CSCI 3230 Data Structures

Graph

Weitian Tong, Ph.D.

Department of Computer Science
Georgia Southern University
Website: www.weitiantong.com
Email:wtong@georgiasouthern.edu

Table of contents

Graph

Graph

Introduction

Graph

A graph G is a pair (V, E), where

- V is a set of nodes, called vertices
- E is a collection of pairs of vertices, called edges

Note that vertices and edges are positions and store elements

Graph

A graph G is a pair (V, E), where

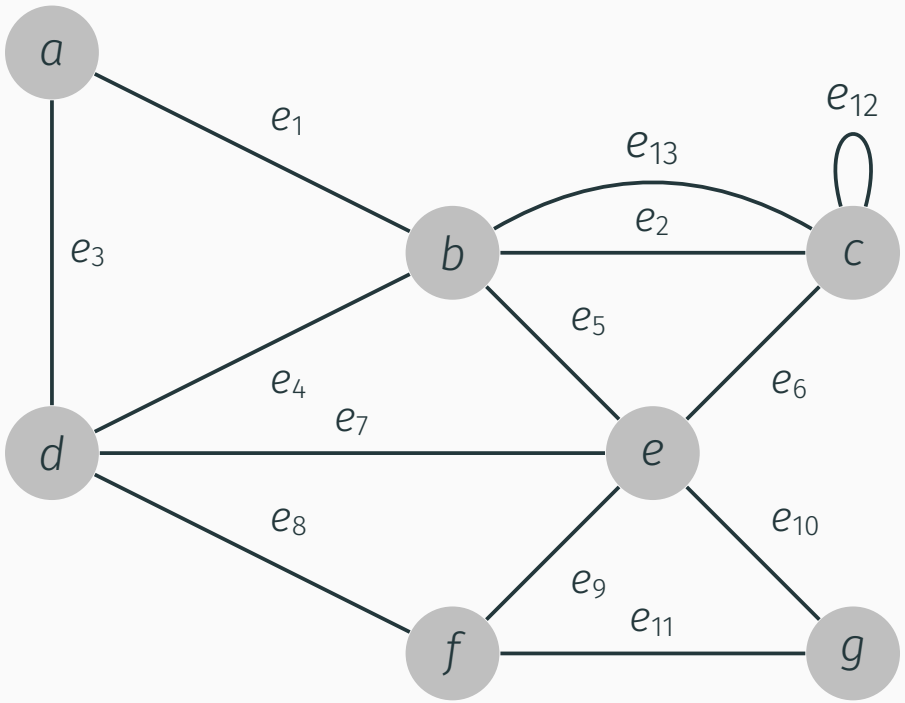
- V is a set of nodes, called vertices
- E is a collection of pairs of vertices, called edges

Note that vertices and edges are positions and store elements

- **Directed edge**: ordered pair of vertices (u, v)
- **Undirected edge**: unordered pair of vertices (u, v)
- **Directed graph (Digraph)**: all the edges are directed
- **Undirected graph**: all the edges are undirected

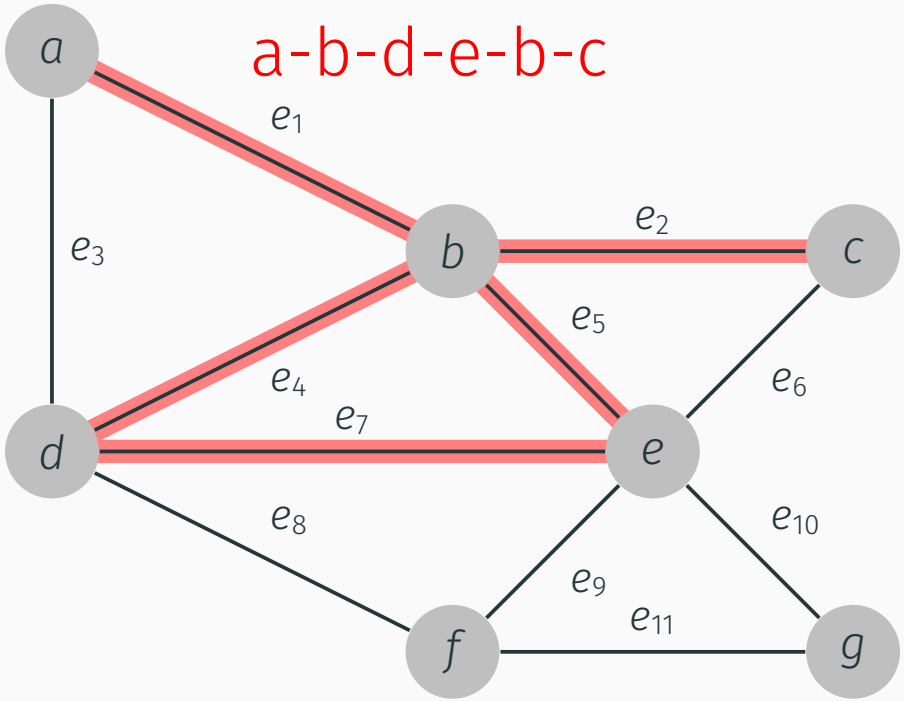
Basic Concepts

Endpoints of an edge: a and a are the endpoints of e_1
Edges incident on a vertex: e_1 and e_3 are incident on a
Adjacent vertices: a and b are adjacent
Degree of a vertex: d has degree 4
Parallel edges: e_2 and e_{13} are parallel edges
Self-loop: e_{12} is a self-loop



More Basic Concepts: Paths and Cycles

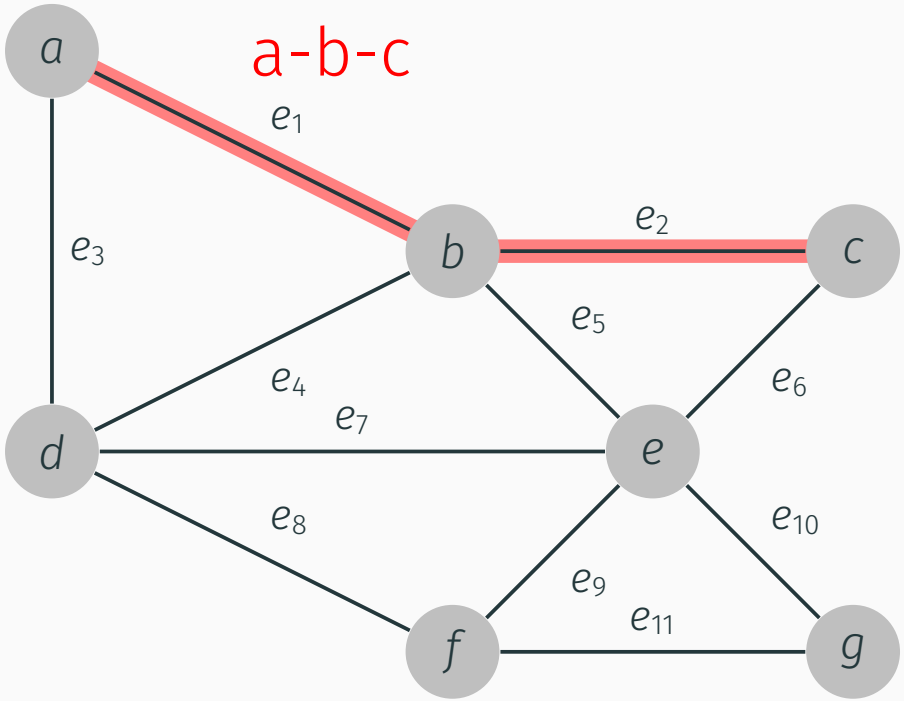
Path: a sequence of alternating vertices and edges, begins with a vertex, ends with a vertex.



More Basic Concepts: Paths and Cycles

Path: a sequence of alternating vertices and edges, begins with a vertex, ends with a vertex.

Simple path: path such that all its vertices and edges are distinct

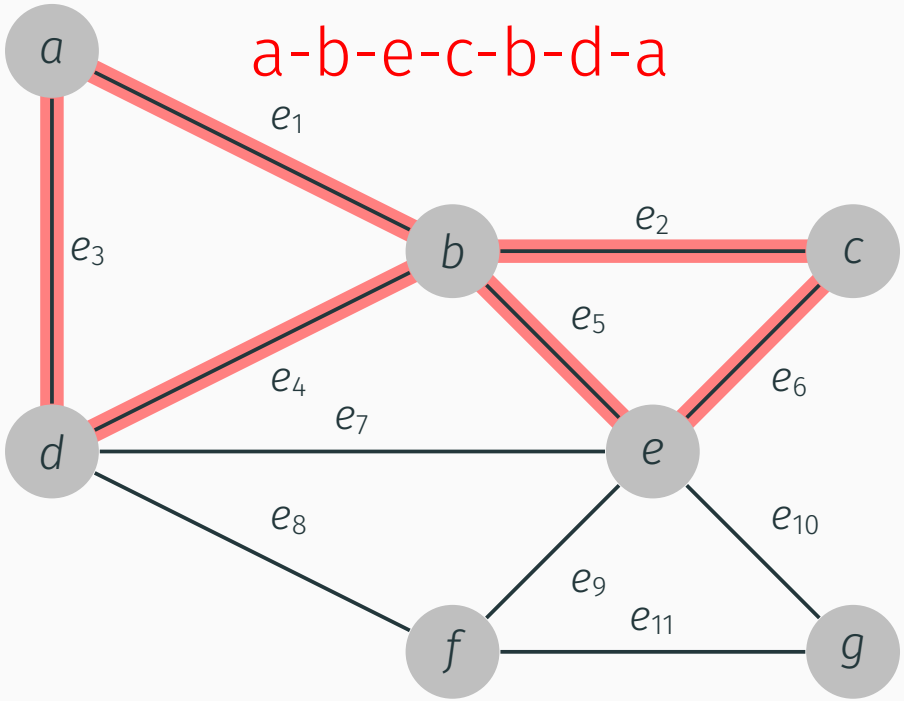


More Basic Concepts: Paths and Cycles

Path: a sequence of alternating vertices and edges, begins with a vertex, ends with a vertex.

Simple path: path such that all its vertices and edges are distinct

Cycle: circular sequence of alternating vertices and edges



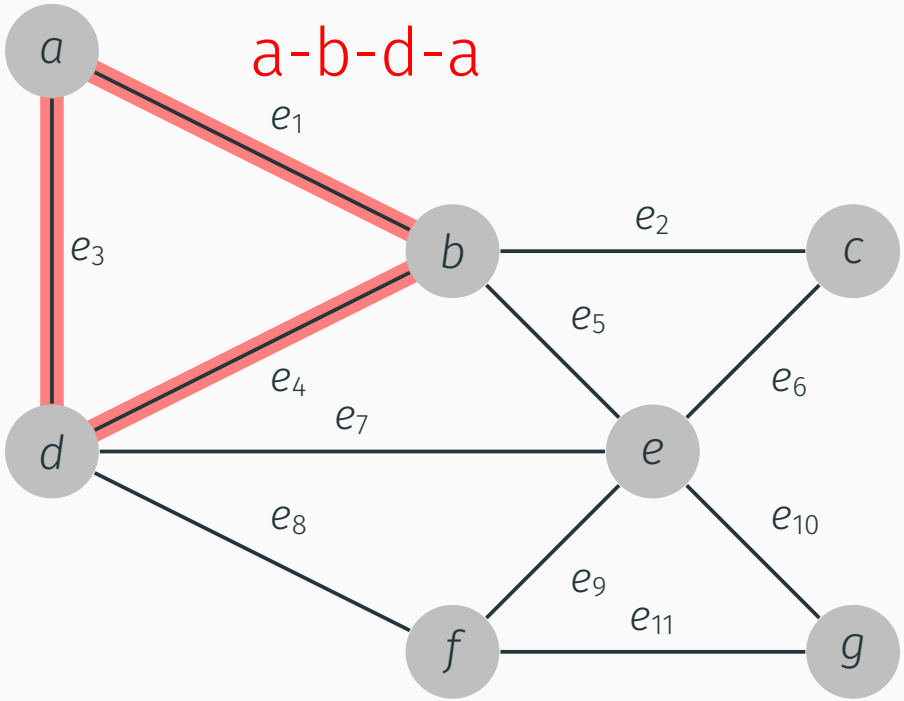
More Basic Concepts: Paths and Cycles

Path: a sequence of alternating vertices and edges, begins with a vertex, ends with a vertex.

Simple path: path such that all its vertices and edges are distinct

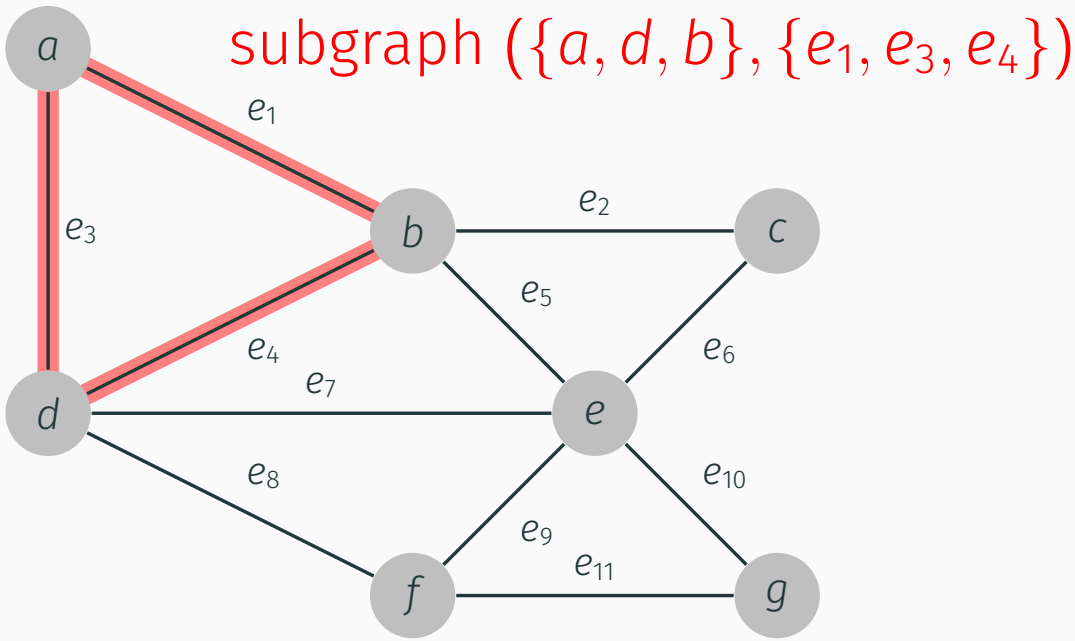
Cycle: circular sequence of alternating vertices and edges

Simple cycle: cycle such that all its vertices and edges are distinct



More Basic Concepts: Subgraphs and Trees

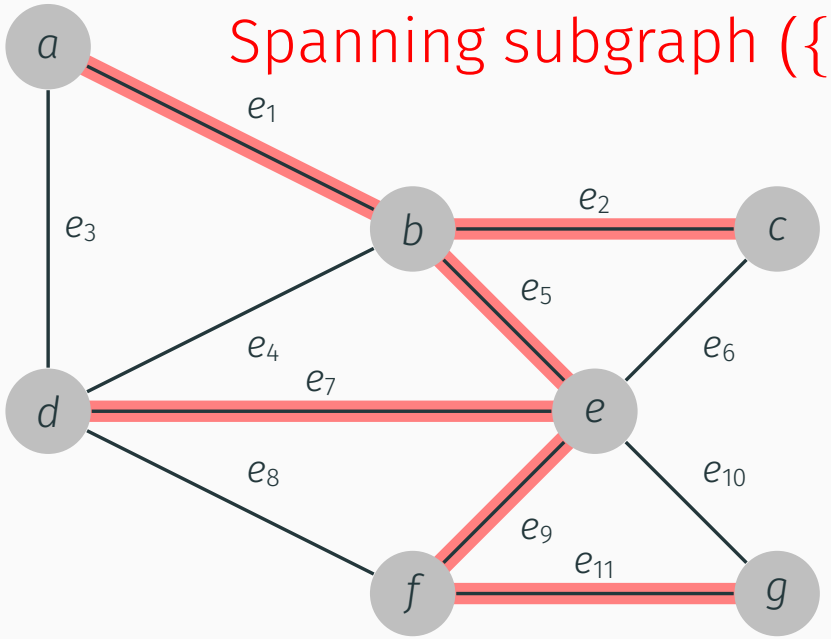
Subgraph $S = (V_S, E_S)$: $V_S \subset V$, and $E_S \subset E$



More Basic Concepts: Subgraphs and Trees

Subgraph $S = (V_S, E_S)$: $V_S \subset V$, and $E_S \subset E$

Spanning subgraph $S = (V_S, E_S)$: $V_S = V$, and $E_S \subset E$



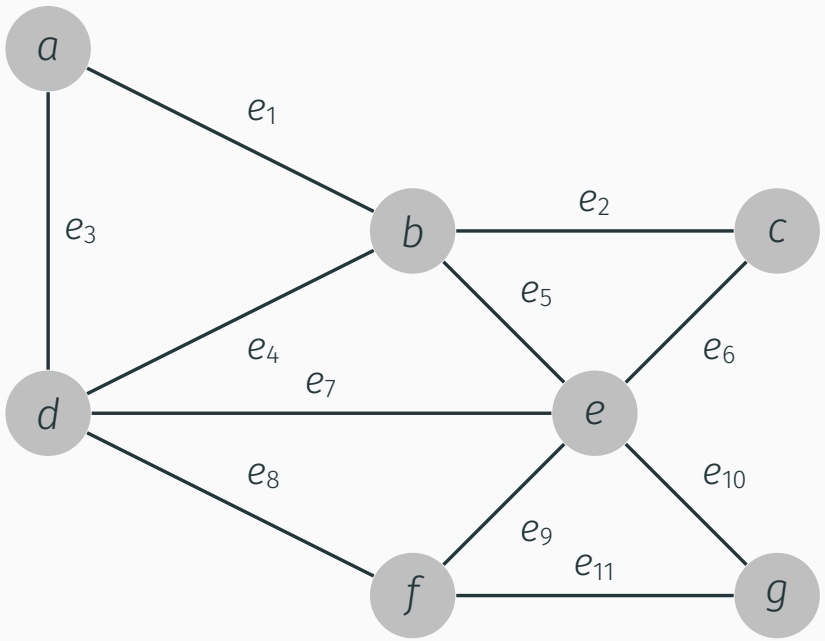
Spanning subgraph $(\{a, b, c, d, e, f, g\}, \{\text{red edges}\})$

