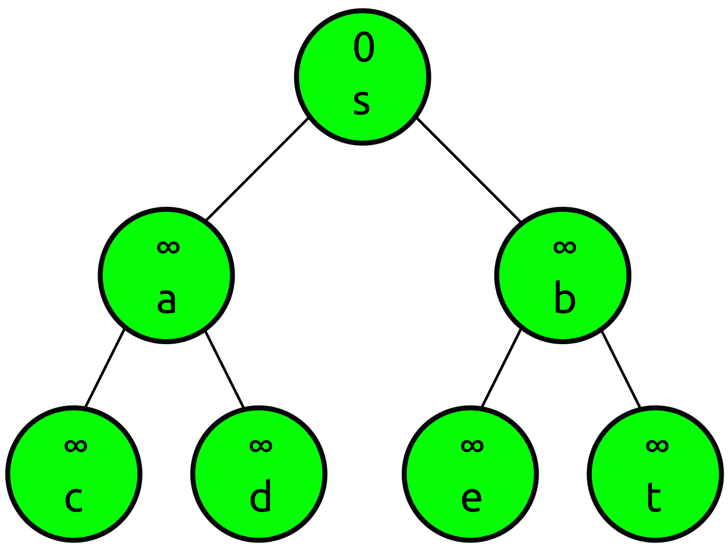
- Store each 2-tuple $(v, Dist[v])$ in priority queue B.
  - Use $Dist[v]$ as key.
  - No longer need $Disc[v]$.
- Must support $DecreaseKey$ operation.

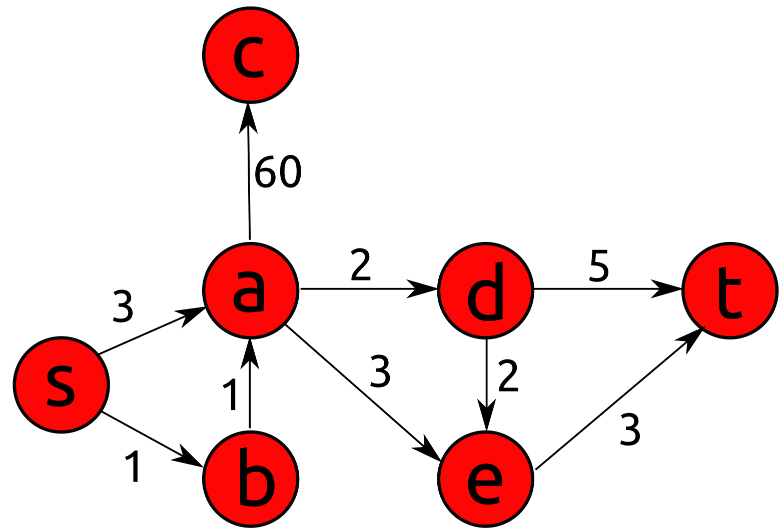
Pseudocode

```
Init arrays Dist and Parent
Init priority queue B
For each v in V:
    Dist[v] = Infinity
    Parent[v] = NULL
    Insert(B,v)
While |B| > 0:  // while priority queue not empty
    v = ExtractMin(B)
    For each neighbor n of v:
        new_cost = Dist[v] + cost(v,n)
        If new_cost < Dist[n]:
            DecreaseKey(B,n,Dist[n] - new_cost)
            Dist[n] = new_cost
            Parent[n] = v
```

Dijkstra's Algorithm

Example



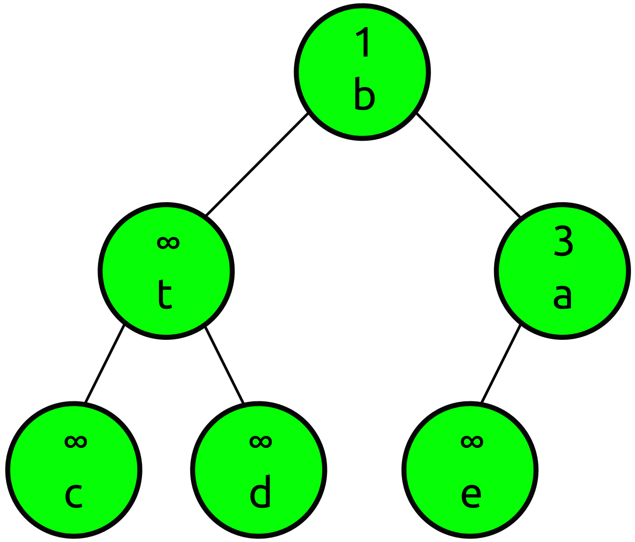


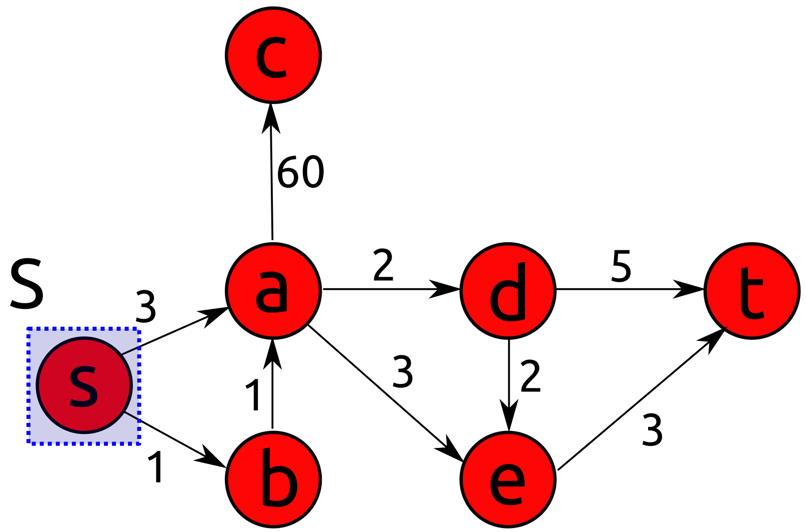
v	s	a	b	c	d	e	t
$Position[v]$	1	2	3	4	5	6	7

- Above is the $Position$ array contained within the priority queue, since the priority queue must know the position of each node in its internal binary heap in order to support the Extended API (i.e. $DecreaseKey$, etc.).

Dijkstra's Algorithm

Example

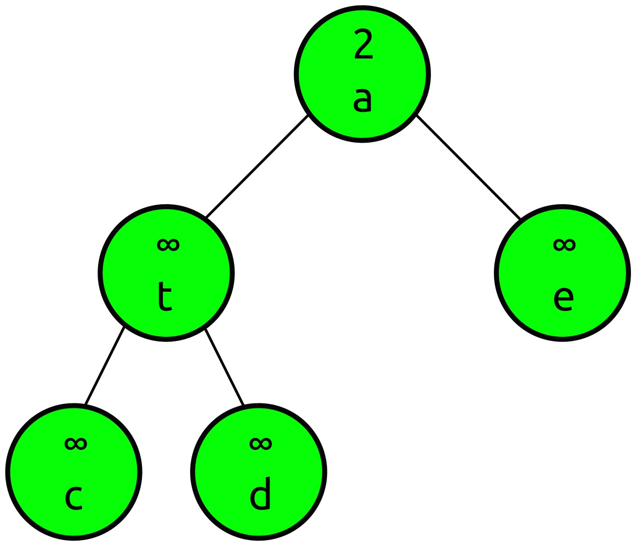


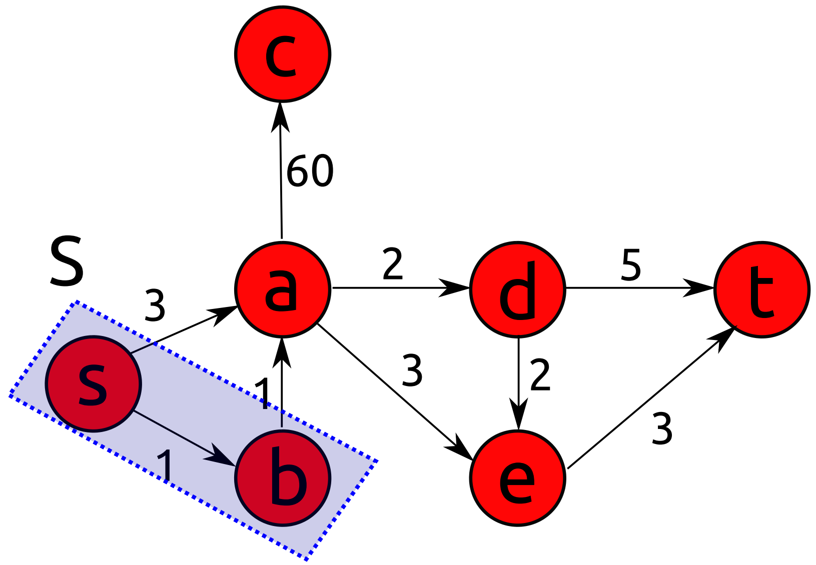


v	s	a	b	c	d	e	t
$Position[v]$	-1	3	1	4	5	6	2

Dijkstra's Algorithm

Example

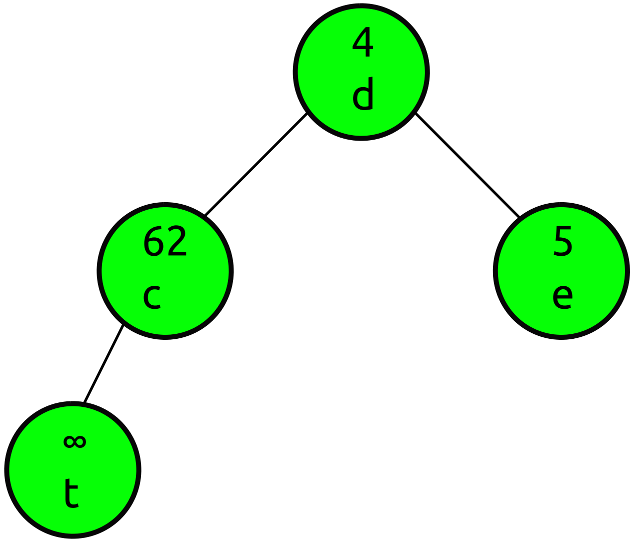


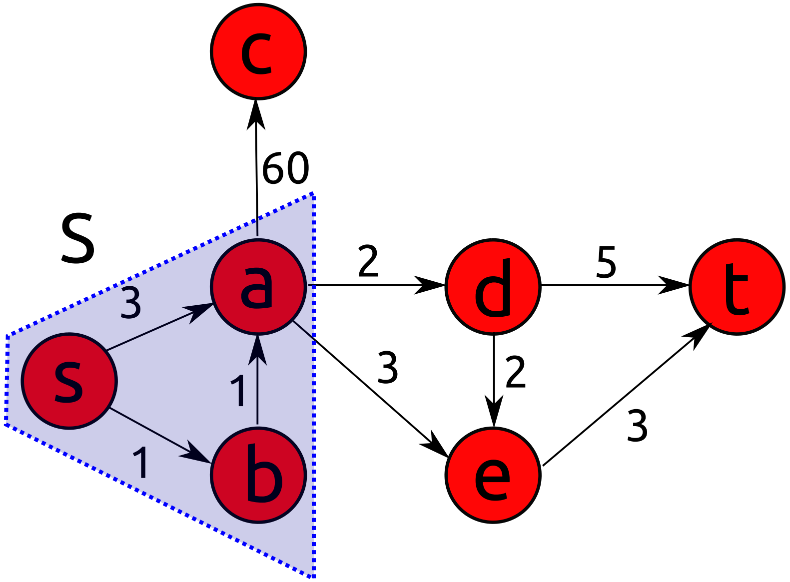


v	s	a	b	c	d	e	t
$Position[v]$	-1	1	-1	4	5	3	2

Dijkstra's Algorithm

Example

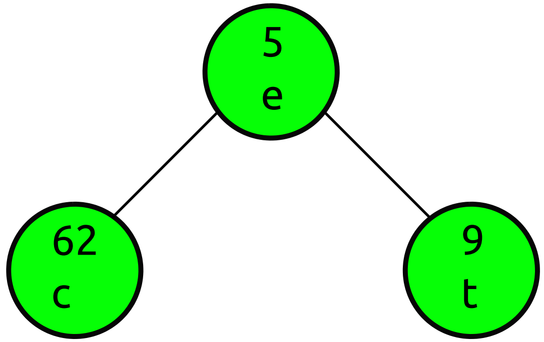


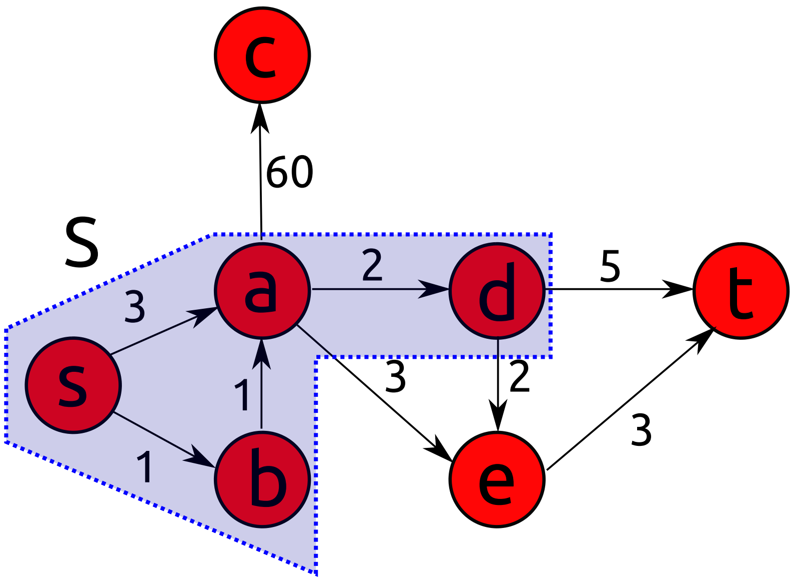


v	s	a	b	c	d	e	t
$Position[v]$	-1	-1	-1	2	1	3	4

Dijkstra's Algorithm

Example





v	s	a	b	c	d	e	t
$Position[v]$	-1	-1	-1	2	-1	1	3

- And so on...

Dijkstra's Algorithm

Implementation and Analysis

Pseudocode

```
Init arrays Dist and Parent
Init priority queue B
For each v in V:
    Dist[v] = Infinity
    Parent[v] = NULL
    Insert(B,v)
While |B| > 0:  // while priority queue not empty
    v = ExtractMin(B)
    For each neighbor n of v:
        new_cost = Dist[v] + cost(v,n)
        If new_cost < Dist[n]:
            DecreaseKey(B,n,Dist[n] - new_cost)
            Dist[n] = new_cost
            Parent[n] = v
```

Time Complexity

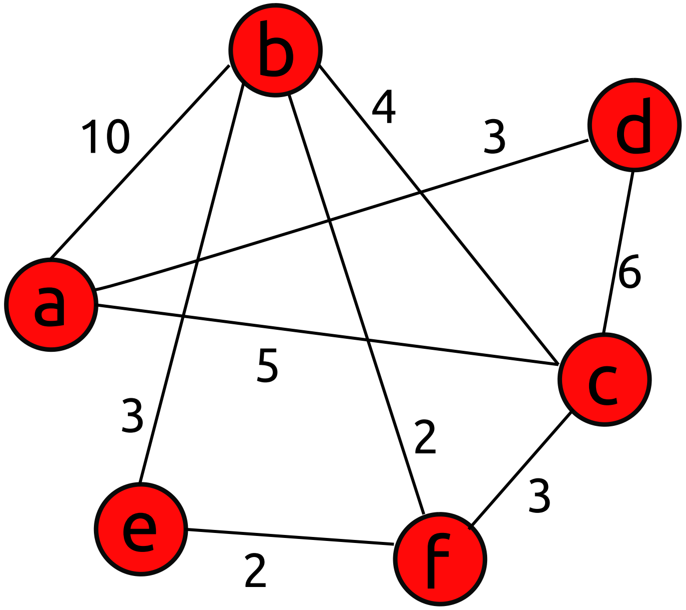
- Theorem 4.15 (from p.142 of primary textbook): $O(m)$ time plus the time for n $ExtractMin$ operations and m $DecreaseKey$ operations.
- With binary heap, each priority queue operation takes $O(\lg n)$ time.
- **Total**: $O(m + n\lg n + m\lg n) = O(m\lg n)$ time.
    - Note that $m \geq n$ (for any graph worth doing SSSP on).