More Basic Concepts: Subgraphs and Trees

Subgraph $S = (V_S, E_S)$: $V_S \subset V$, and $E_S \subset E$

Spanning subgraph $S = (V_S, E_S)$: $V_S = V$, and $E_S \subset E$

Connected component: a maximal connected subgraph of G

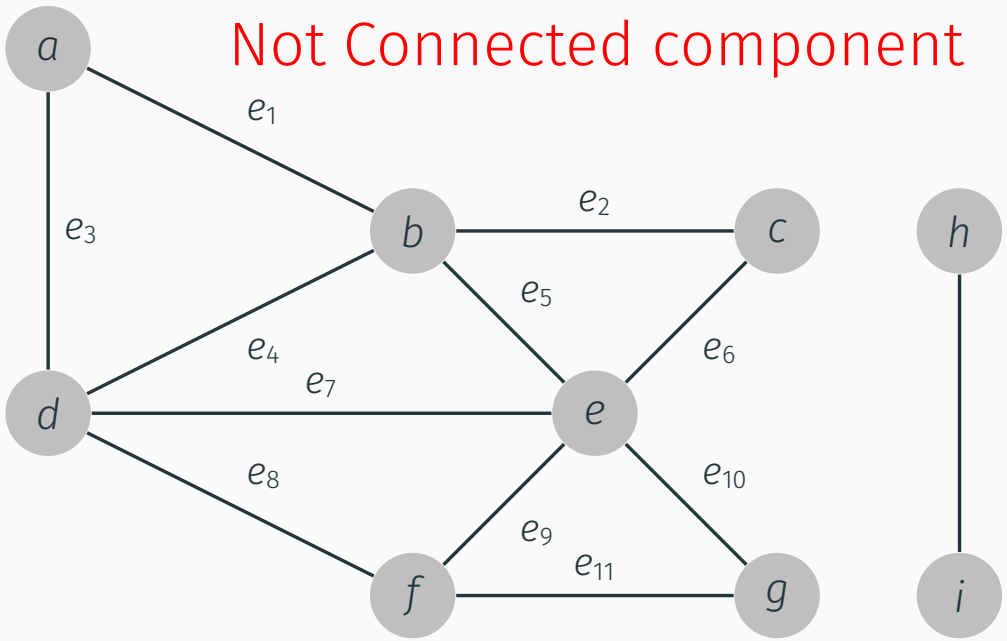


More Basic Concepts: Subgraphs and Trees

Subgraph $S = (V_S, E_S)$: $V_S \subset V$, and $E_S \subset E$

Spanning subgraph $S = (V_S, E_S)$: $V_S = V$, and $E_S \subset E$

Connected component: a maximal connected subgraph of G

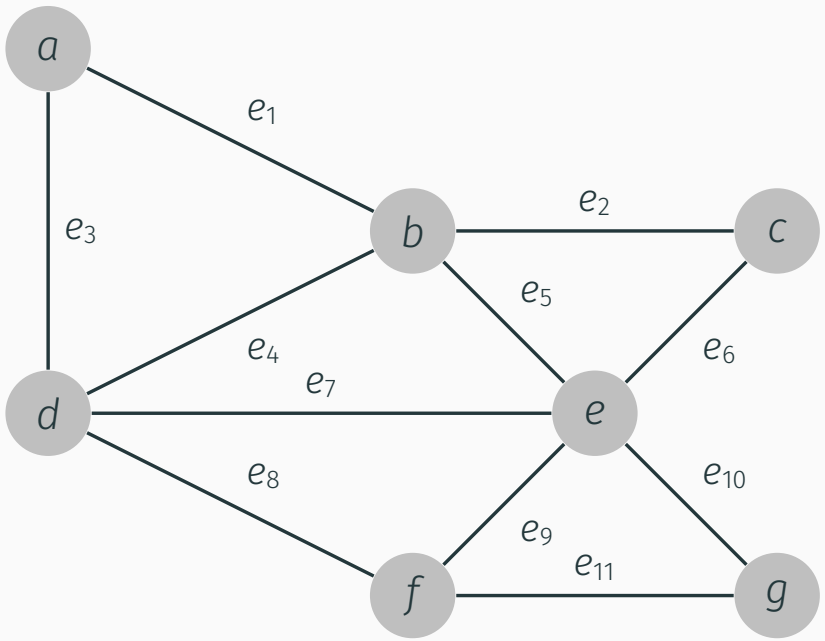


More Basic Concepts: Subgraphs and Trees

Subgraph $S = (V_S, E_S)$: $V_S \subset V$, and $E_S \subset E$

Spanning subgraph $S = (V_S, E_S)$: $V_S = V$, and $E_S \subset E$

Connected component: a maximal connected subgraph of G



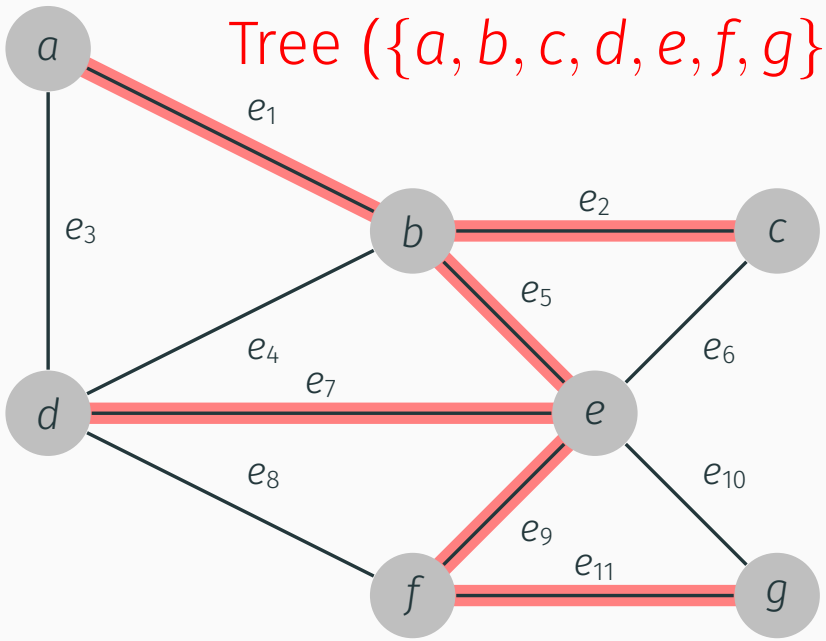
More Basic Concepts: Subgraphs and Trees

Subgraph $S = (V_S, E_S)$: $V_S \subset V$, and $E_S \subset E$

Spanning subgraph $S = (V_S, E_S)$: $V_S = V$, and $E_S \subset E$

Connected component: a maximal connected subgraph of G

Tree: a connected, acyclic, undirected graph



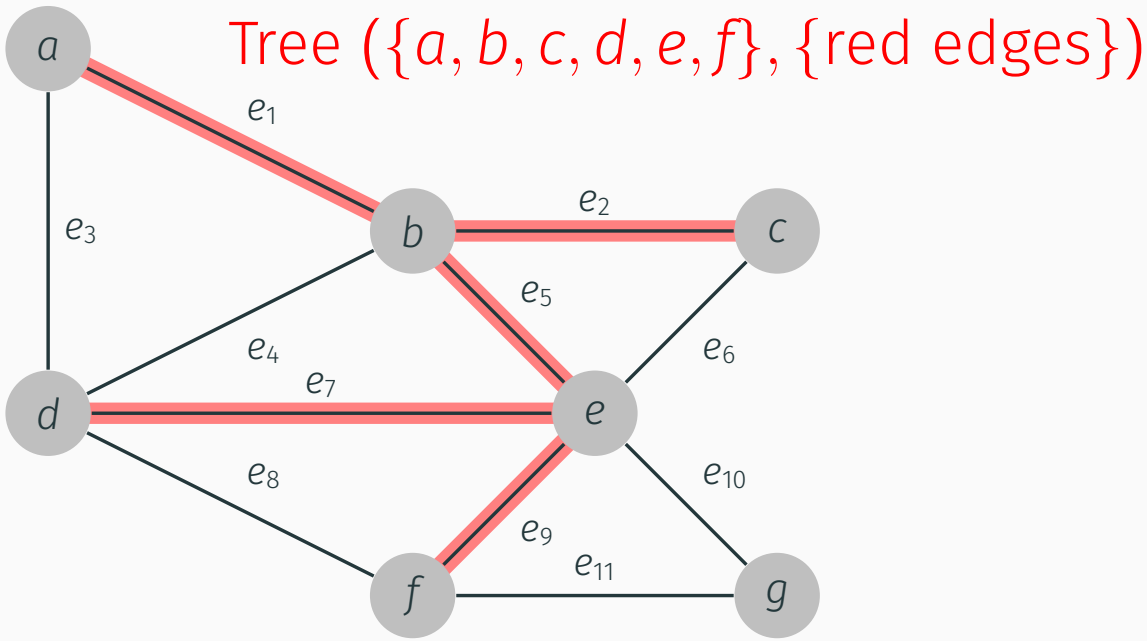
More Basic Concepts: Subgraphs and Trees

Subgraph $S = (V_S, E_S)$: $V_S \subset V$, and $E_S \subset E$

Spanning subgraph $S = (V_S, E_S)$: $V_S = V$, and $E_S \subset E$

Connected component: a maximal connected subgraph of G

Tree: a connected, acyclic, undirected graph



More Basic Concepts: Subgraphs and Trees

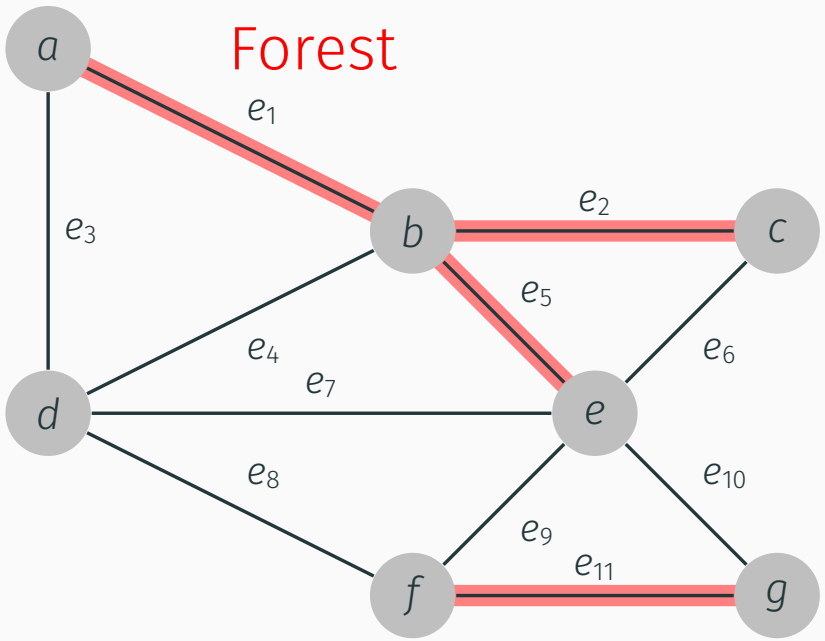
Subgraph $S = (V_S, E_S)$: $V_S \subset V$, and $E_S \subset E$

Spanning subgraph $S = (V_S, E_S)$: $V_S = V$, and $E_S \subset E$

Connected component: a maximal connected subgraph of G

Tree: a connected, acyclic, undirected graph

Forest: a set of trees (not necessarily connected)



More Basic Concepts: Subgraphs and Trees

Subgraph $S = (V_S, E_S)$: $V_S \subset V$, and $E_S \subset E$

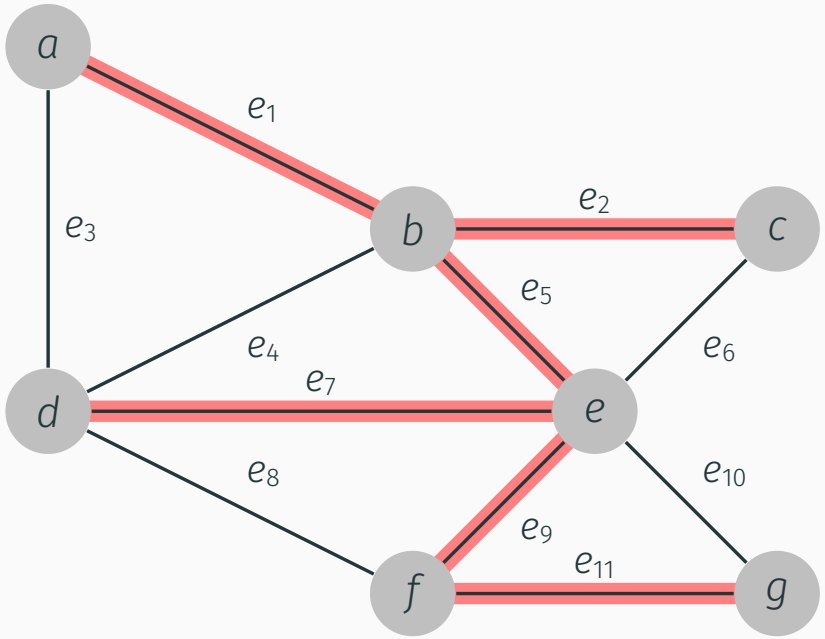
Spanning subgraph $S = (V_S, E_S)$: $V_S = V$, and $E_S \subset E$

Connected component: a maximal connected subgraph of G

Tree: a connected, acyclic, undirected graph

Forest: a set of trees (not necessarily connected)

Spanning Tree: a spanning subgraph that is a tree



Properties

Property 1

$\sum_v deg(v) = 2|E|$

Properties

Property 1

$\sum_v deg(v) = 2|E|$

Proof: each edge is counted twice

Properties

Property 1

$\sum_v deg(v) = 2|E|$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges
$|E| \leq |V|(|V| - 1)/2$

Properties

Property 1

$\sum_v deg(v) = 2|E|$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges
$|E| \leq |V|(|V| - 1)/2$

Proof: each vertex has degree at most $(|V| - 1)$

Properties

Property 1

$\sum_v deg(v) = 2|E|$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges
$|E| \leq |V|(|V| - 1)/2$

Proof: each vertex has degree at most $(|V| - 1)$

What is the bound for a digraph?

Properties

Property 1

$\sum_v deg(v) = 2|E|$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges
$|E| \leq |V|(|V| - 1)/2$

Proof: each vertex has degree at most $(|V| - 1)$

What is the bound for a digraph? $|E| \leq |V|(|V| - 1)$

Graph Interface: Graph.java

```
1  public interface Graph<V,E> {
2    numVertices();
3    numEdges();
4
5    outDegree(Vertex<V> v);
6    inDegree(Vertex<V> v);
7
8    getEdge(Vertex<V> u, Vertex<V> v);
9    endVertices(Edge<E> e);
10
11   insertVertex(V element);
12   insertEdge(Vertex<V> u, Vertex<V> v, E element);
13
14   removeVertex(Vertex<V> v);
15   removeEdge(Edge<E> e);
16 }
```

Graph

Represent and Implement Graphs

Representing Graphs

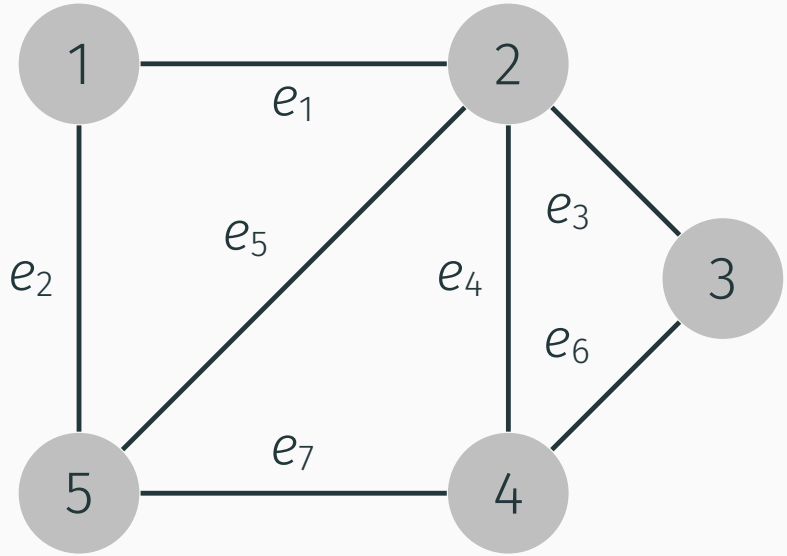
Three basic methods

- Edge List
- Adjacency List
- Adjacency Matrix

Representing Graphs

Three basic methods

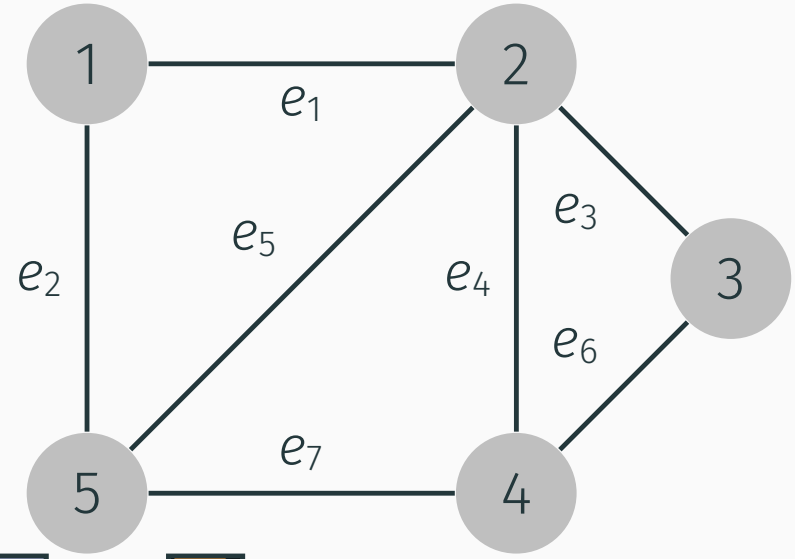
- Edge List
- Adjacency List
- Adjacency Matrix