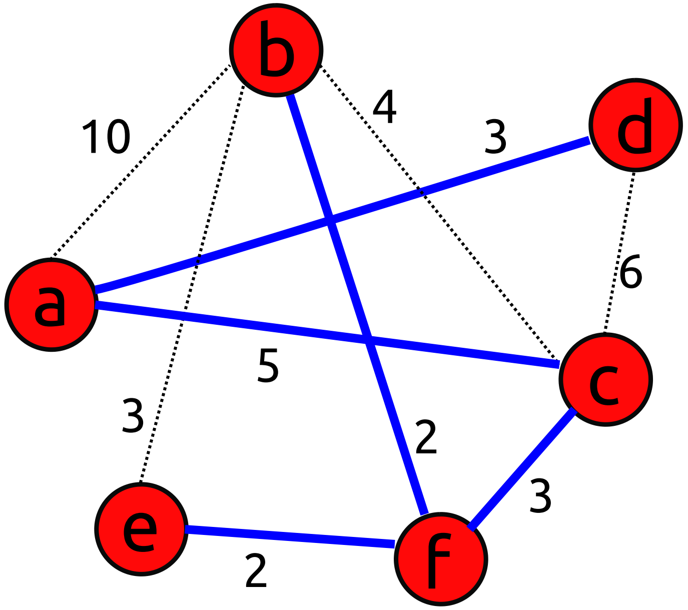
Minimum Spanning Tree (MST) Problem

Problem Description

- Input: undirected, connected graph $G = (V, E)$ where each $\{u, v\} \in E$ has positive cost/weight c_e
- Output: subset $T \subseteq E$ such that the graph (V, T) is connected and $\sum_{e \in T} c_e$ is *minimized*.
- Many applications: electrical grids, computer networks, transportation networks, etc.

Example



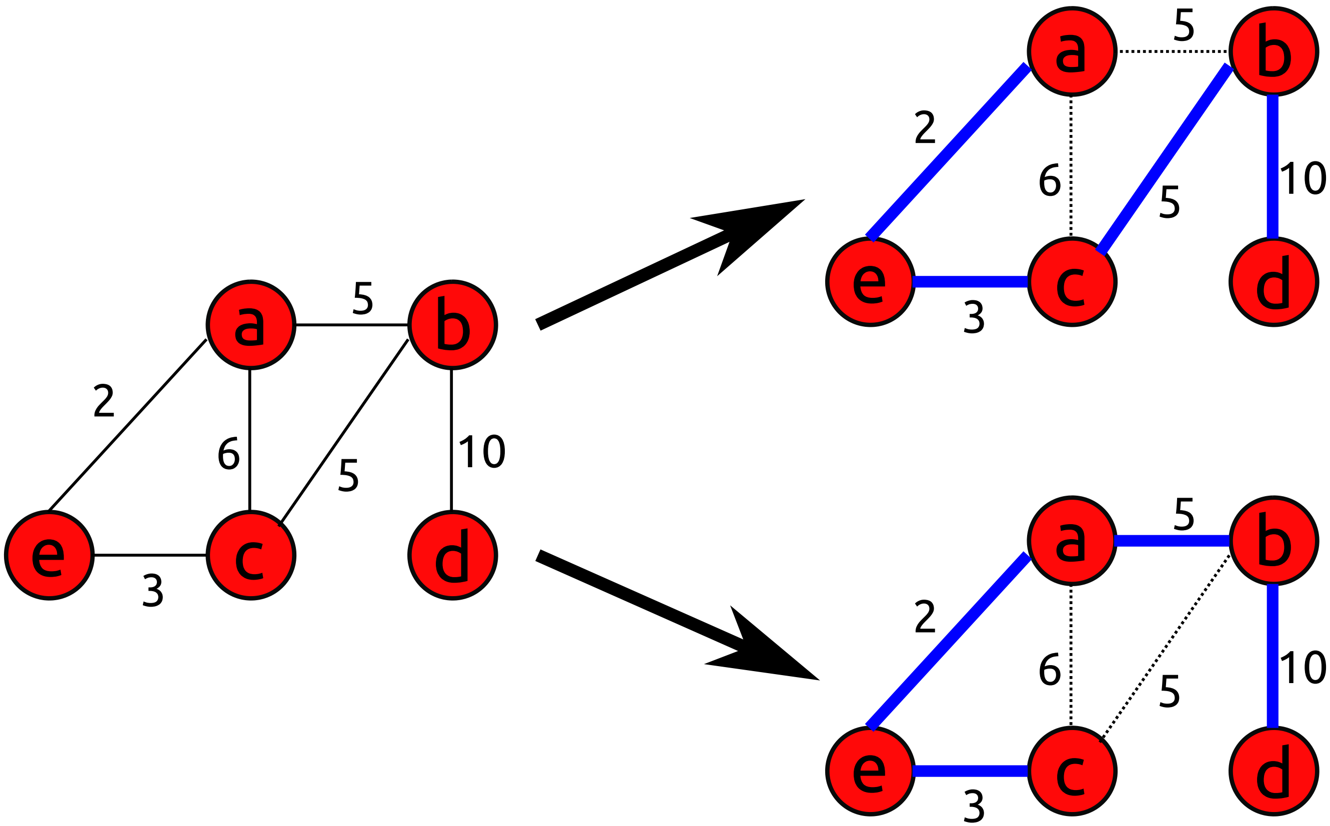


Minimum Spanning Tree (MST) Problem

Multiple Optimal Solutions

- Can be multiple minimum spanning trees.

Example



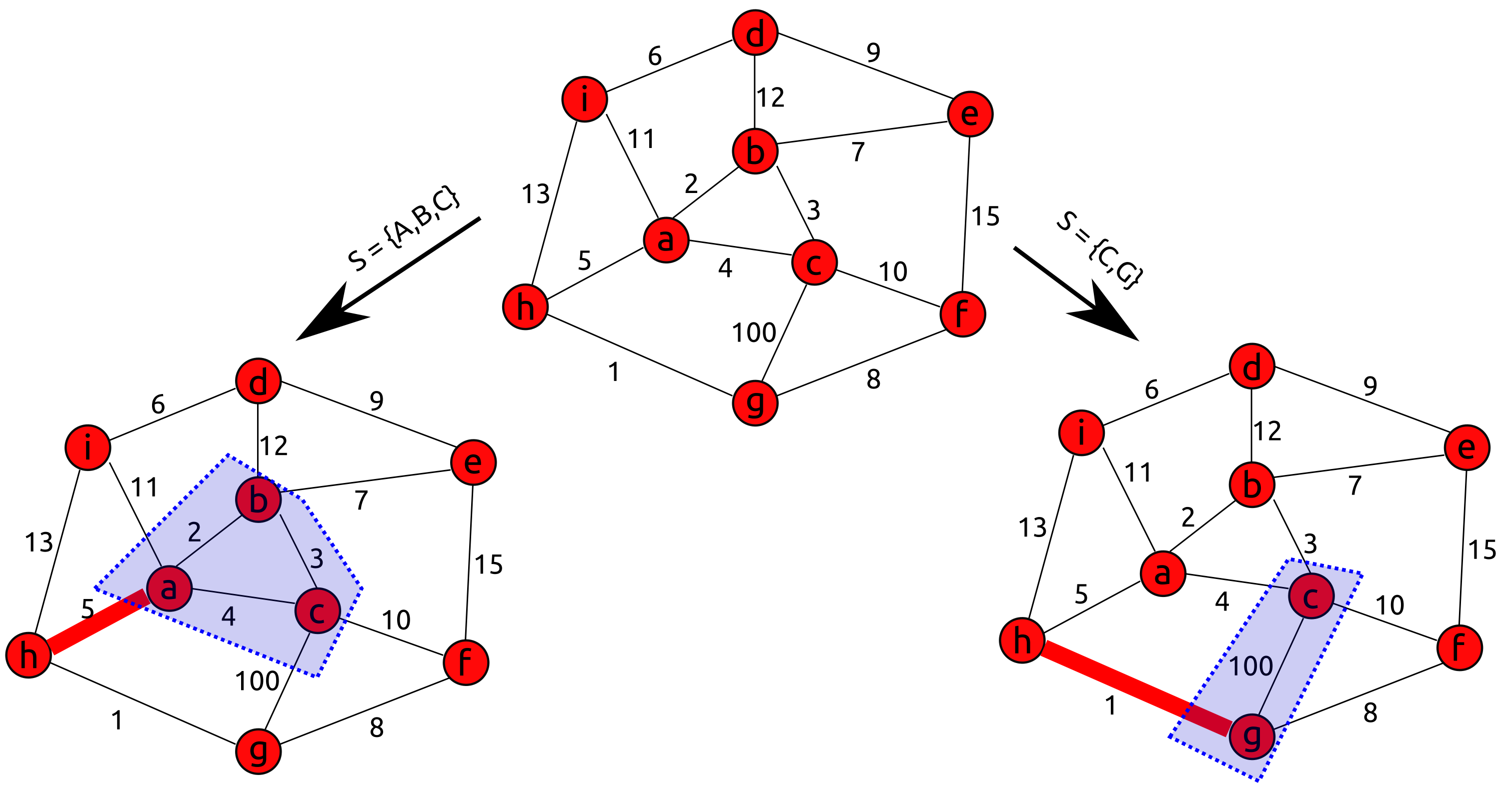
Minimum Spanning Tree (MST) Problem

Designing a Greedy Algorithm

- Choose one edge at a time. Which edge is safe to choose?

Cut Property

- Theorem 4.17: (Assumes distinct edge costs.) Let $S \subset V$ such that $|S| > 0$. Let edge $e = \{u, w\}$ be the minimum cost edge with one end in S and the other in $V - S$. *Every* minimum spanning tree contains the edge e.



Minimum Spanning Tree (MST) Problem

Designing a Greedy Algorithm

- Choose one edge at a time. Which edge is safe to choose?

Cut Property

- Theorem 4.17: (Assumes distinct edge costs.) Let $S \subset V$ such that $|S| > 0$. Let edge $e = \{u, w\}$ be the minimum cost edge with one end in S and the other in $V - S$. *Every* minimum spanning tree contains the edge e.

Important Notes and Consequences[1]

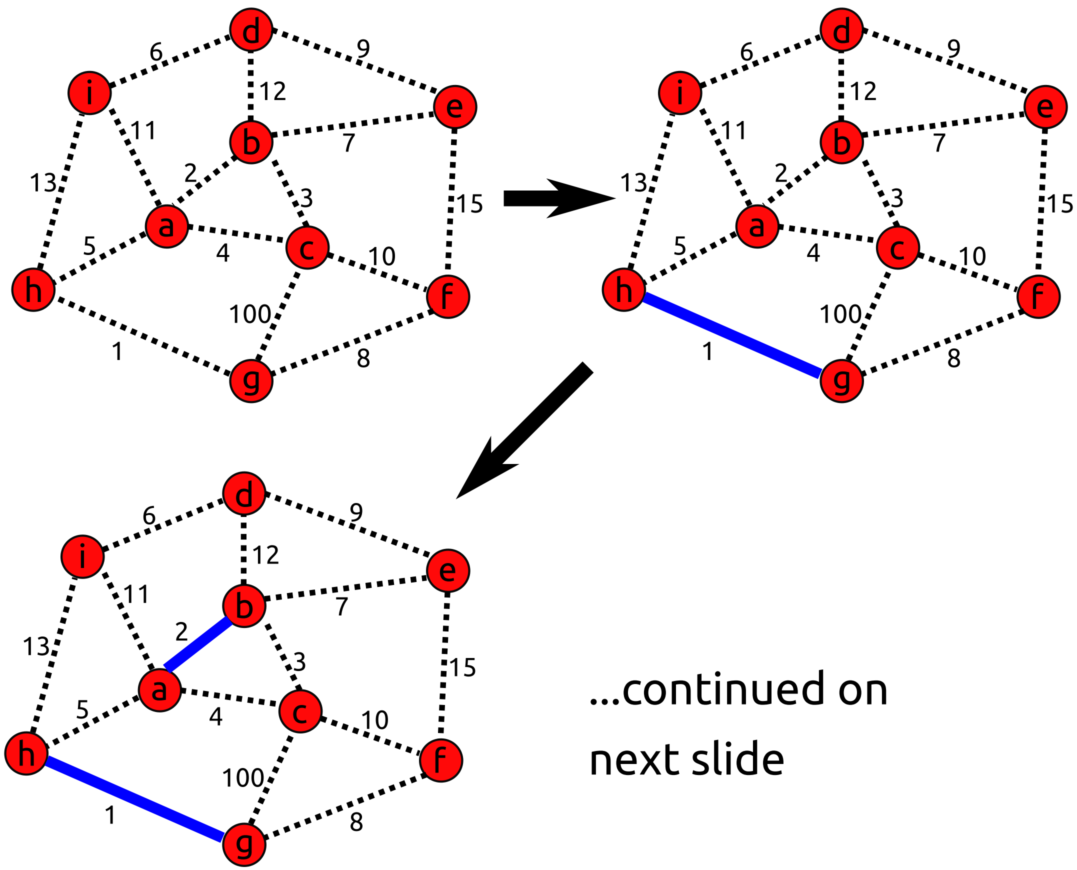
- Applies to *any* $S \subset V$ such that $|S| > 0$.
- For each $v \in V$, the cheapest edge adjacent to v is in the MST of G.