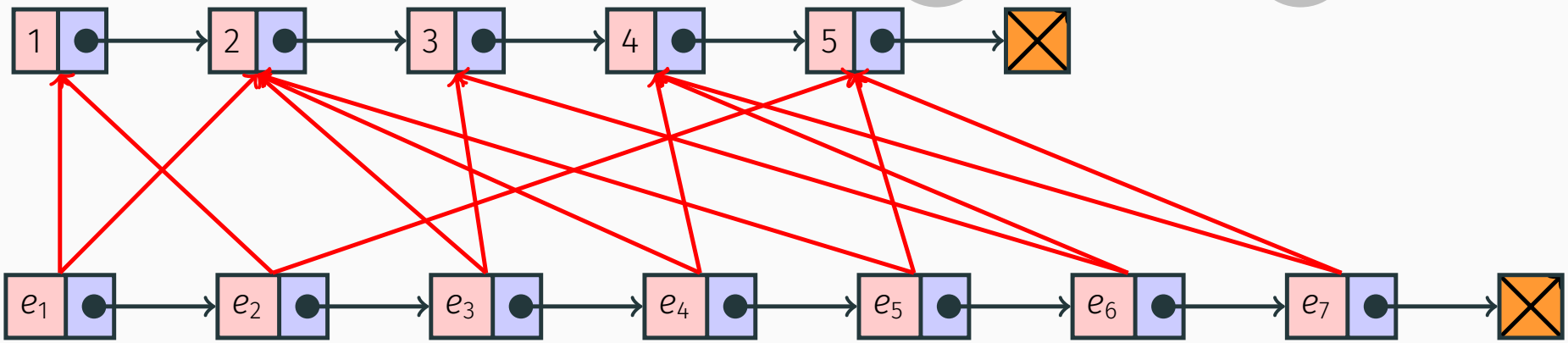


Representing Graphs

Three basic methods

- Edge List
- Adjacency List
- Adjacency Matrix





Vertex object:

- element

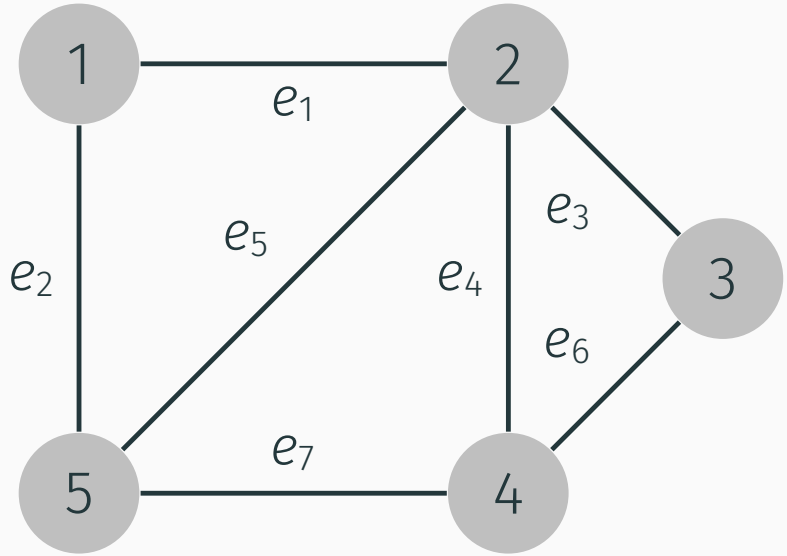
Edge object:

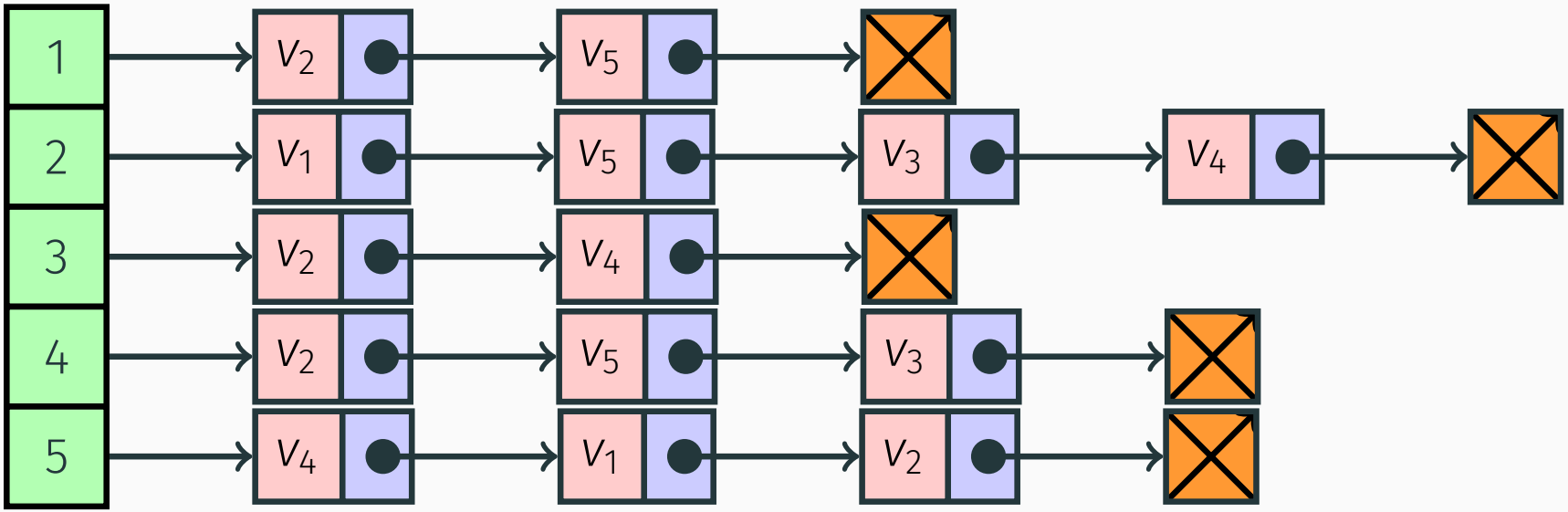
- element
- origin vertex object
- destination vertex object

Representing Graphs

Three basic methods

- Edge List
- Adjacency List
- Adjacency Matrix

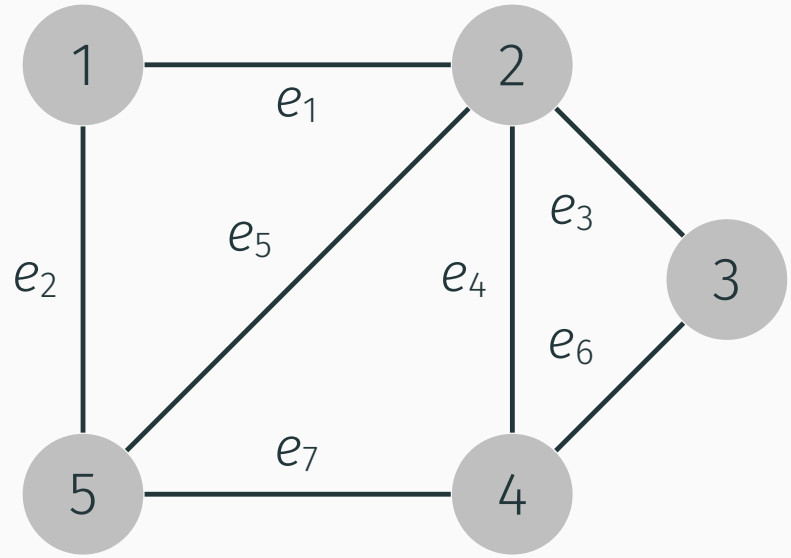




Representing Graphs

Three basic methods

- Edge List
- Adjacency List
- Adjacency Matrix



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Performance

- n vertices, m edges
- no parallel edges
- no self-loops Edge

	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
areAdjacent(v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	n + m	n^2
removeEdge(e)	1	deg(v) + deg(w)	1

Depth First Search

Depth First Search

Main Idea:

- Continue searching "deeper" into the graph, until we get "stuck".
- If we get stuck, "backtrack" to the first "available" vertex.

Used to help solve many graph problems, including

- Nodes that are reachable from a specific node v
- Detection of cycles
- Extraction of strongly connected components
- Topological sorts

Depth First Search

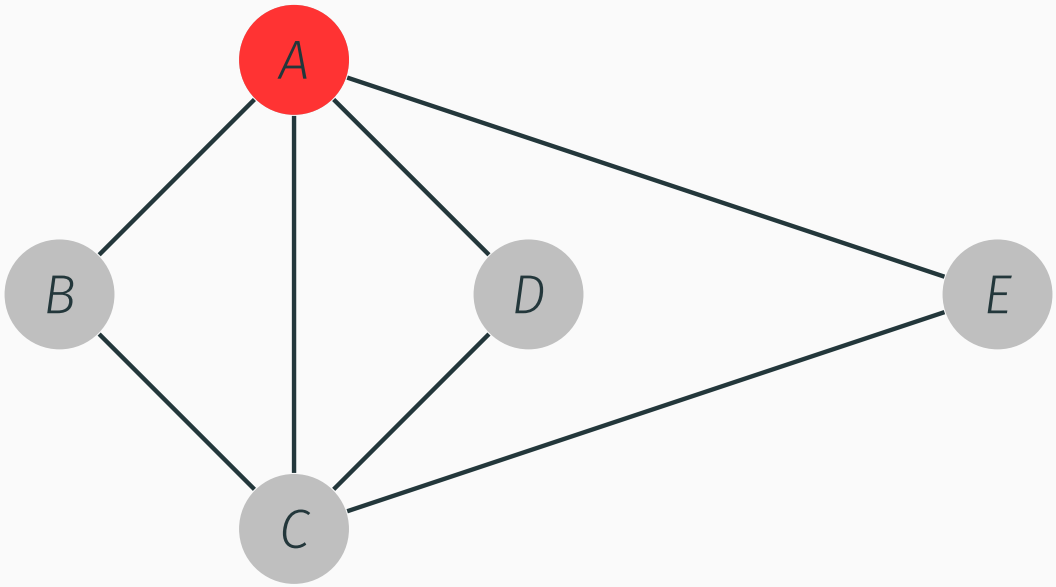
Keep track of progress by colouring vertices:

- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).

Depth First Search

Keep track of progress by colouring vertices:

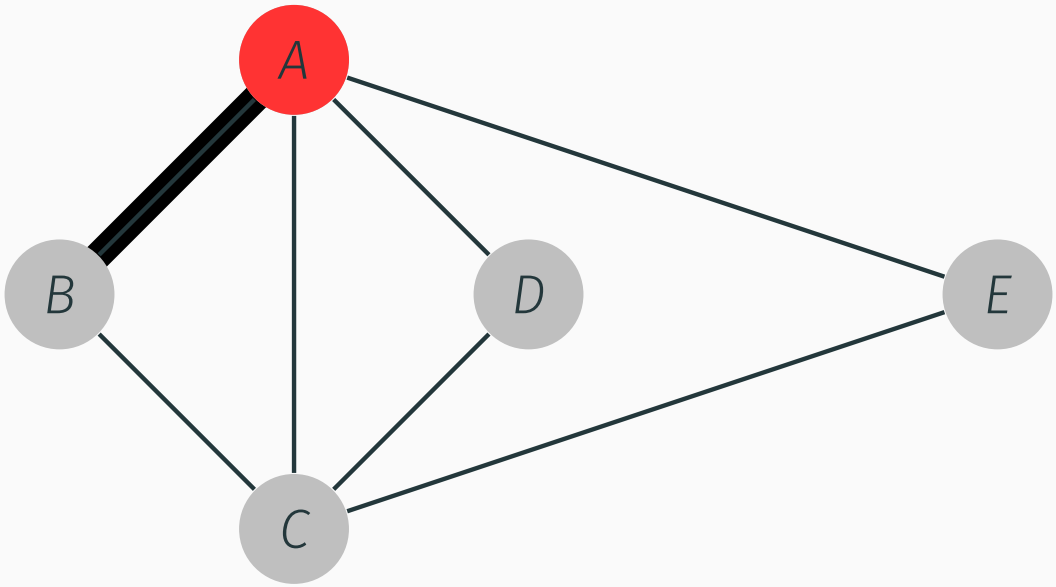
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

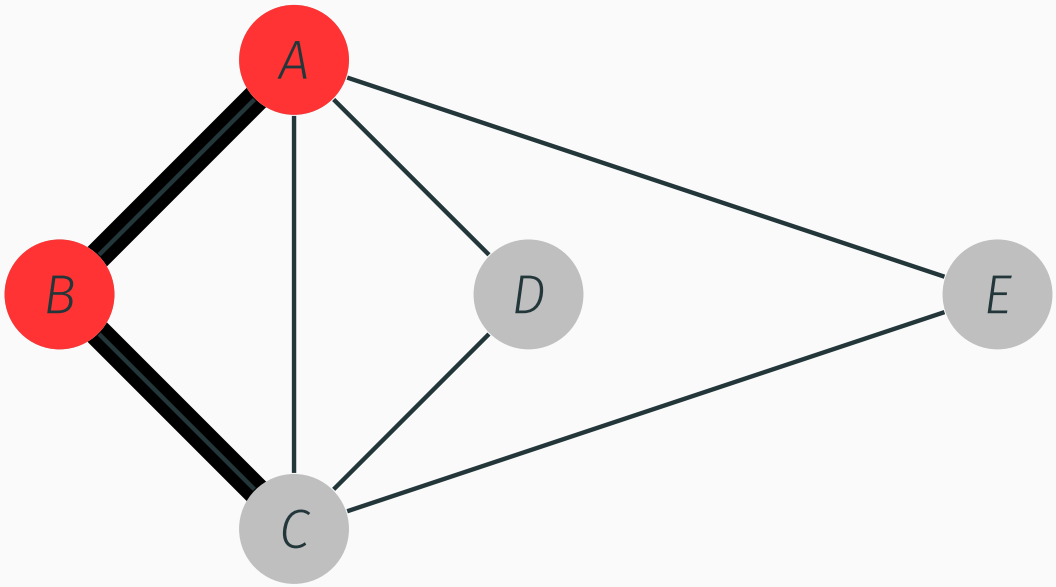
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

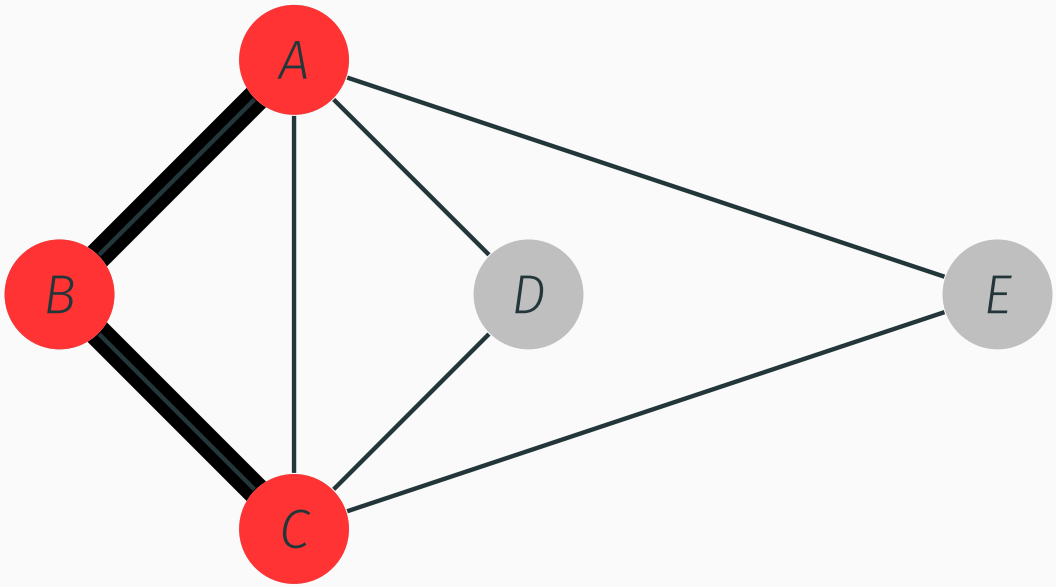
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

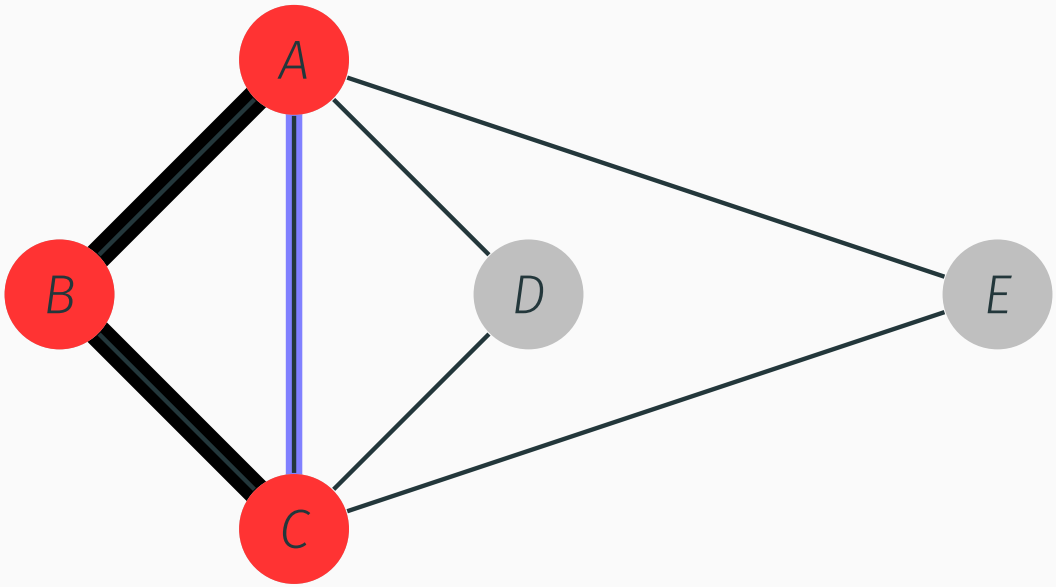
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

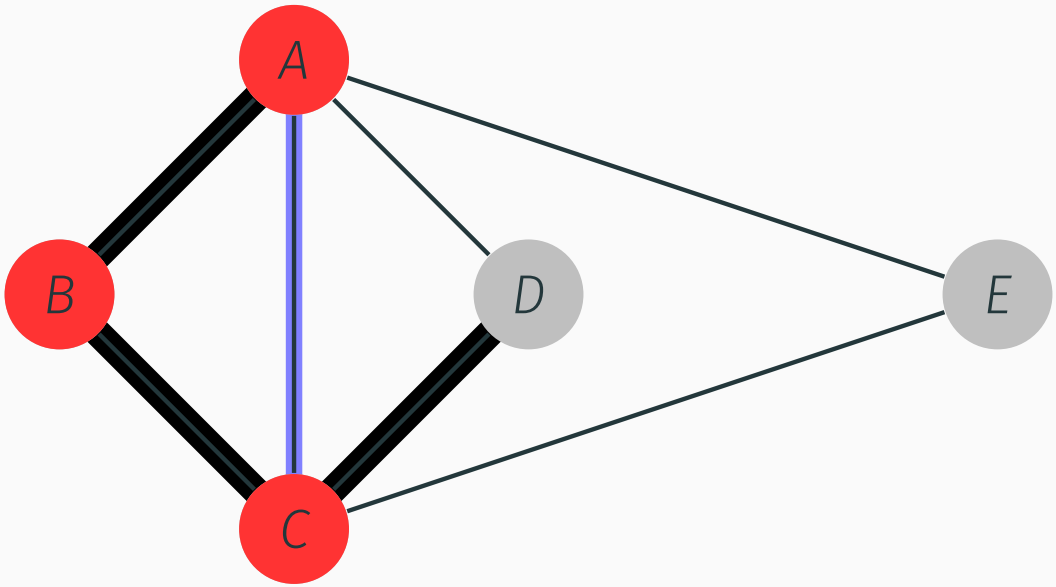
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

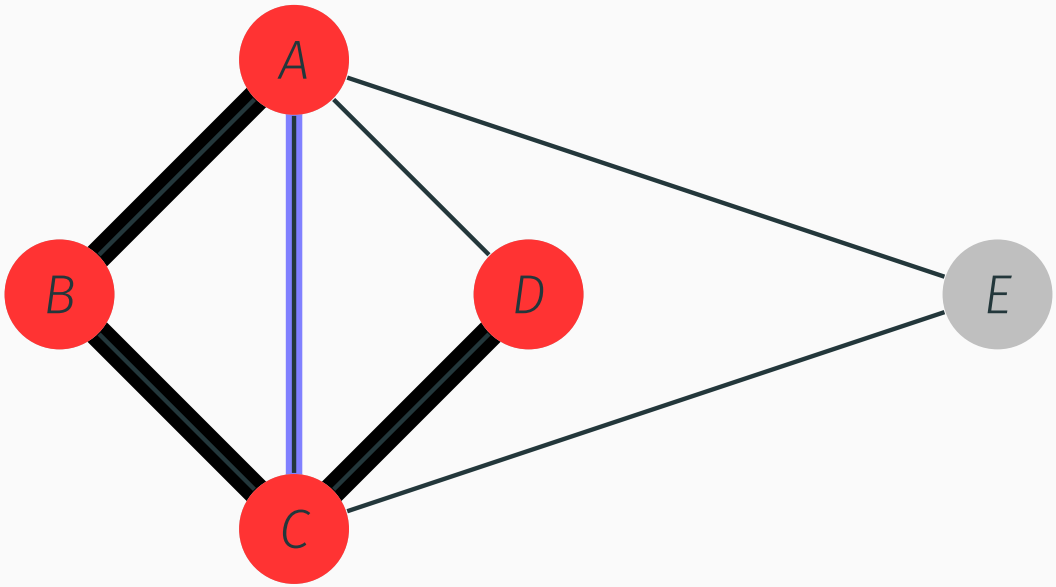
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

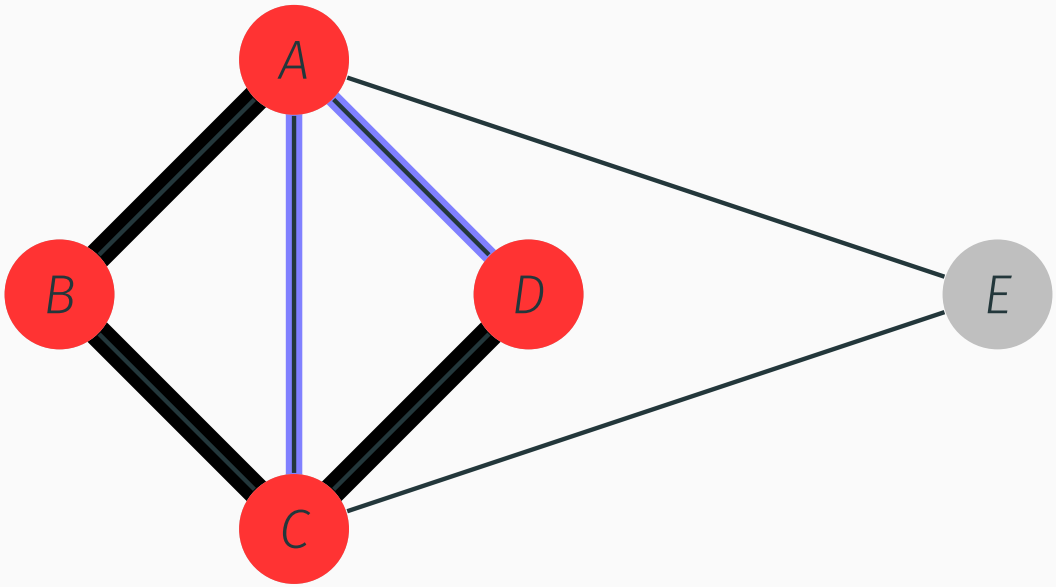
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

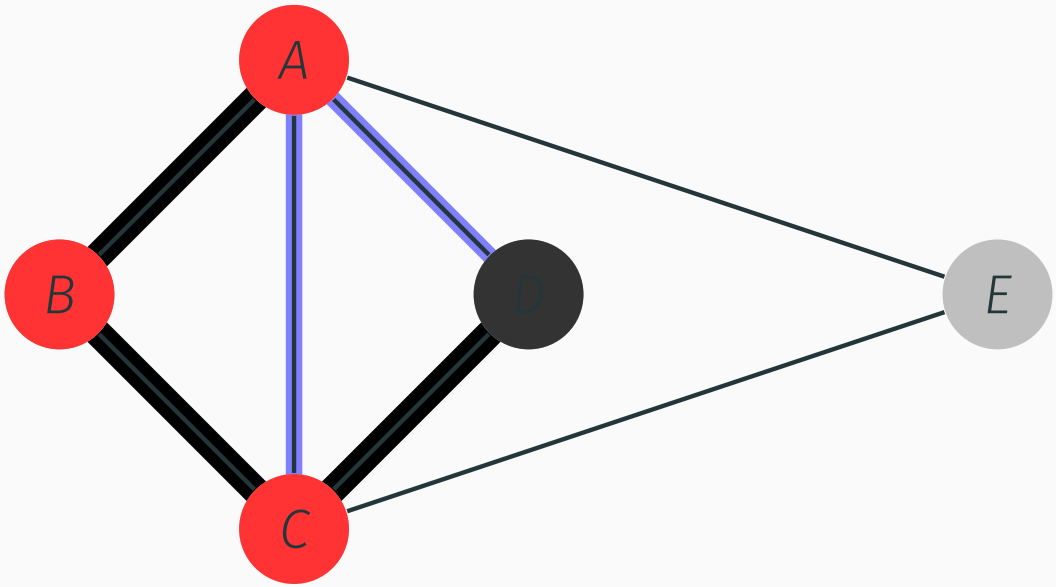
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

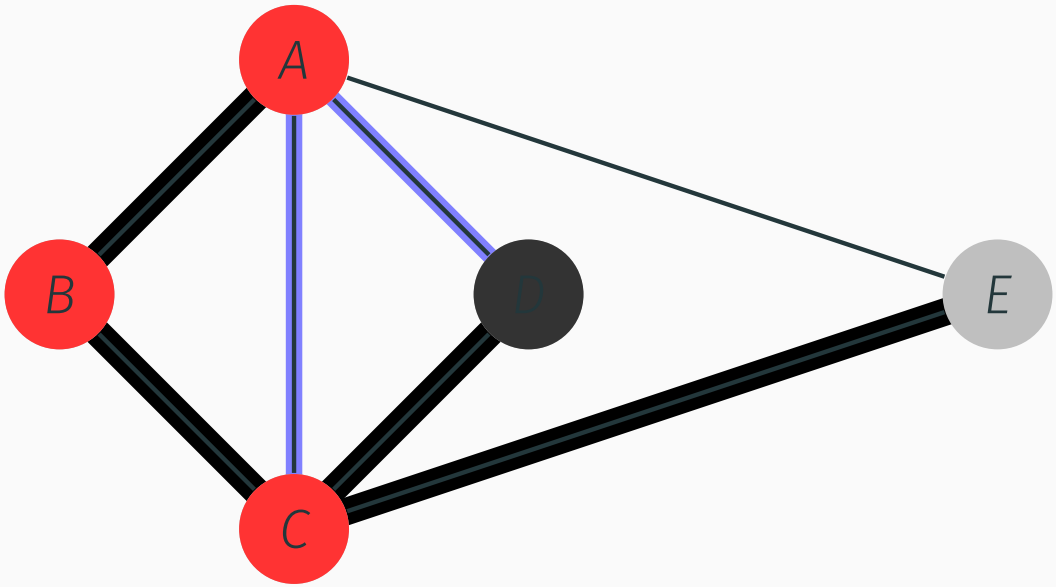
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

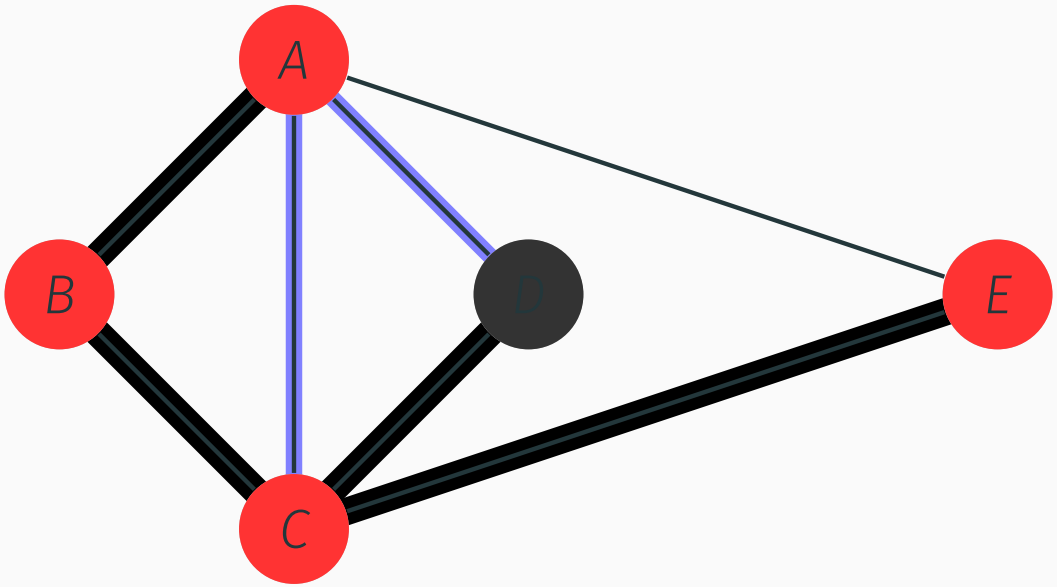
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

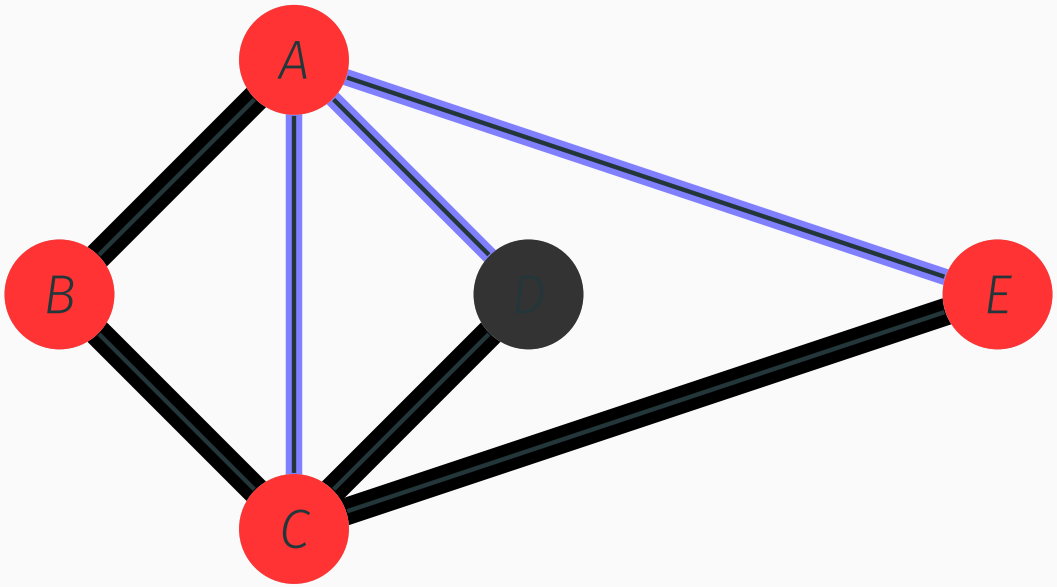
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

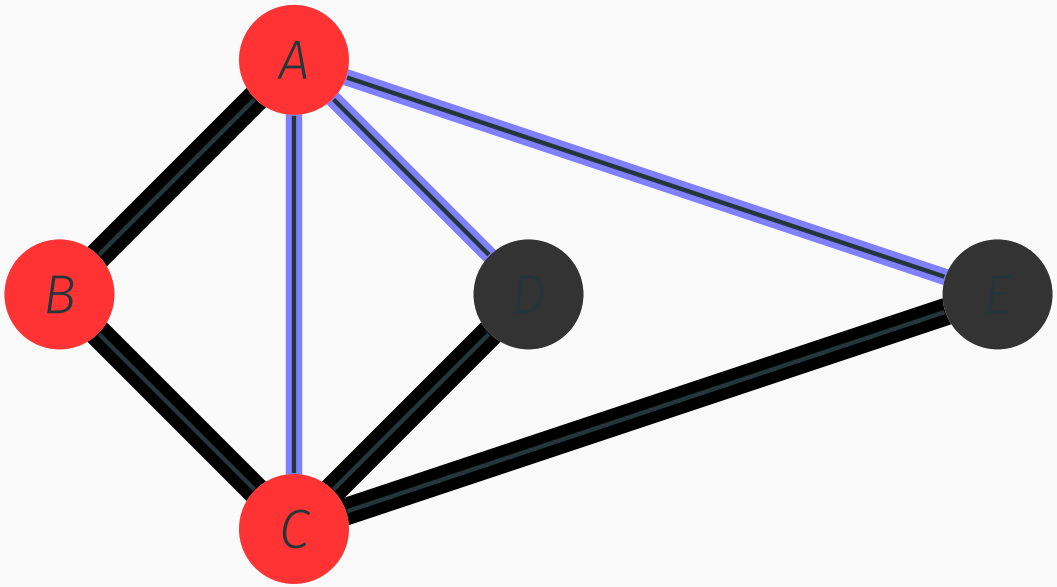
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

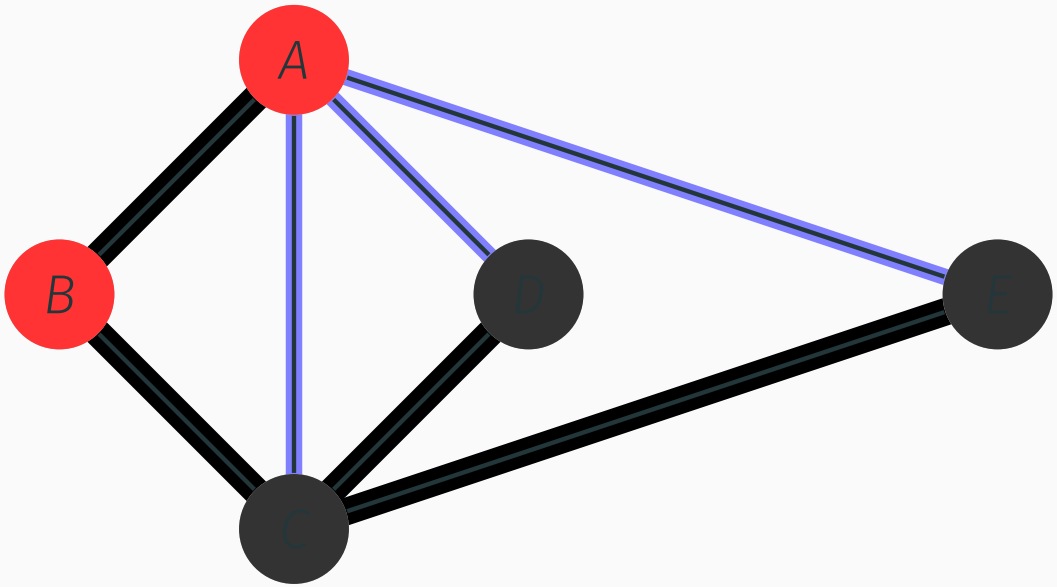
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

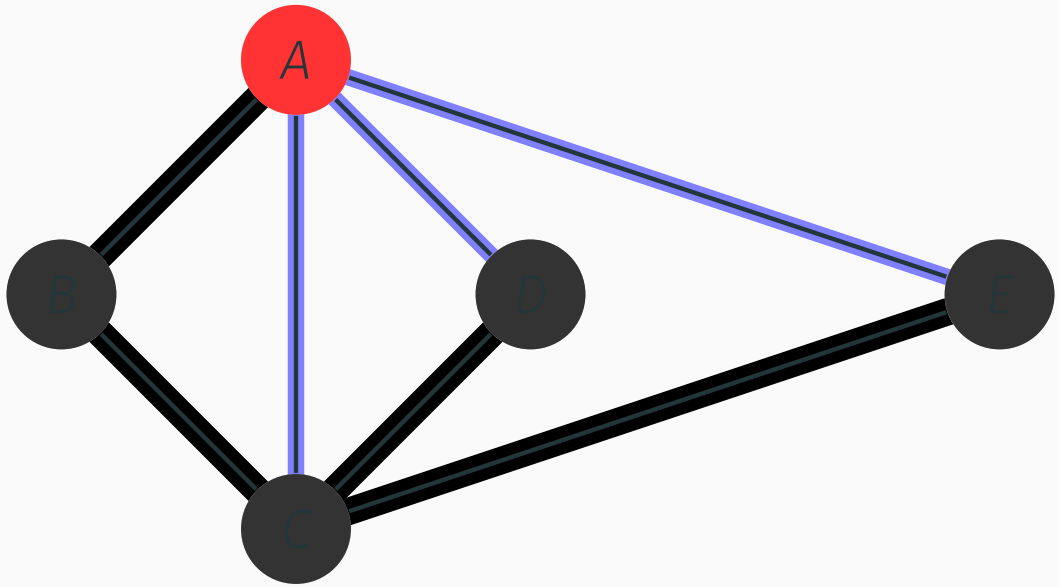
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

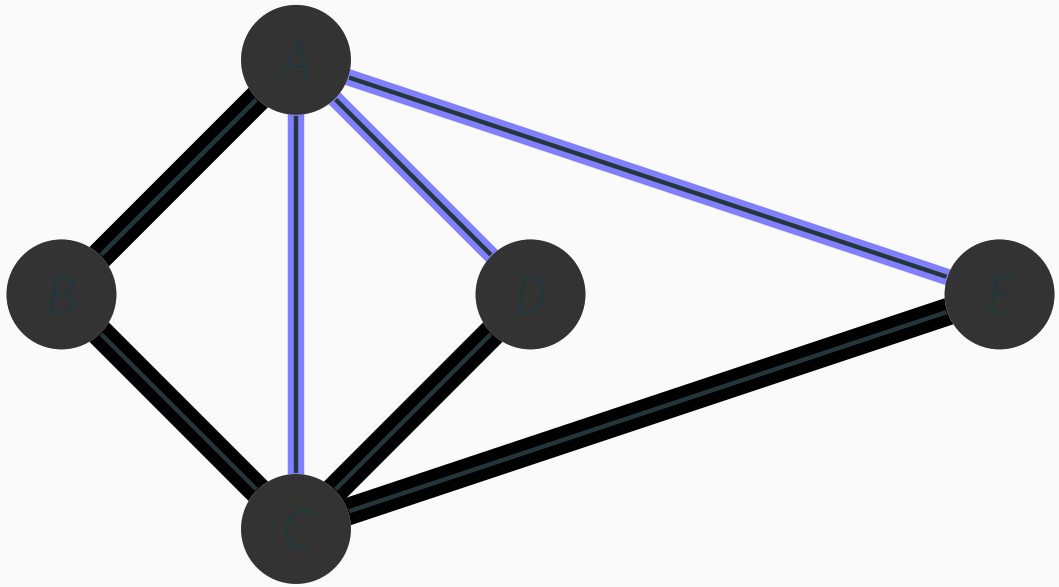
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).