1. Again, this all assumes distinct edge costs.

Minimum Spanning Tree (MST) Problem

Kruskal's Algorithm

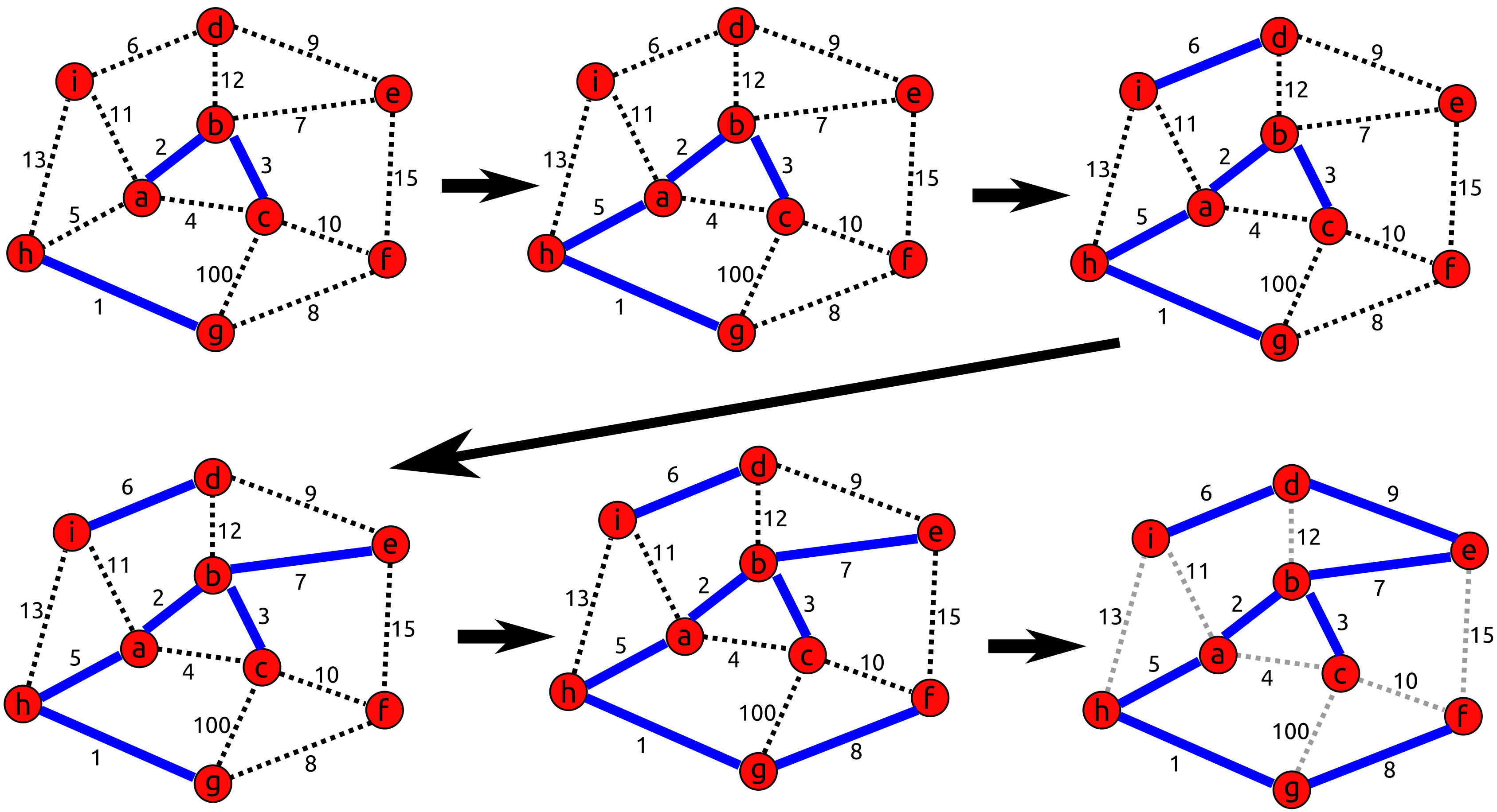
1. Start with $X = \emptyset$. (X will be the set of the MST's edges.)
2. Select $e \in E$ with smallest cost, such that $e \notin X$ and $T = (V, X \cup \{e\})$ is acyclic. Set $X = X \cup \{e\}$.
3. Repeat #2 until $T = (V, X)$ is a spanning tree.



...continued on next slide

Minimum Spanning Tree (MST) Problem

Kruskal's Algorithm: Example (Continued)



Minimum Spanning Tree (MST) Problem

Kruskal's Algorithm: Implementation

Concerns

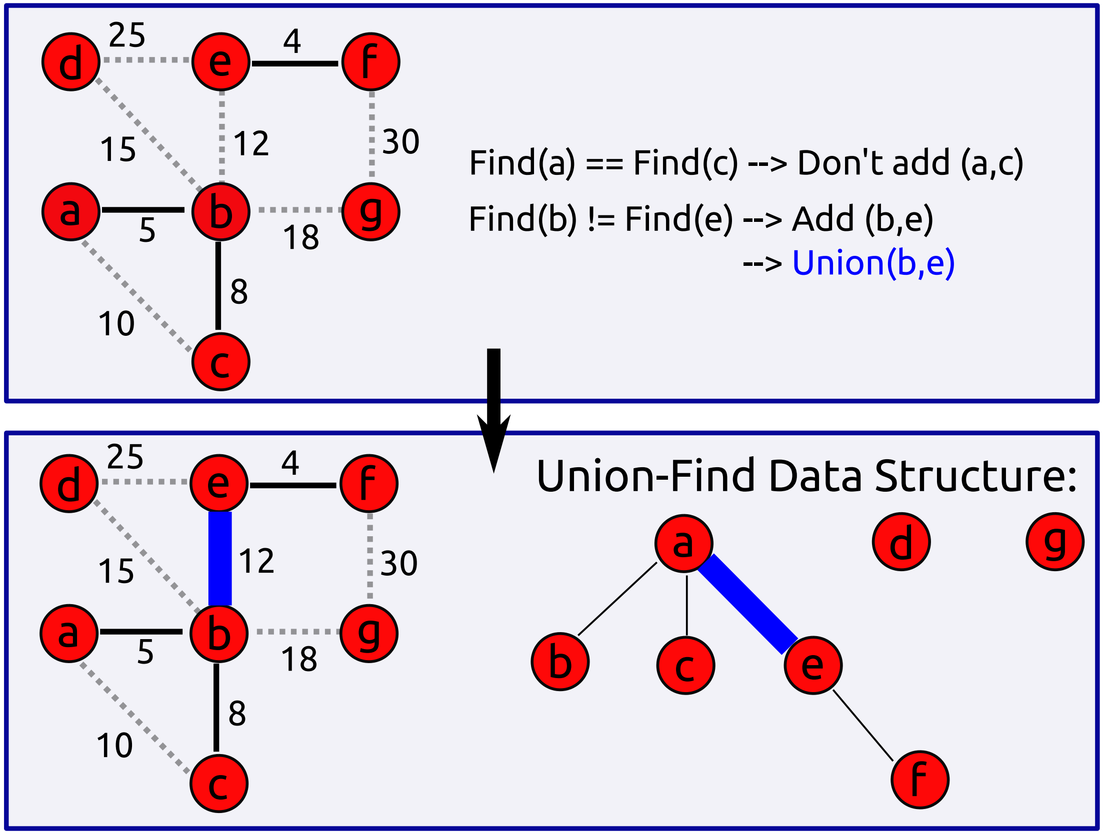
- When considering adding edge e:
  - Must *efficiently* (not naively) check that adding e won't create cycle.
- After adding edge e:
  - *Efficiently* determine the current connected components.

Minimum Spanning Tree (MST) Problem

Kruskal's Algorithm: Union-Find

Desired Operations

- Check that adding $e = \{u, v\}$ won't create cycle.
  - **Check if** $Find(u) == Find(v)$.
- Determine the current connected components.
  - **Update the "name" of each member after a union.**



Minimum Spanning Tree (MST) Problem

Kruskal's Algorithm: Union-Find

Worst-Case Time Complexity

- Must sort edges by weight.
  - $O(m \lg m) = O(m \lg n^2) = O(2m \lg n) = O(m \lg n)$ time. (This is a heavy "abuse of notation".)
- $2m\, Find$ operations, where $m = |E|$.
  - Check $2m$ edges.
  - Each check takes $O(\lg n)$ time with smart union.
- $n - 1\, Union$ operations, where $n = |V|$.
  - n vertices to add to the MST.
  - Each $Union$ operation takes $O(1)$ time.
    - Unnecessary to do two $Find$ operations in each $Union$ operation (in contrast to what footnote on Discussion 3, Slide 9).
- **Conclusion**: Worst-case time complexity is $O(m \lg n + m \lg n + n) = O(m \lg n)$.