Depth First Search

Keep track of progress by colouring vertices:

- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).

```
1  DFS(G)
2  // initialize vertex
3  for each vertex u in V
4    u.color = GRAY
5  for each vertex u in V
6    if u.color = GRAY
7      DFS-Visit(u)
```

```
1  DFS-Visit(u)
2
3  u.color = RED
4  for each v in u.Adj
5    if v.color = GRAY
6      DFS-Visit(v)
7  u.color = BLACK
```

Depth First Search

Keep track of progress by colouring vertices:

- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).

```
1  DFS(G)
2  // initialize vertex
3  for each vertex u in V
4    u.color = GRAY
5  for each vertex u in V
6    if u.color = GRAY
7      DFS-Visit(u)
```

```
1  DFS-Visit(u)
2
3  u.color = RED
4  for each v in u.Adj
5    if v.color = GRAY
6      DFS-Visit(v)
7  u.color = BLACK
```

Running time: $O(n + m)$ (assuming adjacency list)

Depth First Search

Keep track of progress by colouring vertices:

- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).

```
1 DFS(G)
2 // initialize vertex
3 for each vertex u in V
4   u.color = GRAY
5 for each vertex u in V
6   if u.color = GRAY
7     DFS-Visit(u)
```

```
1 DFS-Visit(u)
2
3 u.color = RED
4 for each v in u.Adj
5   if v.color = GRAY
6     DFS-Visit(v)
7 u.color = BLACK
```

DFS-Visit(u) visits all the vertices and edges in the connected component of *us*

Depth First Search

Keep track of progress by colouring vertices:

- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished (still exploring from it)
- **Black**: finished (found everything reachable from it).

```
1 DFS(G)
2 // initialize vertex
3 for each vertex u in V
4   u.color = GRAY
5 for each vertex u in V
6   if u.color = GRAY
7     DFS-Visit(u)
```

```
1 DFS-Visit(u)
2
3 u.color = RED
4 for each v in u.Adj
5   if v.color = GRAY
6     DFS-Visit(v)
7 u.color = BLACK
```

The discovery edges labeled by *DFS-Visit(u)* form a spanning tree of the connected component of *u*

DFS Application 1: Path Finding

The DFS pattern can be used to find a path between two given vertices u and z, if one exists

```
1  DFS-Path(u,z,stack)
2
3  u.color = RED
4  push u onto stack
5  if u = z
6    return true
7  for each v in u.Adj // explore edge (u,v)
8    if v.color = GRAY
9      if DFS-Path(v,z,stack)
10       return true
11 u.color = BLACK
12 pop u from stack
13 return false
```

DFS Application 2: Cycle Finding

The DFS pattern can be used to determine whether a graph is acyclic.

```
1  DFS-Cycle(u)
2
3  u.color = RED
4  for each v in u.Adj // explore edge (u,v)
5    if v.color = RED  // detect a back edge
6      return true
7    else if v.color = GRAY
8      if DFS-Cycle(v)
9        return true
10 u.color = BLACK
11 return false
```

Breadth First Search

Breadth First Search

Main Idea:

- Search the graph "as wide as possible"

Used to help solve many graph problems, including

- A BFS traversal of a graph G
- Visits all the vertices and edges of G
- Determines whether G is connected
- Computes the connected components of G
- Computes a spanning forest of G
- Cycle detection
- Find and report a path with the minimum number of edges between two given vertices

Breadth First Search

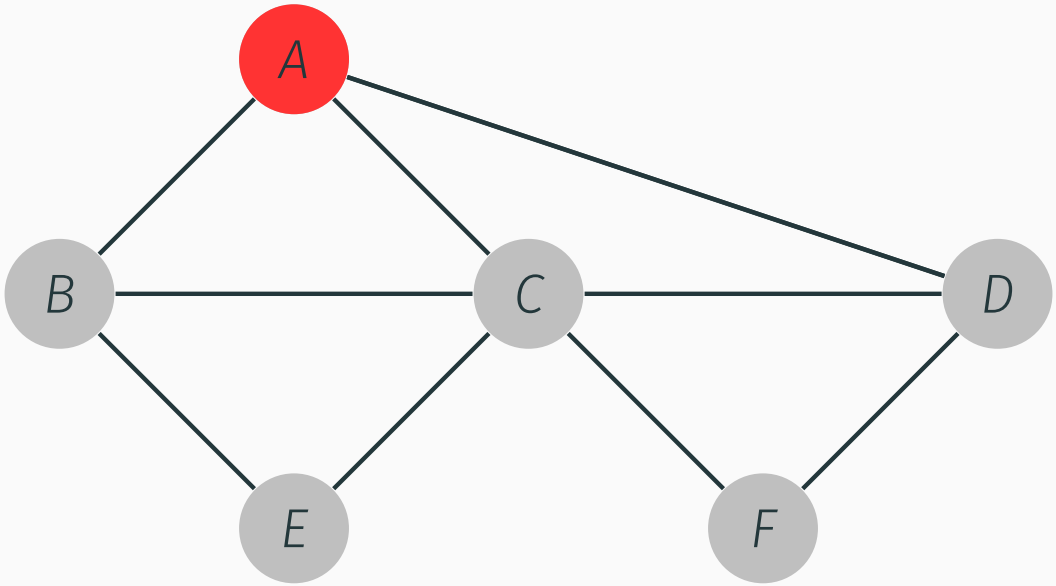
Keep track of progress by colouring vertices:

- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished

Breadth First Search

Keep track of progress by colouring vertices:

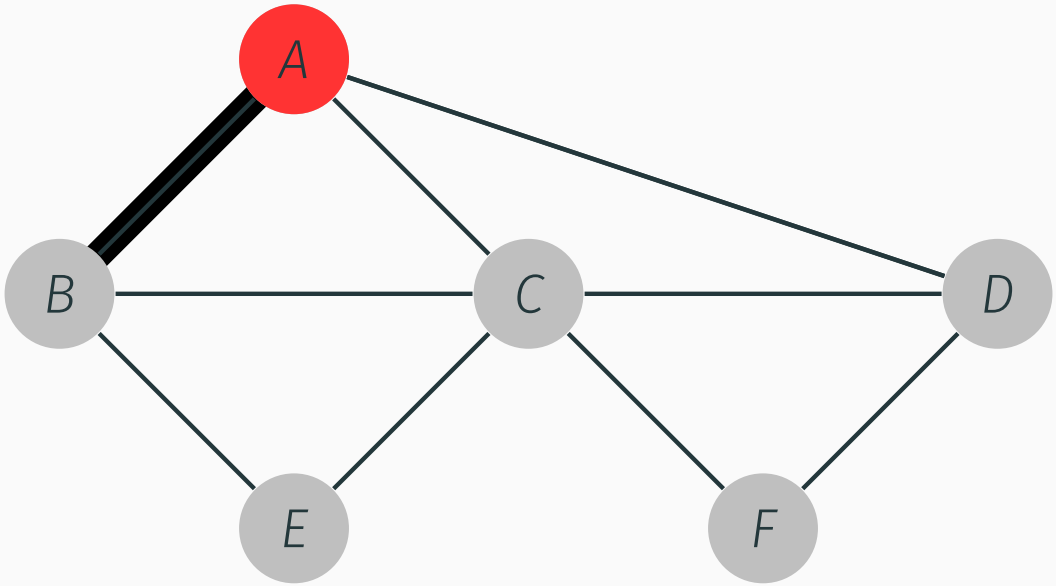
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

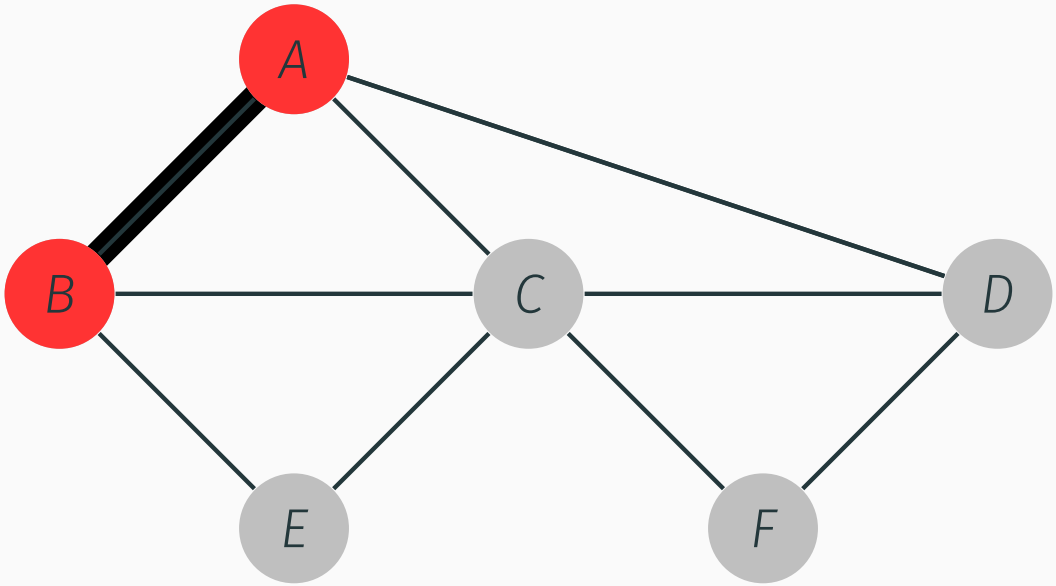
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

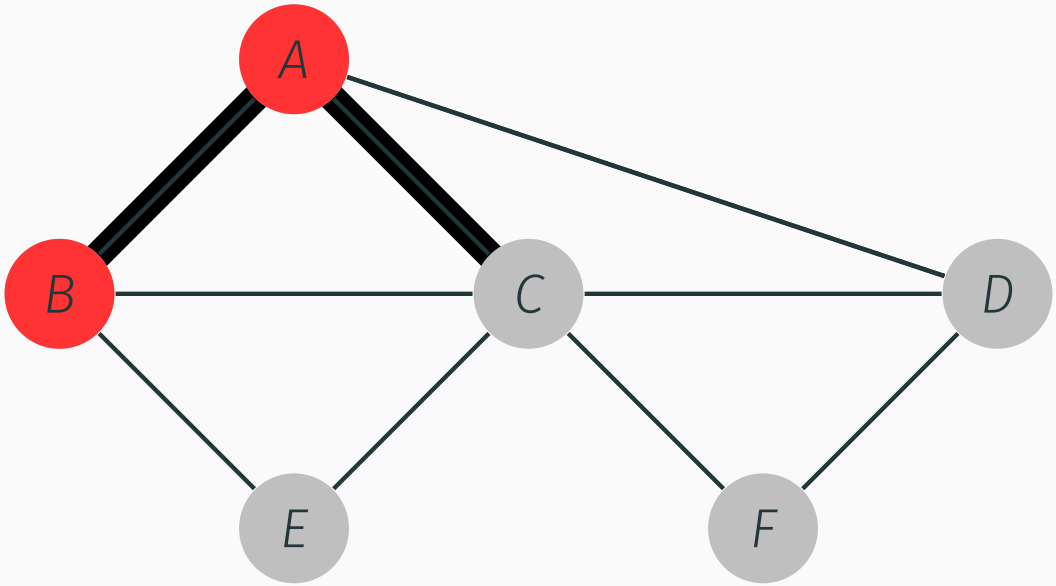
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

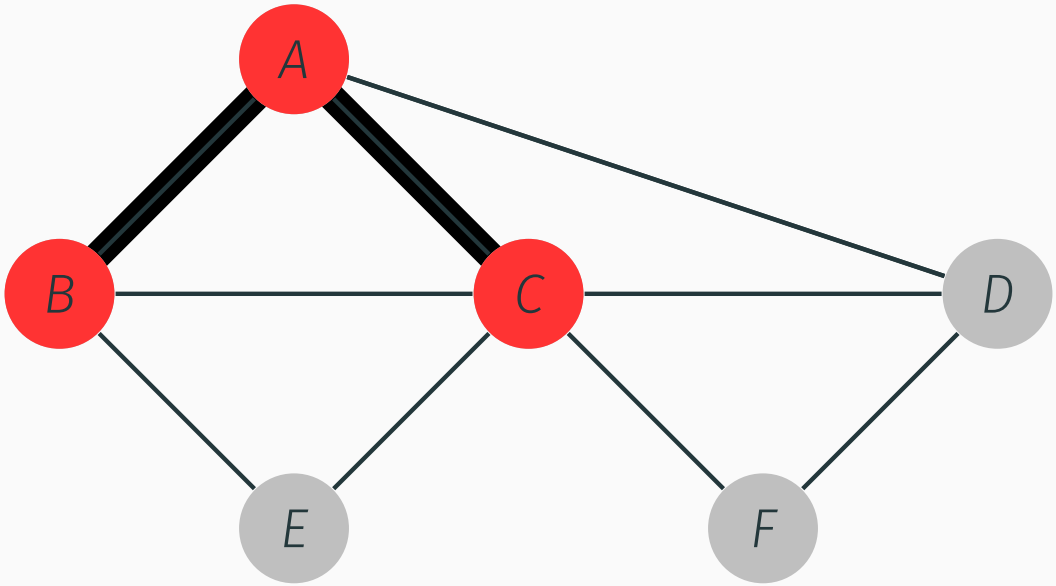
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

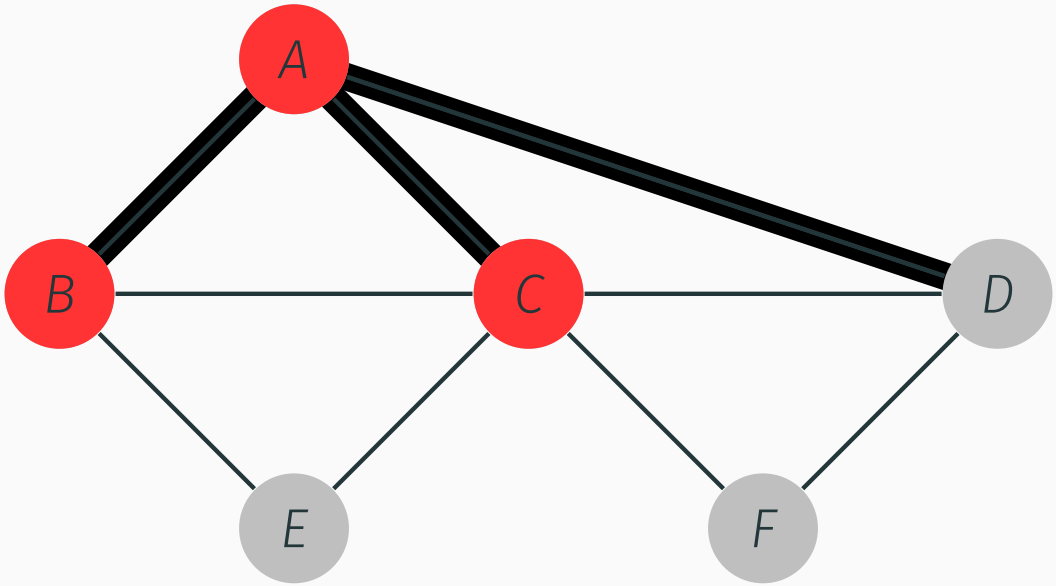
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

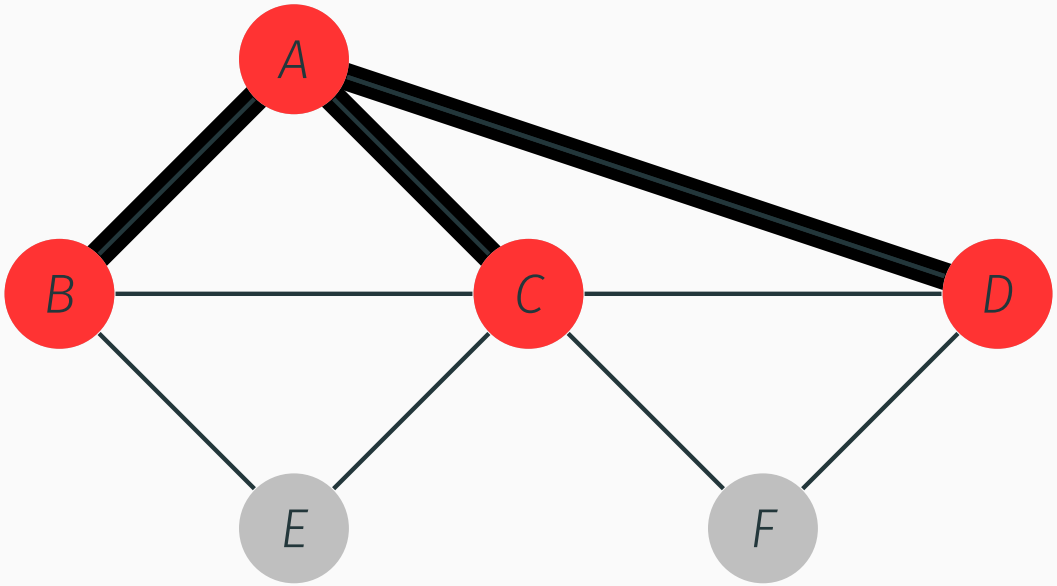
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