Minimum Spanning Tree (MST) Problem

Kruskal's Algorithm: Union-Find

Side Note Regarding Union-Find Big-O

- With path compression[1], each $Find$ operation takes amortized $O(\alpha(n))$ time.
- "$\alpha(n)$, the inverse Ackermann function, has a value $\alpha(n) < 5$ for any value of n that can be written in this physical universe, so the disjoint-set operations take place in essentially constant time"[2].
  - i.e. $O(\alpha(n))$ time effectively means $O(1)$ time.
  - $2m\ Find$ operations $\Rightarrow$ amortized $O(m)$ time.
- Doesn't affect Kruskal's time complexity because of the initial sorting.

Method	**Expanded Runtime**	**Simplified**
Without path compression	$O(m \lg n + m \lg n + n)$	$O(m \lg n)$
With path compression	$O(m \lg n + m + n)$	$O(m \lg n)$

1. See Discussion 3, slide 12.
2. https://en.wikipedia.org/wiki/Disjoint-set_data_structure#Time_complexity

Minimum Spanning Tree (MST) Problem

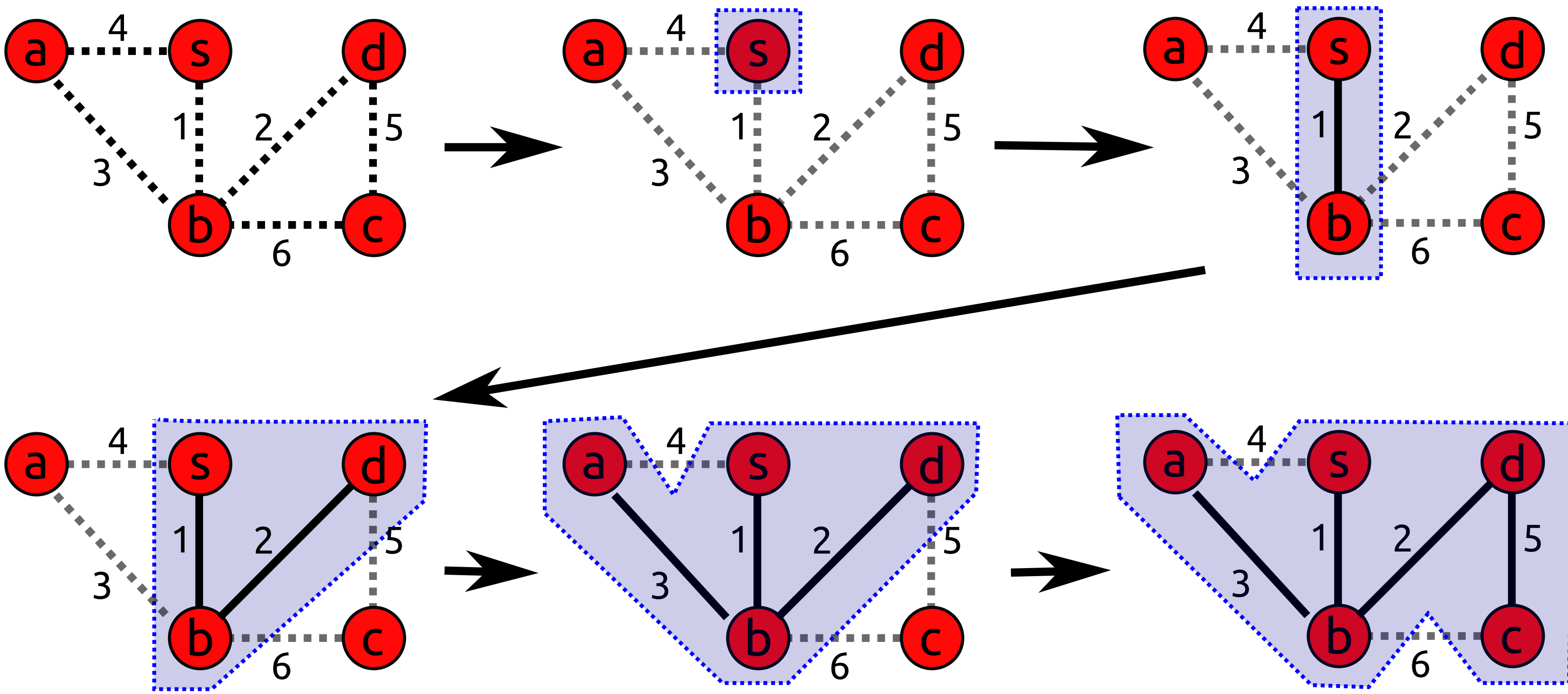
Prim's Algorithm

- **The idea**: start with one root node s and grow the tree outwards from there.
- Steps:
  1. Start with set $S = \{s\}$.
  2. Pick node $v \in V$ that minimizes "attachment cost" and doesn't create a cycle.
  3. Set $S = S \cup \{v\}$.
  4. Repeat #2 until $S = V$.
- Maintain set of edges in MST as well.

Minimum Spanning Tree (MST) Problem

Prim's Algorithm

Example



Minimum Spanning Tree (MST) Problem

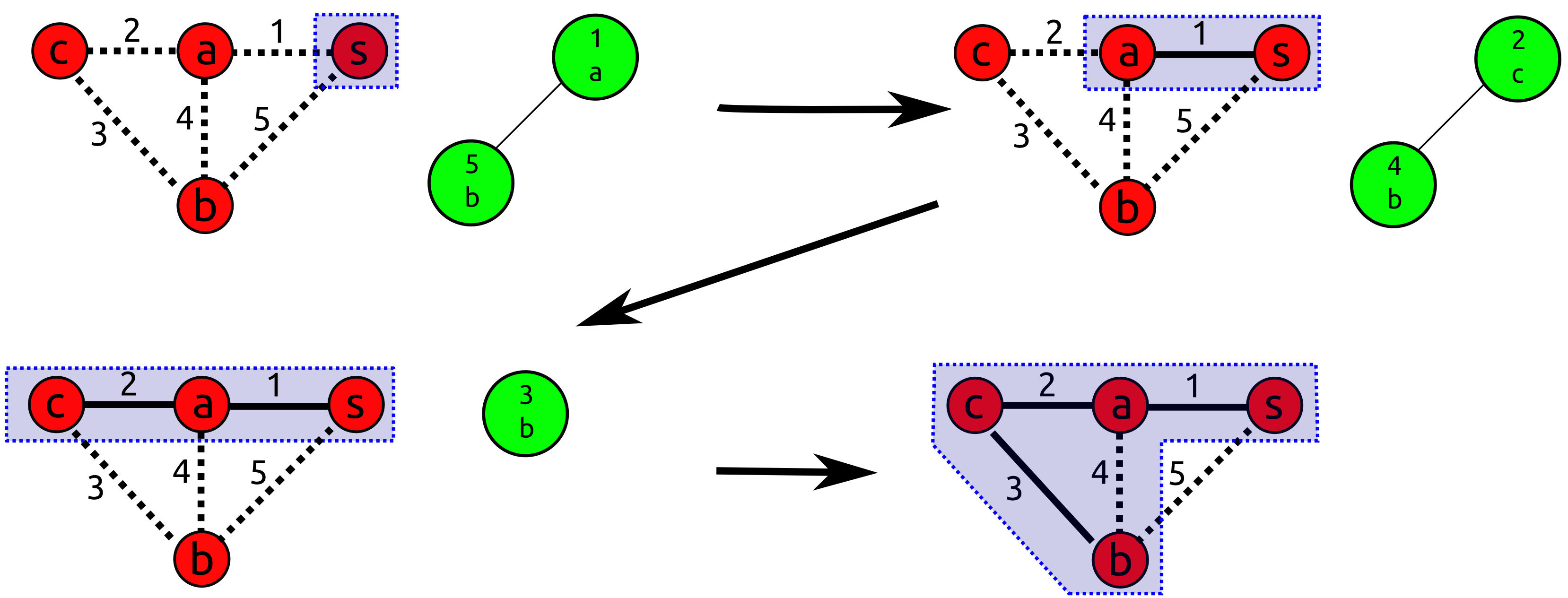
Prim's Algorithm: Implementation and Analysis

- Similar to Dijkstra's.
- Priority queue of nodes in $V - S$, where S is current set of nodes in T, the MST.
  - Use attachment costs $a(v)$ as keys.
  - Select node with $ExtractMin$ operation.
  - Update attachment cost with $DecreaseKey$ operation.
- $n - 1$ $ExtractMin$ operations, each $O(\lg n)$ time.
- m $DecreaseKey$ operations, each $O(\lg n)$ time.
- **Conclusion**: Runs in $O(n \lg n + m \lg n) = O(m \lg n)$ time.
  - Note that $m > n$ for any graph worth trying to find the MST for.