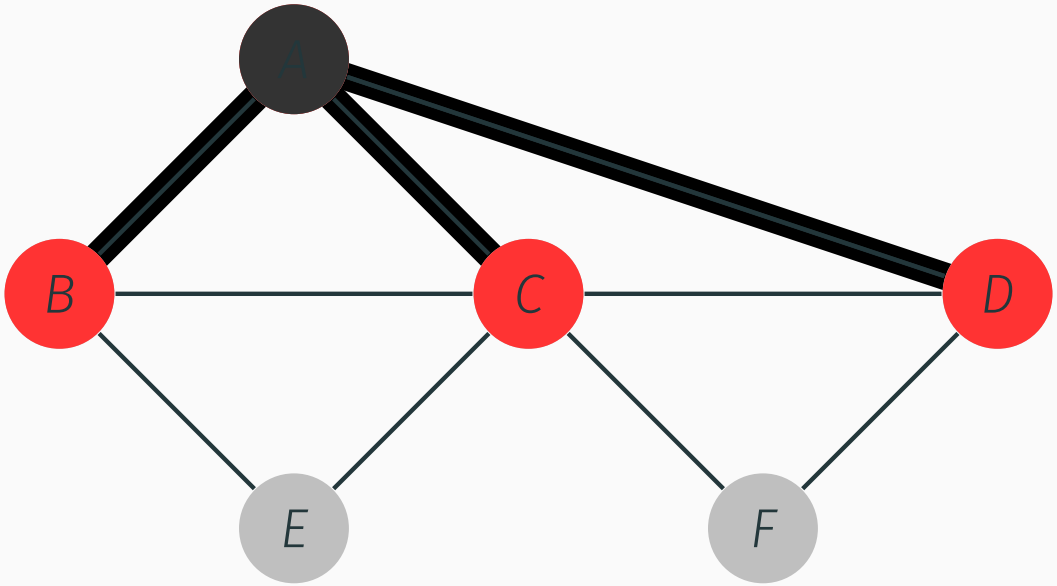
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

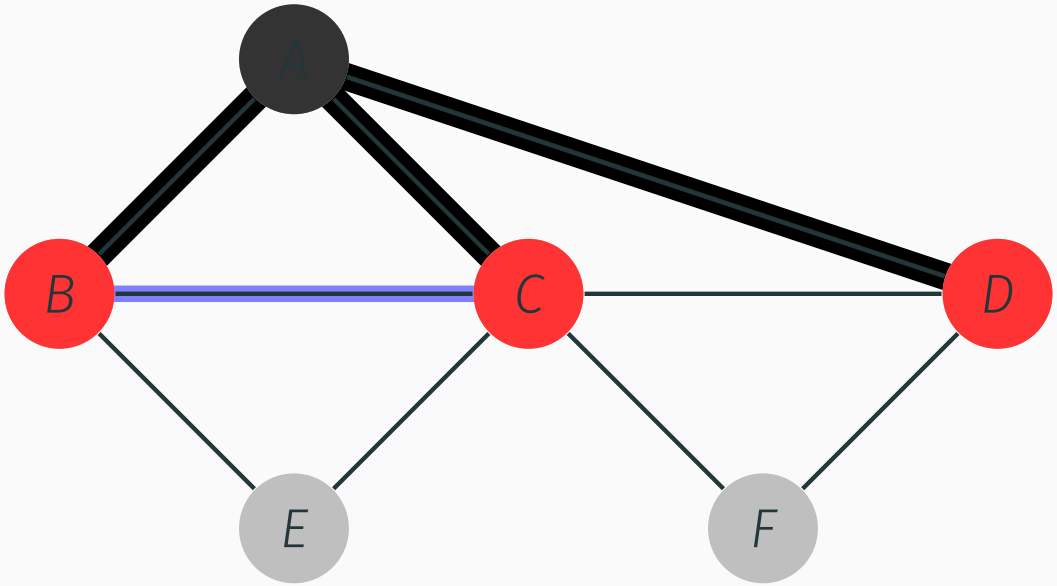
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

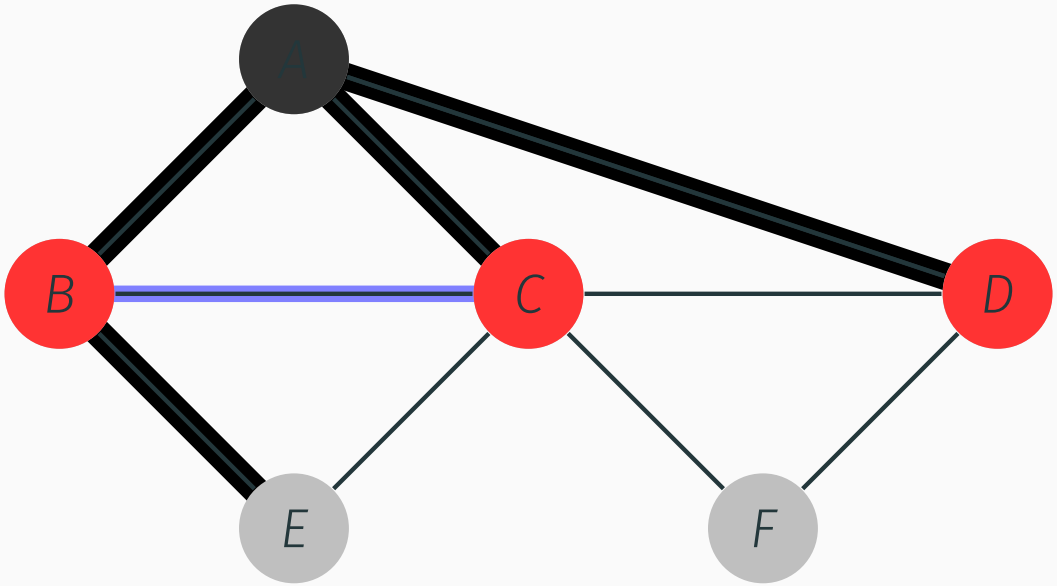
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

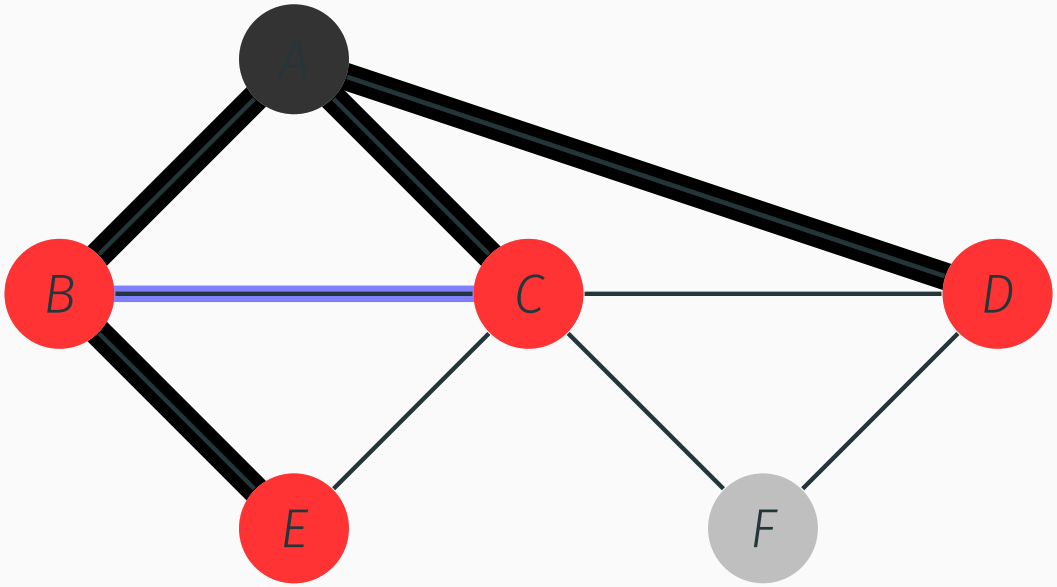
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

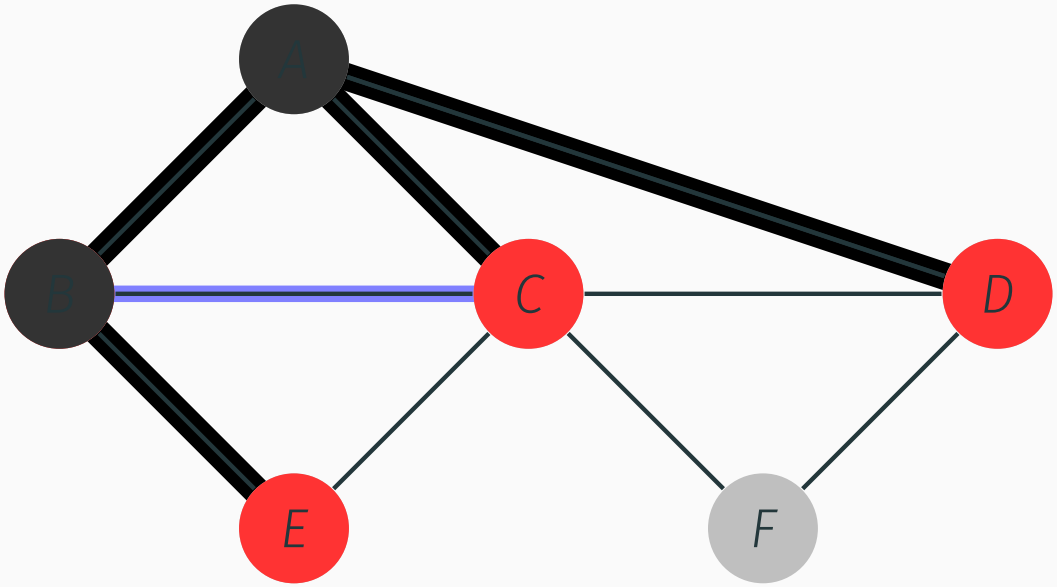
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

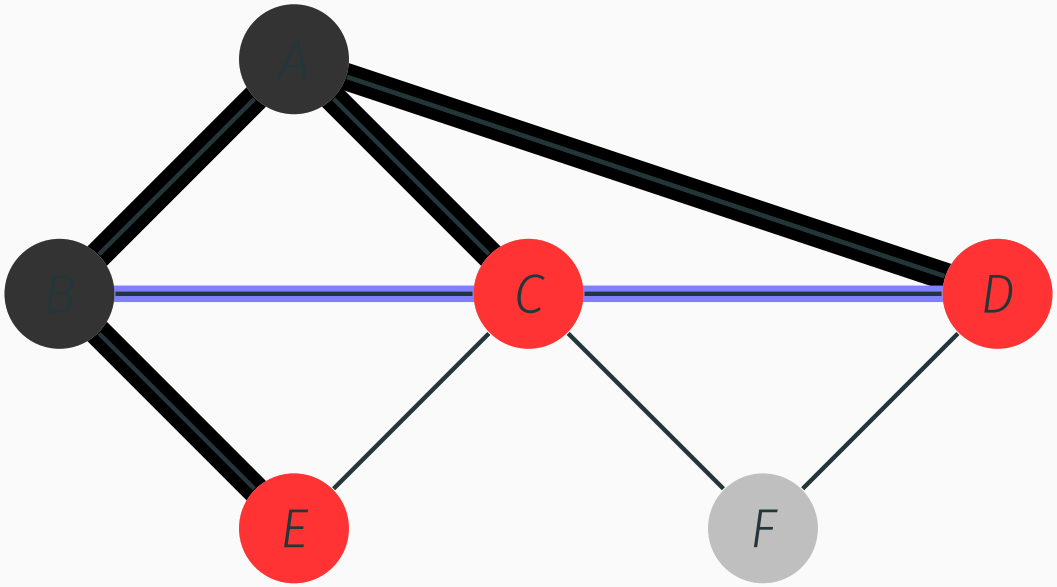
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

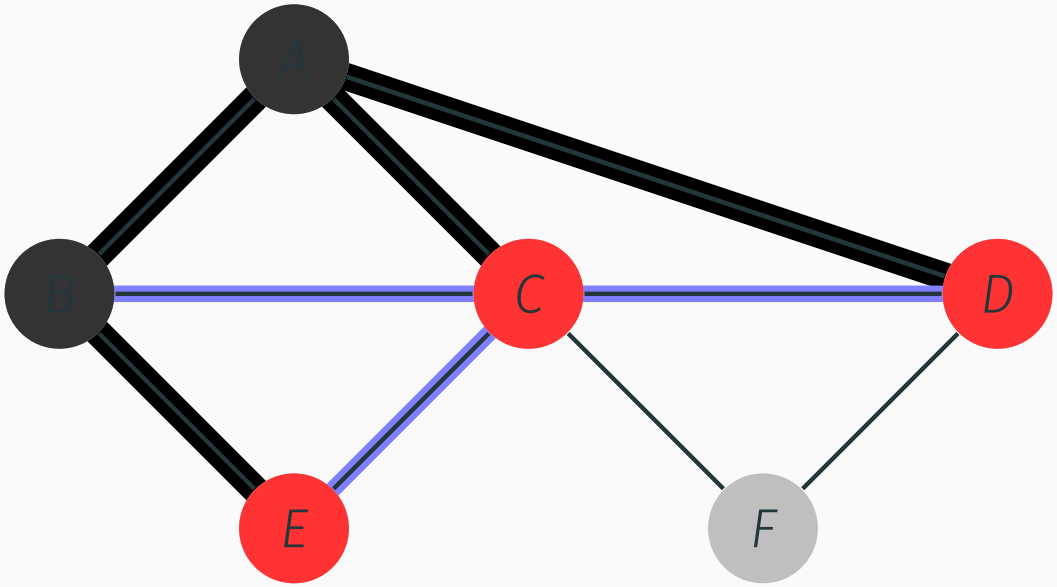
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

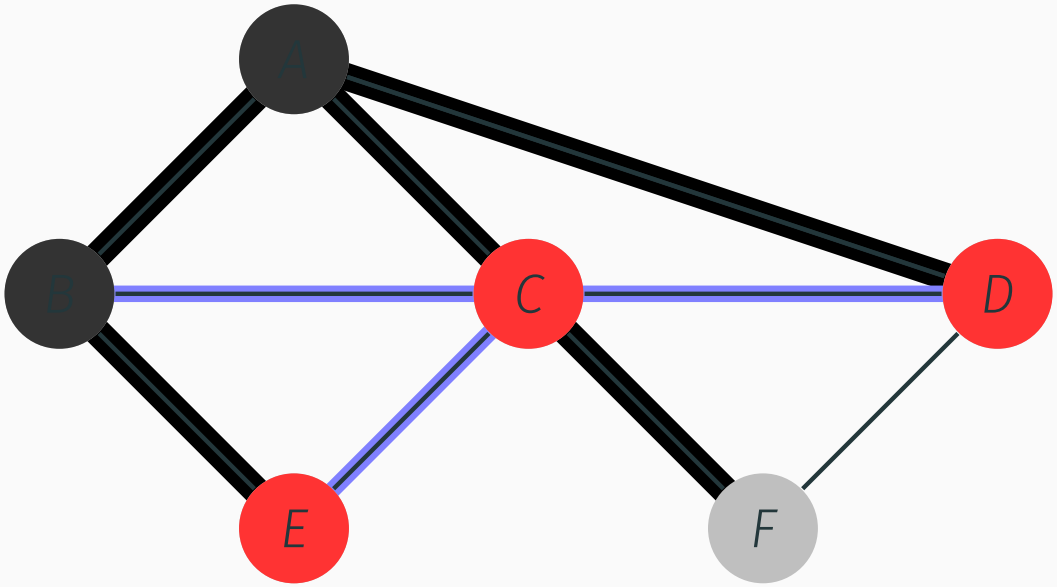
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

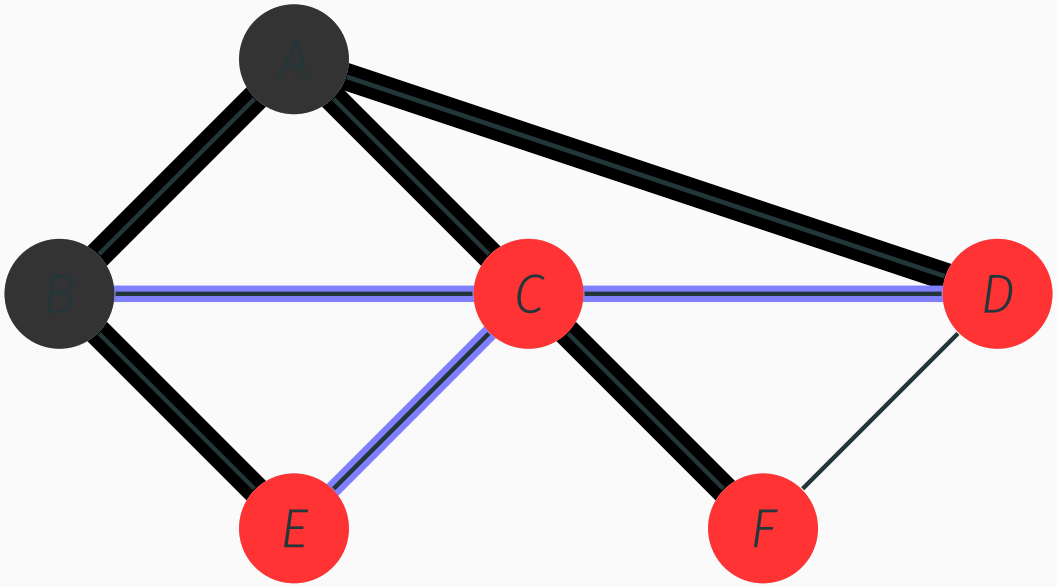
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

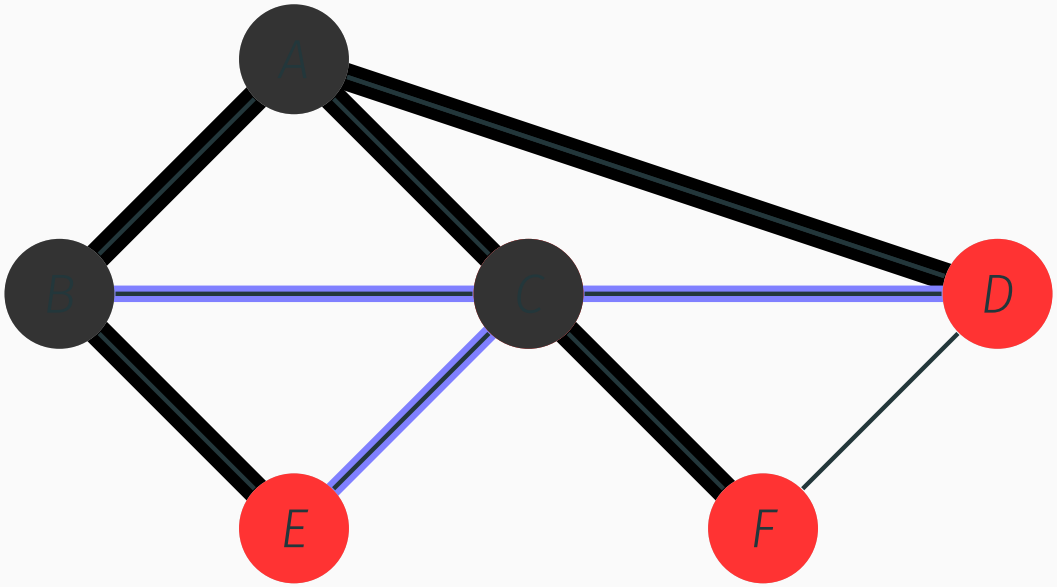
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