Minimum Spanning Tree (MST) Problem

Prim's Algorithm: Implementation and Analysis

Example

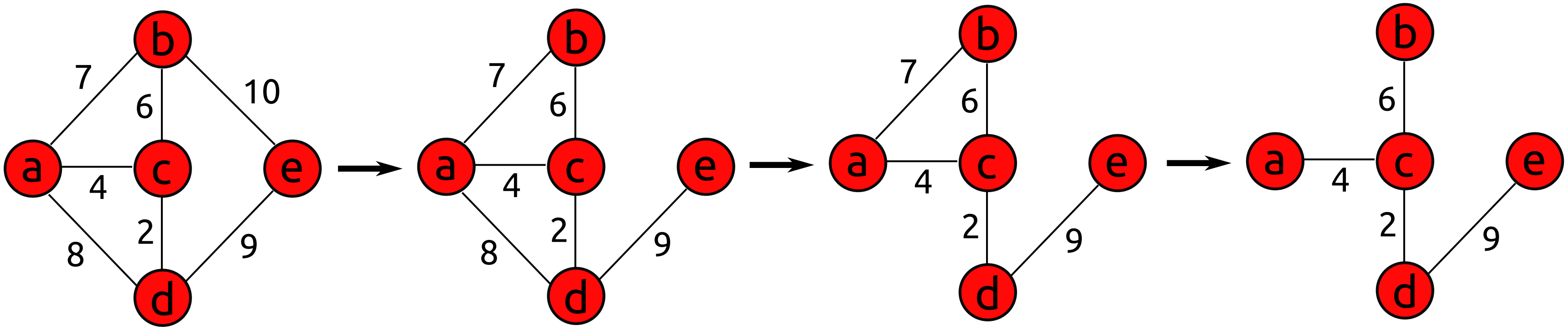


Minimum Spanning Tree (MST) Problem

Reverse-Delete Algorithm

- "Backward" version of Kruskal's algorithm.
- Steps:
  1. Start with full graph $G = (V, E)$.
  2. Pick most expensive edge $e \in E$ (that has not been picked yet).
  3. If deleting e would not disconnect G, delete e.
  4. Repeat #2 and #3 until reach least expensive edge.

Example



Minimum Spanning Tree (MST) Problem

Reverse-Delete Algorithm

Pseudocode/Implementation[1]

```
// Treats E as a list of edges.
// Assumes each v in V knows its neighbors.
// Uses one-based indexing.
ReverseDelete(V,E):
    Sort edges of E in order of decreasing cost.
    i = 1
    While i < |E|:
        If removing E[i] would disconnect G = (V,E):
            i = i + 1
        Else:
            Delete E[i]
    Return E  // the MST edges
```

Analysis

- Not bounded by initial sort.
- Can be optimized to run in $O(m \lg n (\lg \lg n)^3)$ time[2].

1. Based on: https://en.wikipedia.org/wiki/Reverse-delete_algorithm#Pseudocode

2. No, seriously.

Minimum Spanning Tree (MST) Problem

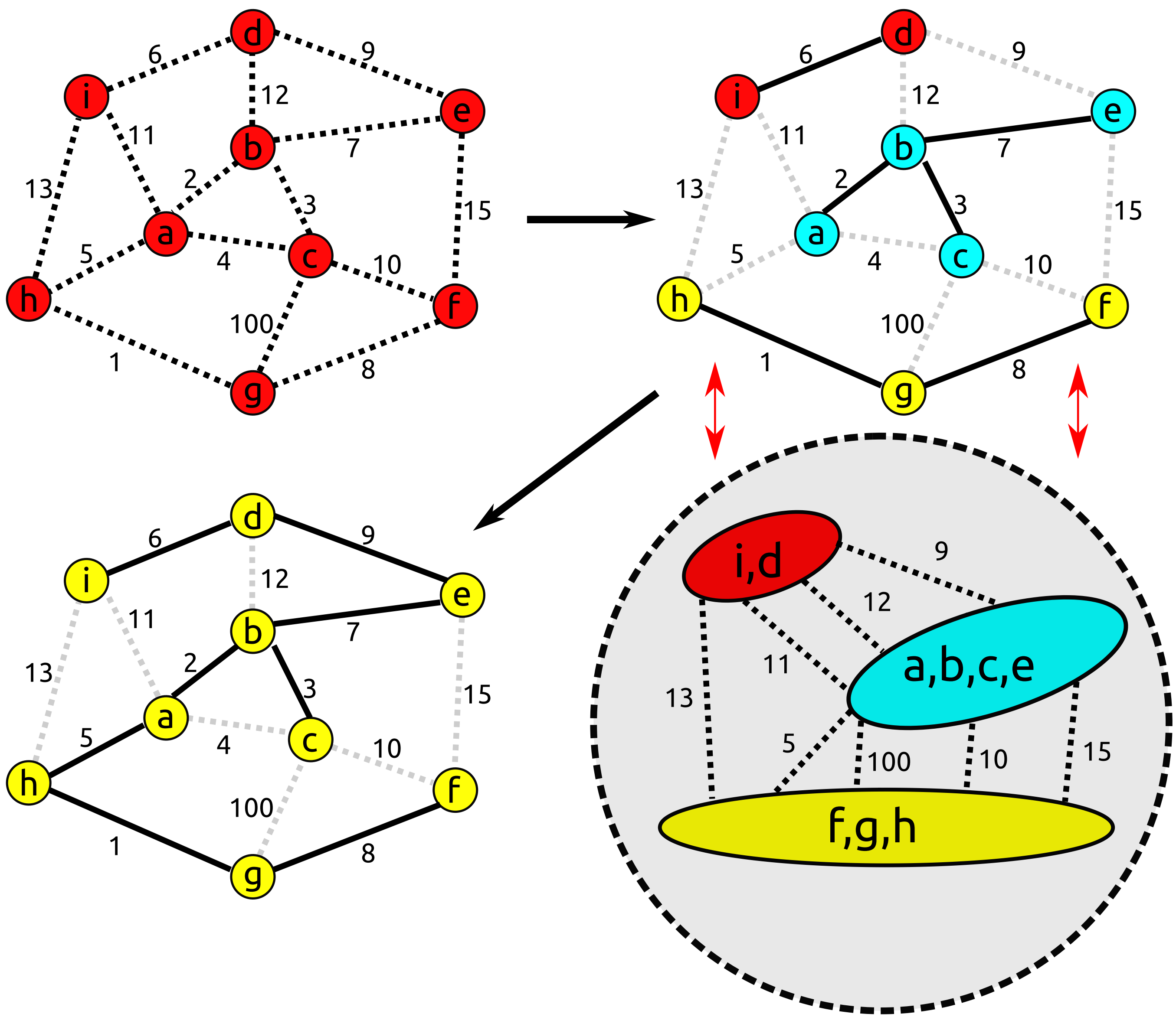
Borůvka's Algorithm (a.k.a. Sollin's Algorithm)

- Due to **cut property** (from earlier): For each $v \in V$, the cheapest edge adjacent to v is in the[1] MST of G.
- Steps:
  1. Initialize empty set of edges.
  2. Initialize each vertex to be its own connected component (because no edges) or "supervertex".
  3. Have each supervertex choose an edge to a neighboring supervertex of minimum cost.
  4. Determine the new connected components.
  5. Repeat #3 and #4 until is only one component/supervertex remaining.

1. Cut property assumes distinct edge costs.

Minimum Spanning Tree (MST) Problem

Borůvka's Algorithm



Minimum Spanning Tree (MST) Problem

Borůvka's Algorithm

Analysis

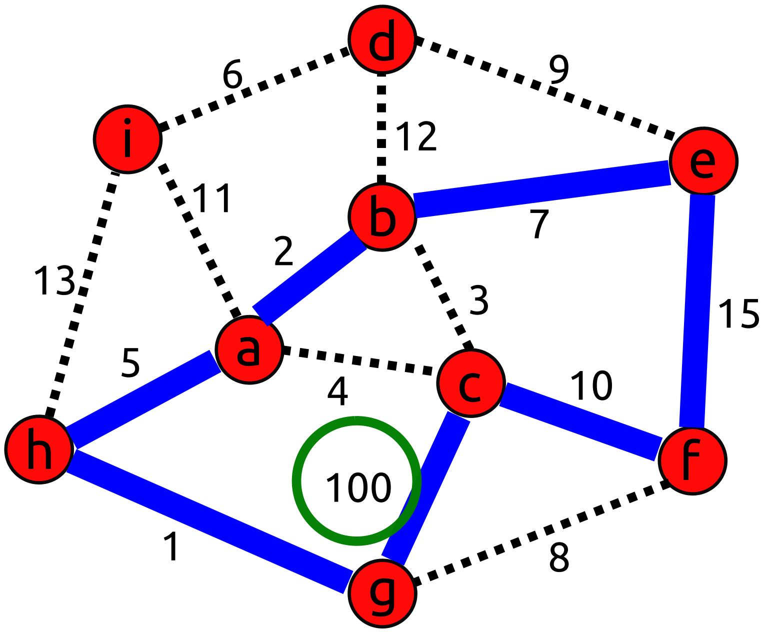
- $O(n + m + m) = O(n + m)$ per phase.
  - Finding the set of all connected components takes $\Theta(n + m)$ time.
    - See Lecture 3, Slide 66.
  - $\Theta(m)$ time to find cheapest edge of each supervertex, where $m = |E|$.
  - $O(n)$ time to check which supervertices to merge.
- If there are X supervertices, the next "phase" will have at most $\frac{X}{2}$ supervertices.
  - $O(\lg n)$ phases.
- **Conclusion**: Runs in $O((n + m) \lg n) = O(m \lg n)$ time.
  - Again, highly likely that $m > n$.

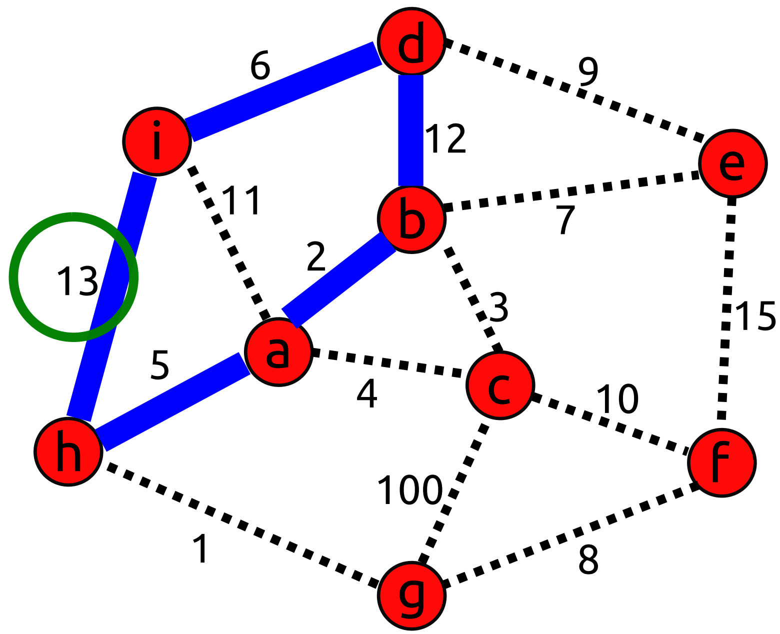
Minimum Spanning Tree (MST) Problem

Other Properties

Cycle Property

- For certain edges, can verify that they are not in any MST of G.
- Theorem 4.20: (Assumes distinct edge costs.) Let C be any cycle in G, and let $e \in E$ be the most expensive edge in C. e cannot belong to any MST of G.



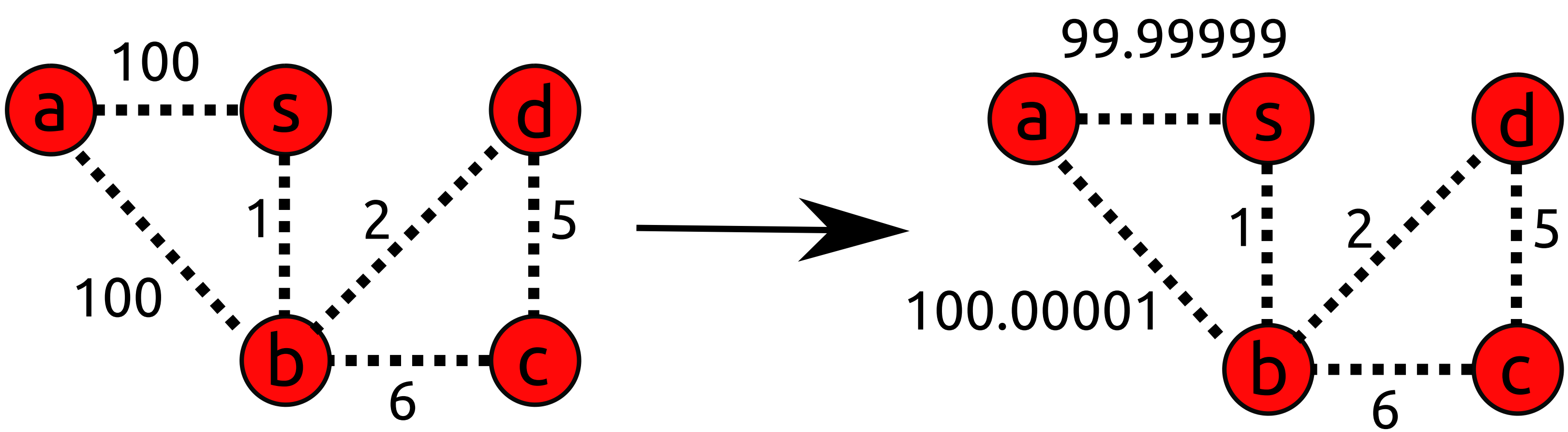


Minimum Spanning Tree (MST) Problem

Other Properties

Regarding Distinct Edge Costs

- If G has at least two minimum spanning trees, then G has at least two edges with the same costs.
  - Not going to prove here.
- For graphs that don't have distinct edge costs, can "perturb all edge costs by different, extremely small numbers, so that [the edge costs] all become distinct."[1]



1. p.149 of Kleinberg/Tardos textbook.

References / Further Reading

- Primary textbook (the "black book") referenced in these slides: *Algorithm Design* by Jon Kleinberg and Éva Tardos.