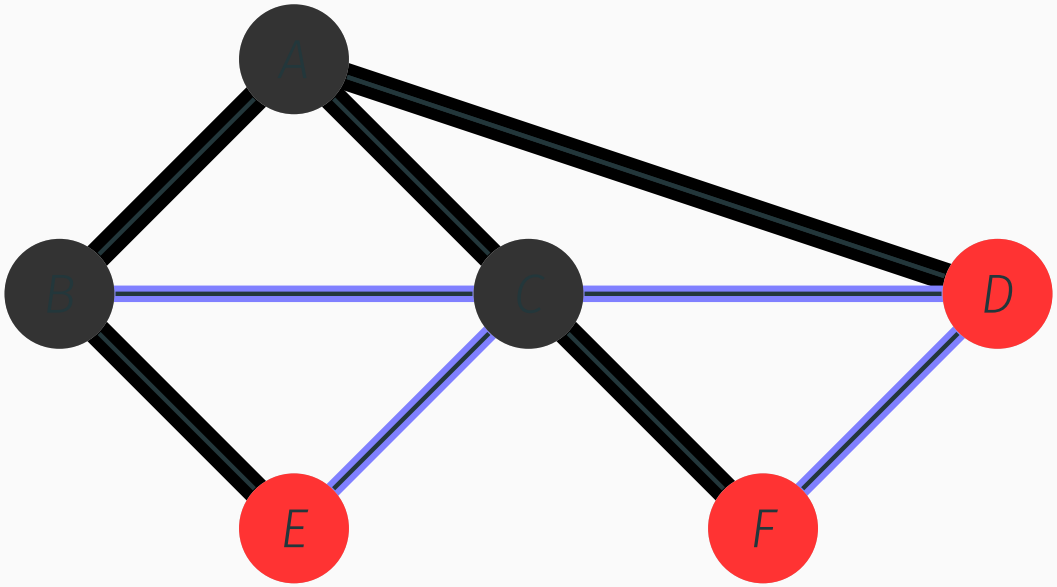
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

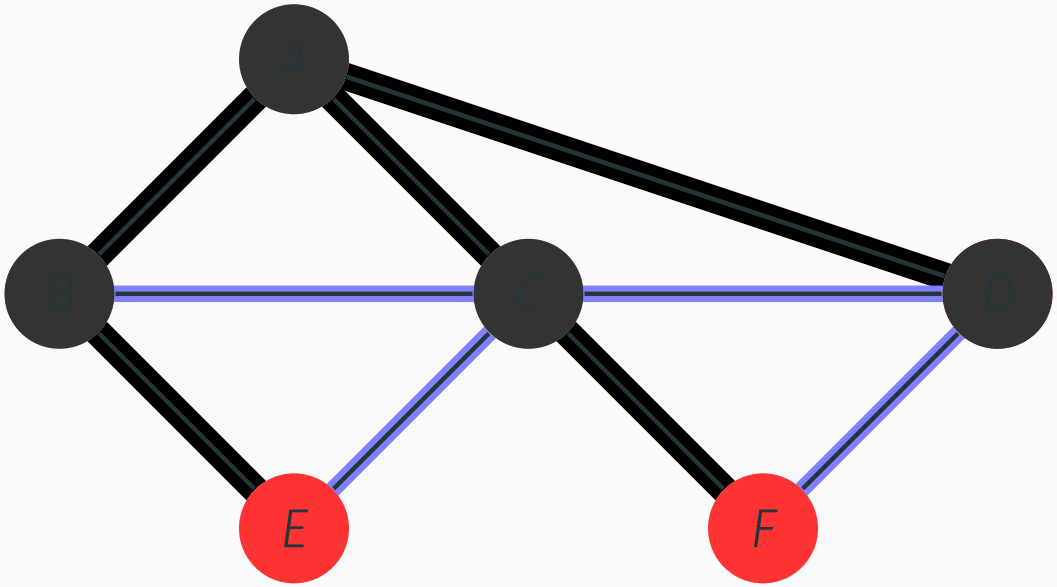
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

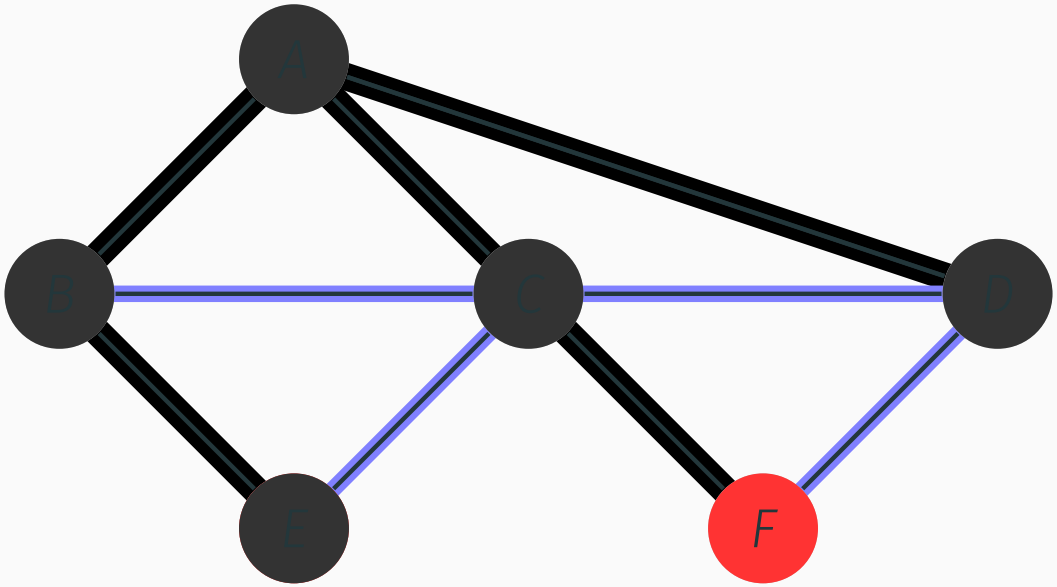
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

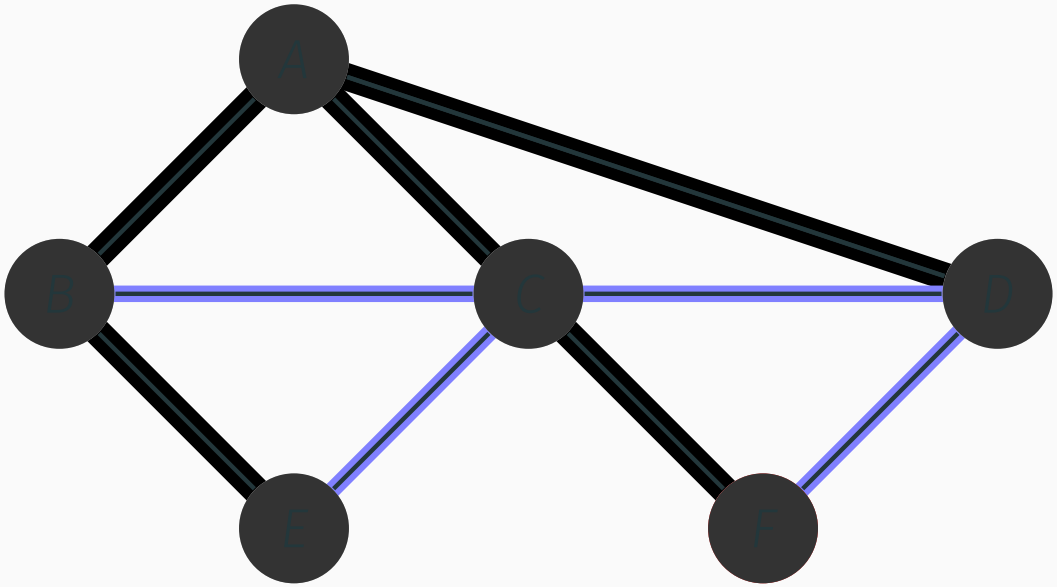
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

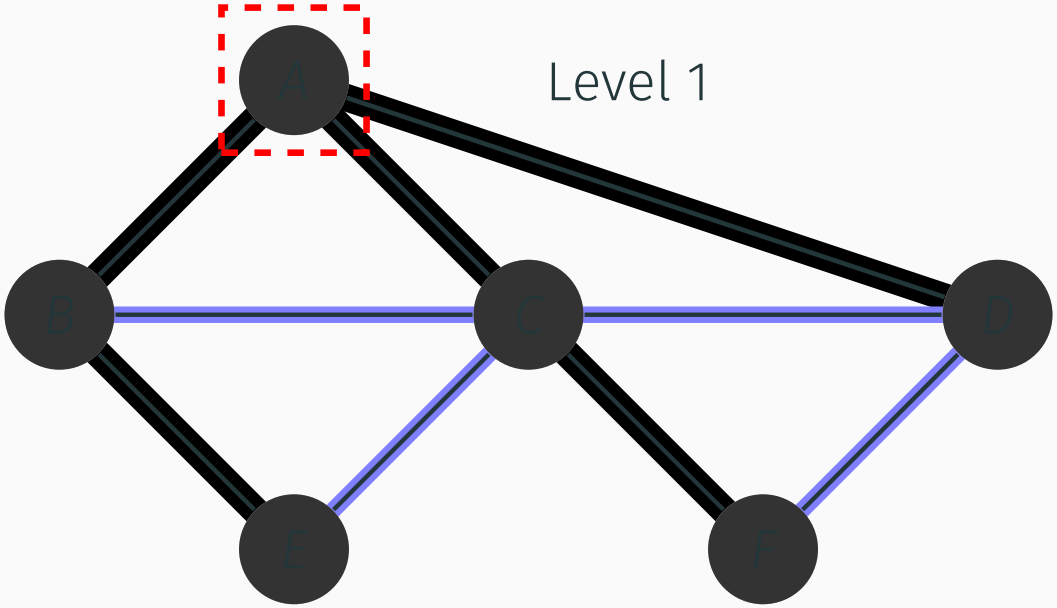
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

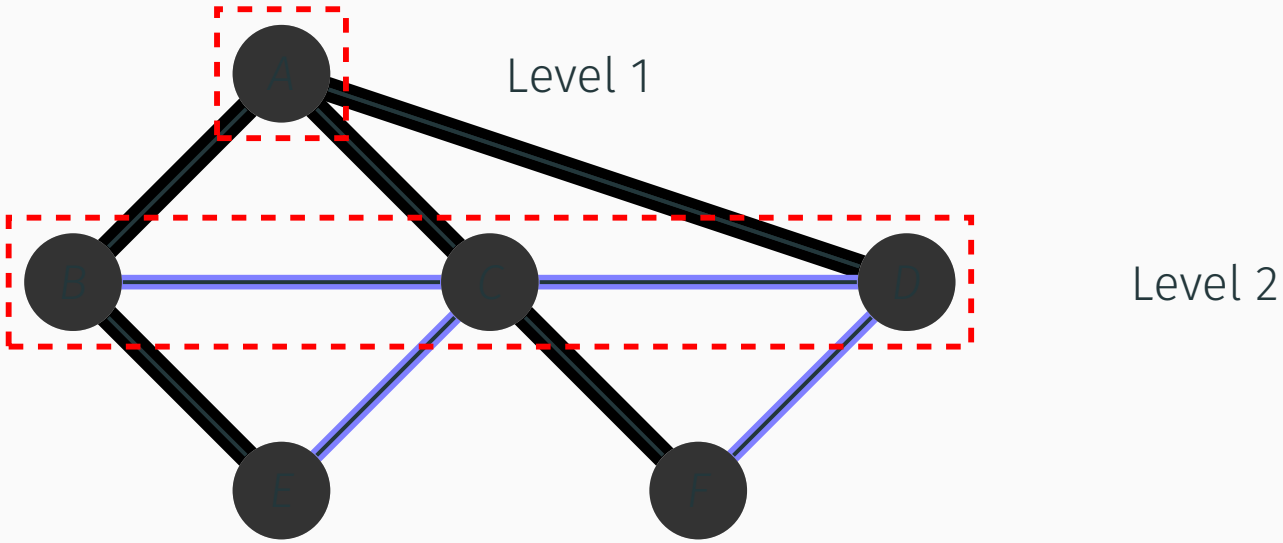
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

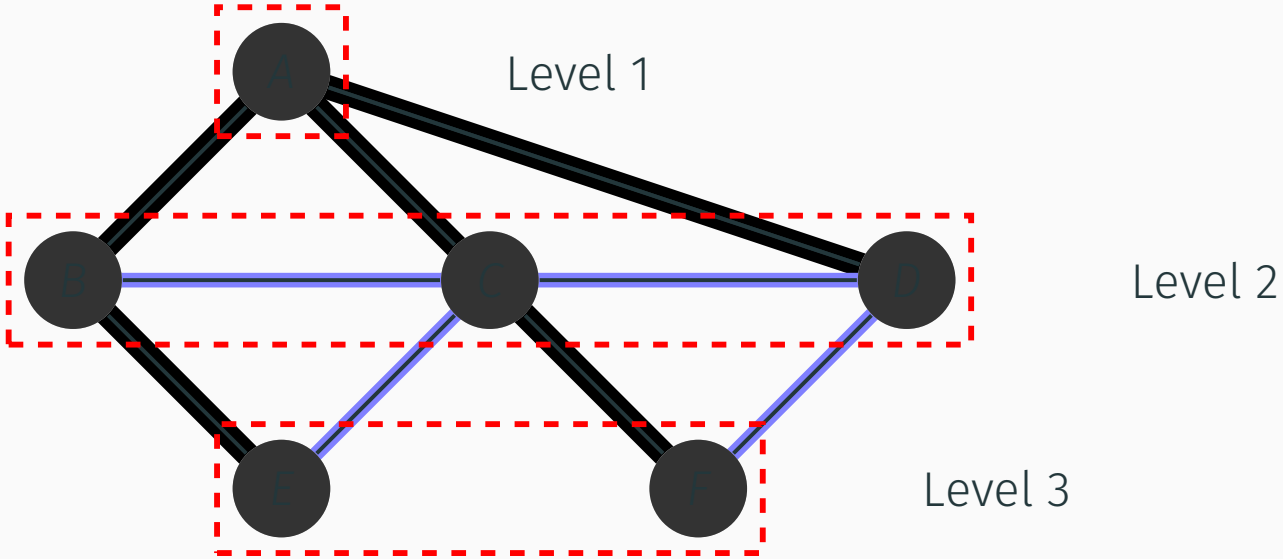
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

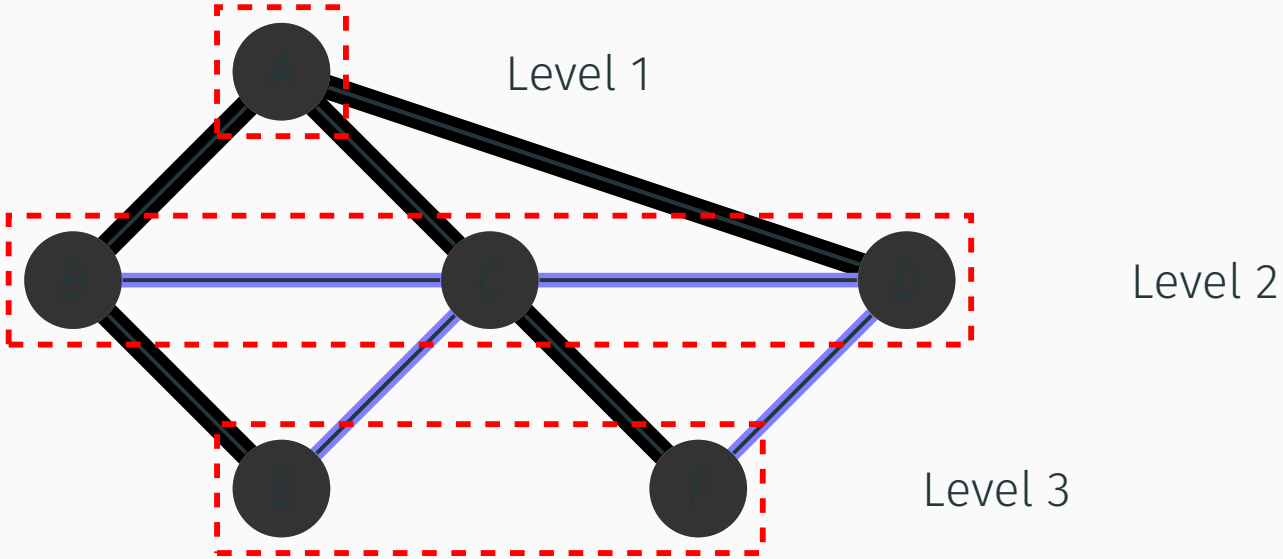
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



Breadth First Search

Keep track of progress by colouring vertices:

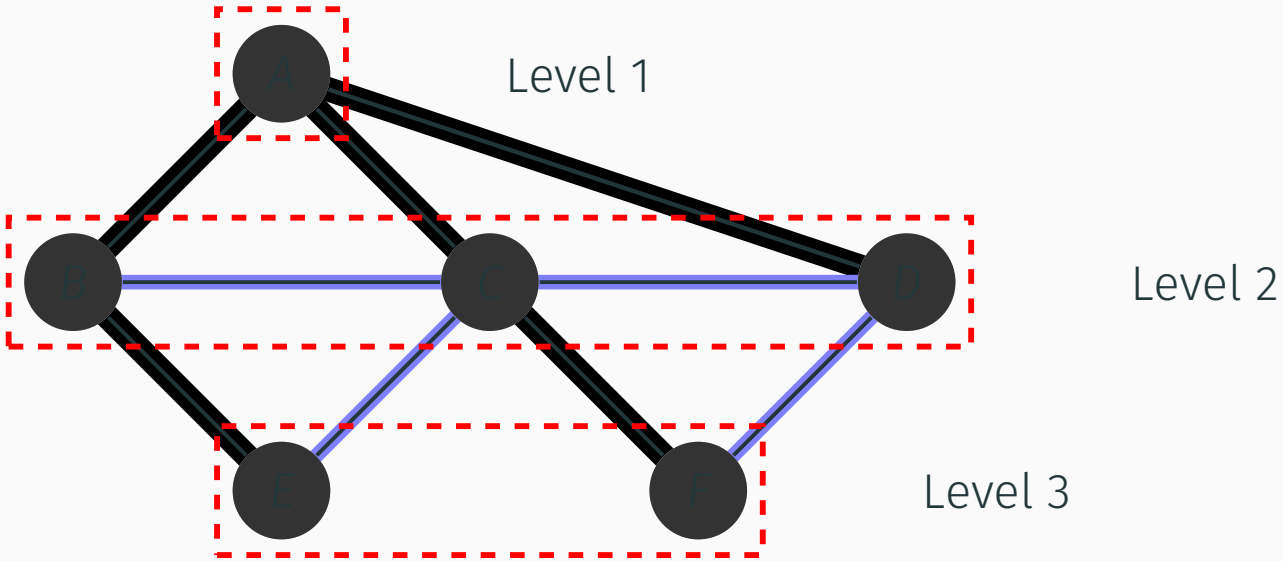
- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished

```
1  BFS(G,s)
2  for each u in V
3    u.color = GRAY // initialize vertex
4  s.color = RED
5  Q.enqueue(s)
6  while Q is not empty
7    u = Q.dequeue()
8    for each v in u.Adj
9      if v.color = GRAY
10       v.color =  red
11       Q.enqueue(v)
12   u.color = BLACK
```

Breadth First Search

Keep track of progress by colouring vertices:

- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished

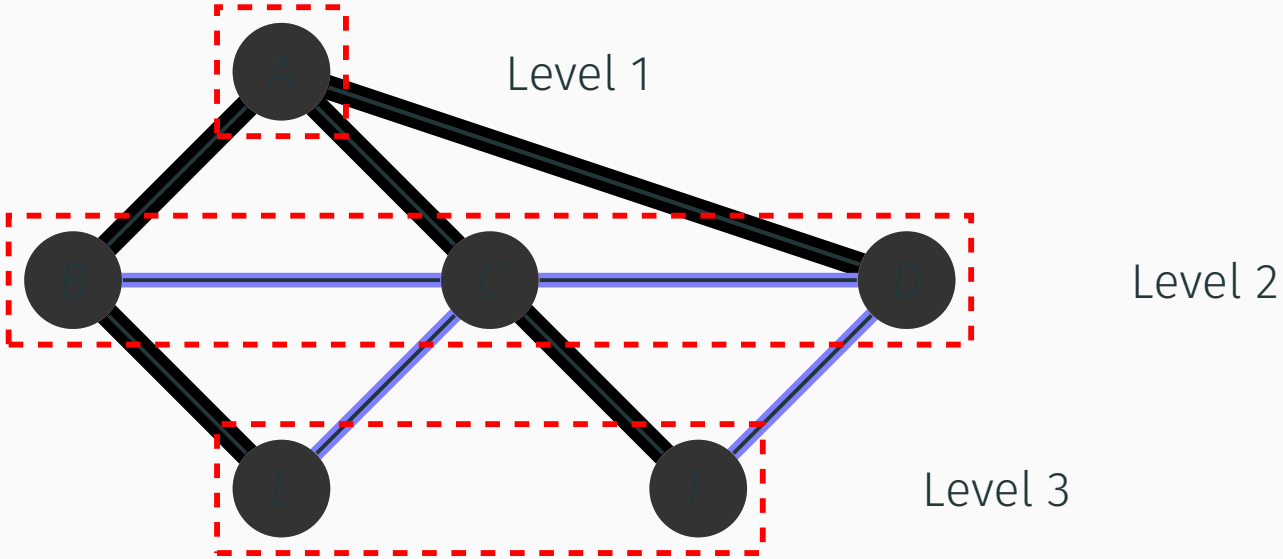


Running time: $O(n + m)$ (assuming adjacency list)

Breadth First Search

Keep track of progress by colouring vertices:

- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished

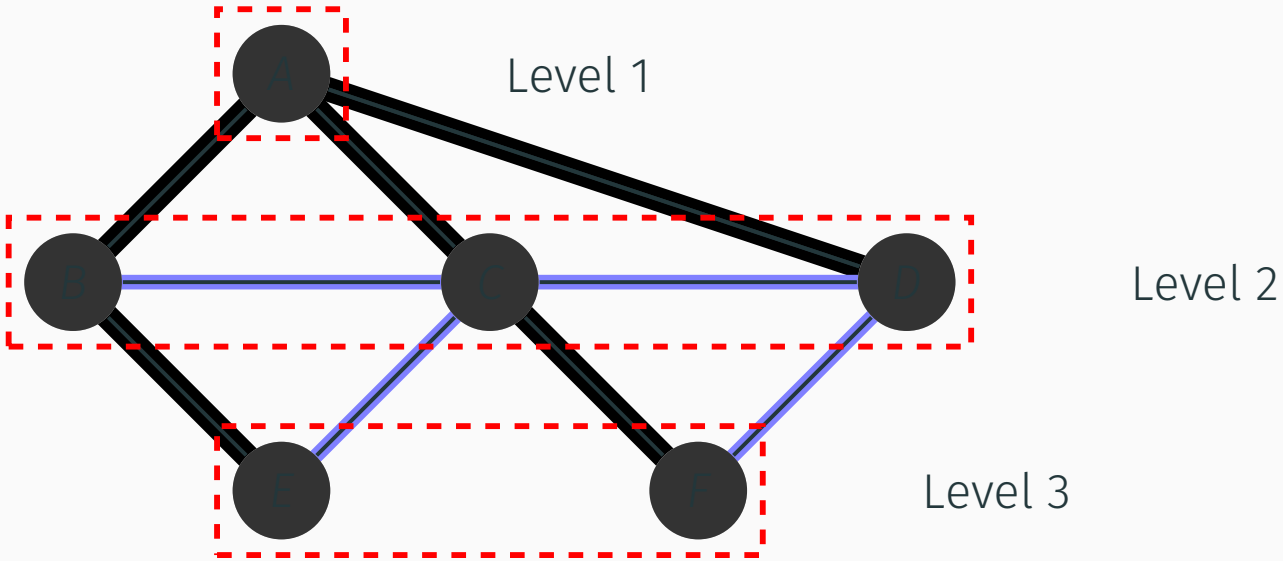


BFS(G,s) visits all the vertices and edges in the connected component of *s*

Breadth First Search

Keep track of progress by colouring vertices:

- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished

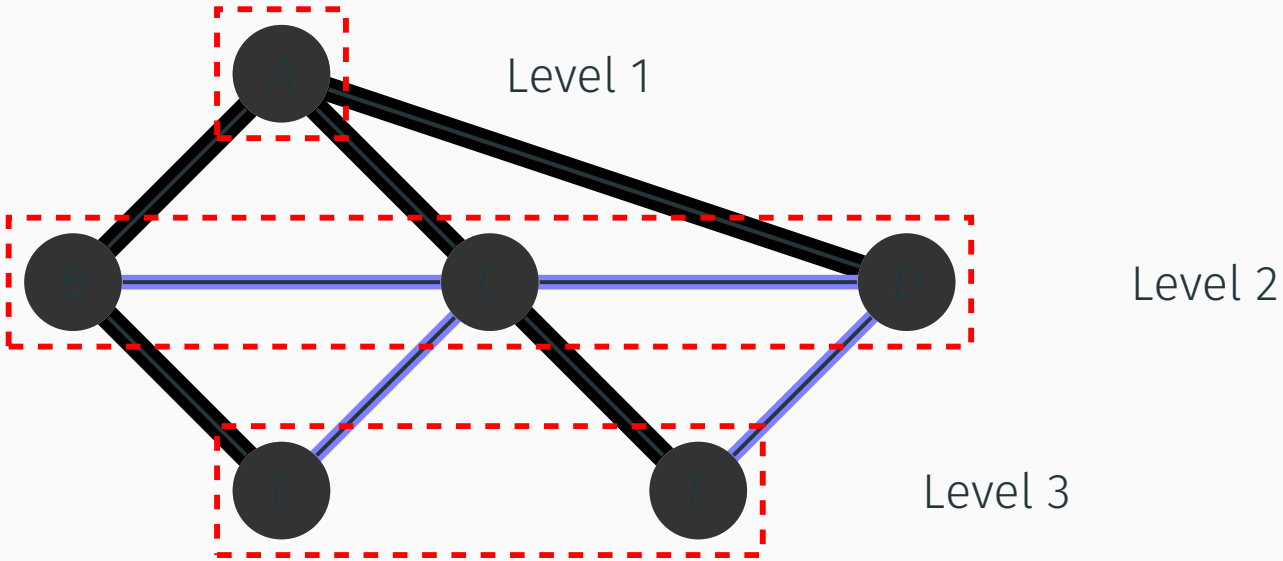


The discovery edges labeled by *BFS(G,s)* form a spanning tree of the connected component of *s*

Breadth First Search

Keep track of progress by colouring vertices:

- **Gray**: undiscovered vertices
- **Red**: discovered, but not finished
- **Black**: finished



For each vertex v in Level i, shortest distance between s and v is i

Application: Shortest Paths on an Unweighted Graph

```
1  BFS(G,s)
2
3  for each u in V  // initialize vertex
4    u.d = ∞
5    // record the shortest distance between s and u
6
7    u.p = null
8    // record the parent of u on the shortest path from
       s to u
9
10   u.color = GRAY
```

Application: Shortest Paths on an Unweighted Graph

```
1  s.color = RED
2  s.d = 0
3  Q.enqueue(s)
4  while Q is not empty
5    u = Q.dequeue()
6    for each v in u.Adj
7      if v.color = GRAY
8        v.color = red
9        v.d = u.d + 1
10       v.p = u
11       Q.enqueue(v)
12   u.color = BLACK
```

Application: Shortest Paths on an Unweighted Graph

```
1  s.color = RED
2  s.d = 0
3  Q.enqueue(s)
4  while Q is not empty
5    u = Q.dequeue()
6    for each v in u.Adj
7      if v.color = GRAY
8        v.color = red
9        v.d = u.d + 1
10       v.p = u
11       Q.enqueue(v)
12   u.color = BLACK
```

Proof of correctness?

Application: Shortest Paths on an Unweighted Graph

```
1  s.color = RED
2  s.d = 0
3  Q.enqueue(s)
4  while Q is not empty
5    u = Q.dequeue()
6    for each v in u.Adj
7      if v.color = GRAY
8        v.color = red
9        v.d = u.d + 1
10       v.p = u
11       Q.enqueue(v)
12   u.color = BLACK
```

Proof of correctness?

A key property: Any subpath of a shortest path is a shortest path

Shortest Path

Shortest Path on Weighted Graphs

For unweighted graph, BFS finds the shortest paths from a source node s to every vertex v in the graph.

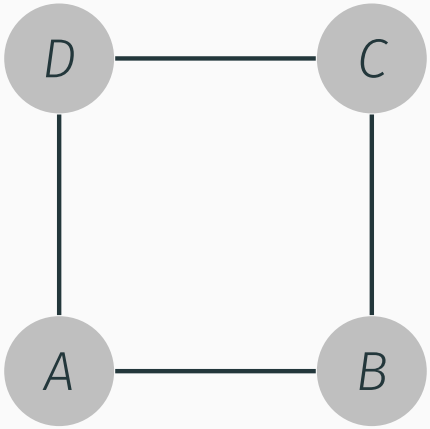
Here, the length of a path is simply the number of edges on the path.

Shortest Path on Weighted Graphs

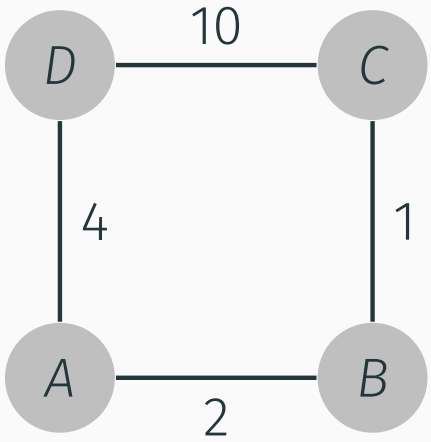
For unweighted graph, BFS finds the shortest paths from a source node s to every vertex v in the graph.

Here, the length of a path is simply the number of edges on the path.

But what if edges have different "costs"?



Shortest path from A to C:
A-B-C or A-D-C



Shortest path from A to C:
only A-B-C

Shortest Path Properties

Optimal Substructure

A subpath of a shortest path is itself a shortest path

Shortest Path Properties

Optimal Substructure

A subpath of a shortest path is itself a shortest path

Shortest Path Tree

There is a tree of shortest paths from a start vertex to all the other vertices

Shortest Path Properties

Optimal Substructure

A subpath of a shortest path is itself a shortest path

Shortest Path Tree

There is a tree of shortest paths from a start vertex to all the other vertices

Note that shortest path trees are not necessarily unique.

Shortest Path Properties

Optimal Substructure

A subpath of a shortest path is itself a shortest path

Shortest Path Tree

There is a tree of shortest paths from a start vertex to all the other vertices

Note that shortest path trees are not necessarily unique.

Consider negative cycle?

Shortest Path Properties

Optimal Substructure

A subpath of a shortest path is itself a shortest path

Shortest Path Tree

There is a tree of shortest paths from a start vertex to all the other vertices

Note that shortest path trees are not necessarily unique.

Consider negative cycle?

Let's **assume** there is no negative edges.

Shortest Path

Dijkstra's Algorithm

Dijkstra's algorithm

Invented by *E. Dijkstra* in 1959.

- Applies to general, weighted, directed or undirected graph (may contain cycles).
- But weights must be non-negative. (But they can be 0!)

Dijkstra's algorithm

Invented by *E. Dijkstra* in 1959.

- Applies to general, weighted, directed or undirected graph (may contain cycles).
- But weights must be non-negative. (But they can be 0!)

Recall BFS:

Dijkstra's algorithm

Invented by *E. Dijkstra* in 1959.

- Applies to general, weighted, directed or undirected graph (may contain cycles).
- But weights must be non-negative. (But they can be 0!)

Recall BFS:

```
1  BFS(G,s)
2  for each u in V  // initialize vertex
3    u.d = ∞
4   // record the shortest distance between s and u
5    u.p = null
6   // record the parent of u on the shortest path from s
       to u
7    u.color = GRAY
```

Dijkstra's algorithm

Invented by *E. Dijkstra* in 1959.

- Applies to general, weighted, directed or undirected graph (may contain cycles).
- But weights must be non-negative. (But they can be 0!)

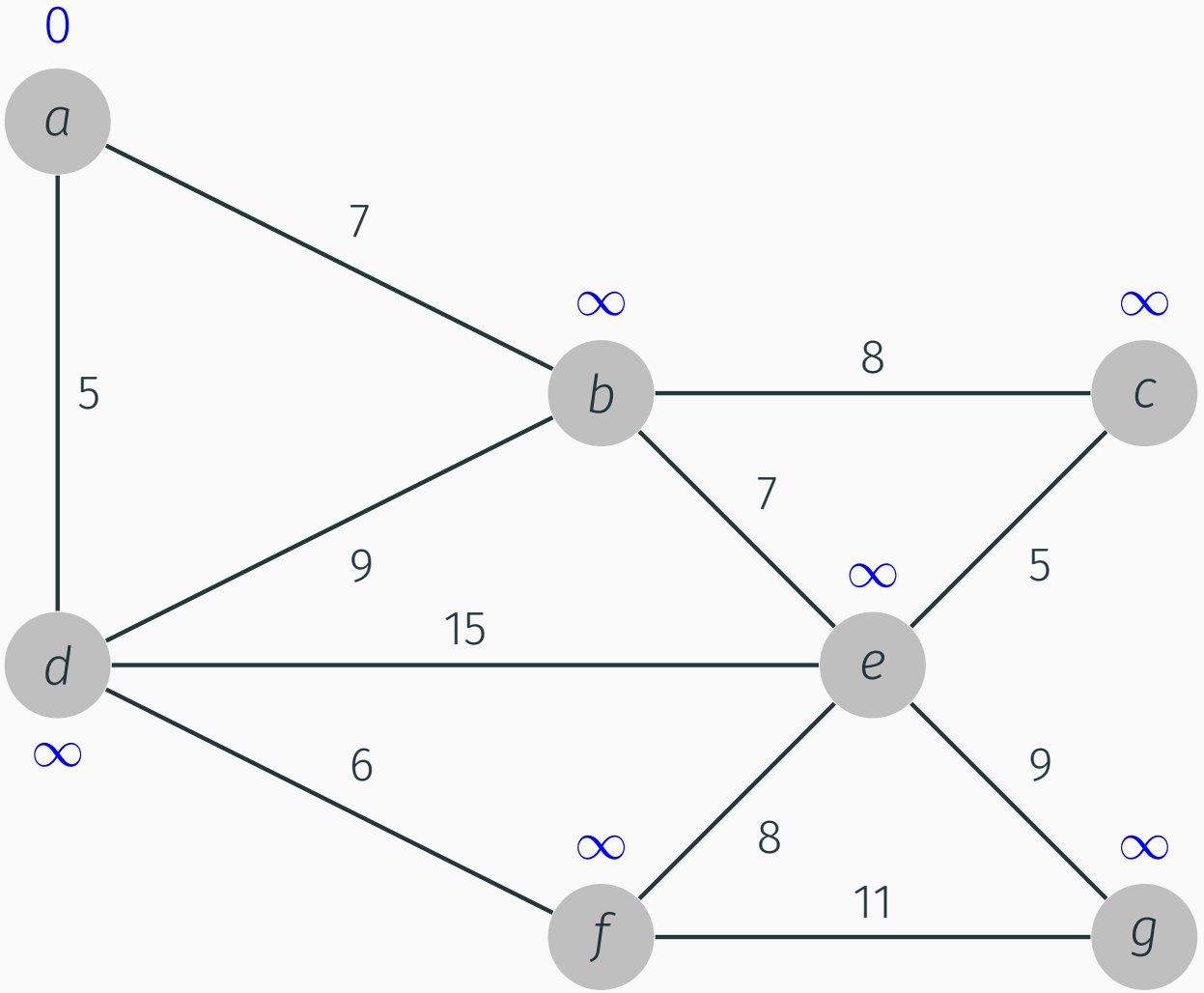
Recall BFS:

```
1  s.color = RED
2  s.d = 0
3  Q.enqueue(s)
```

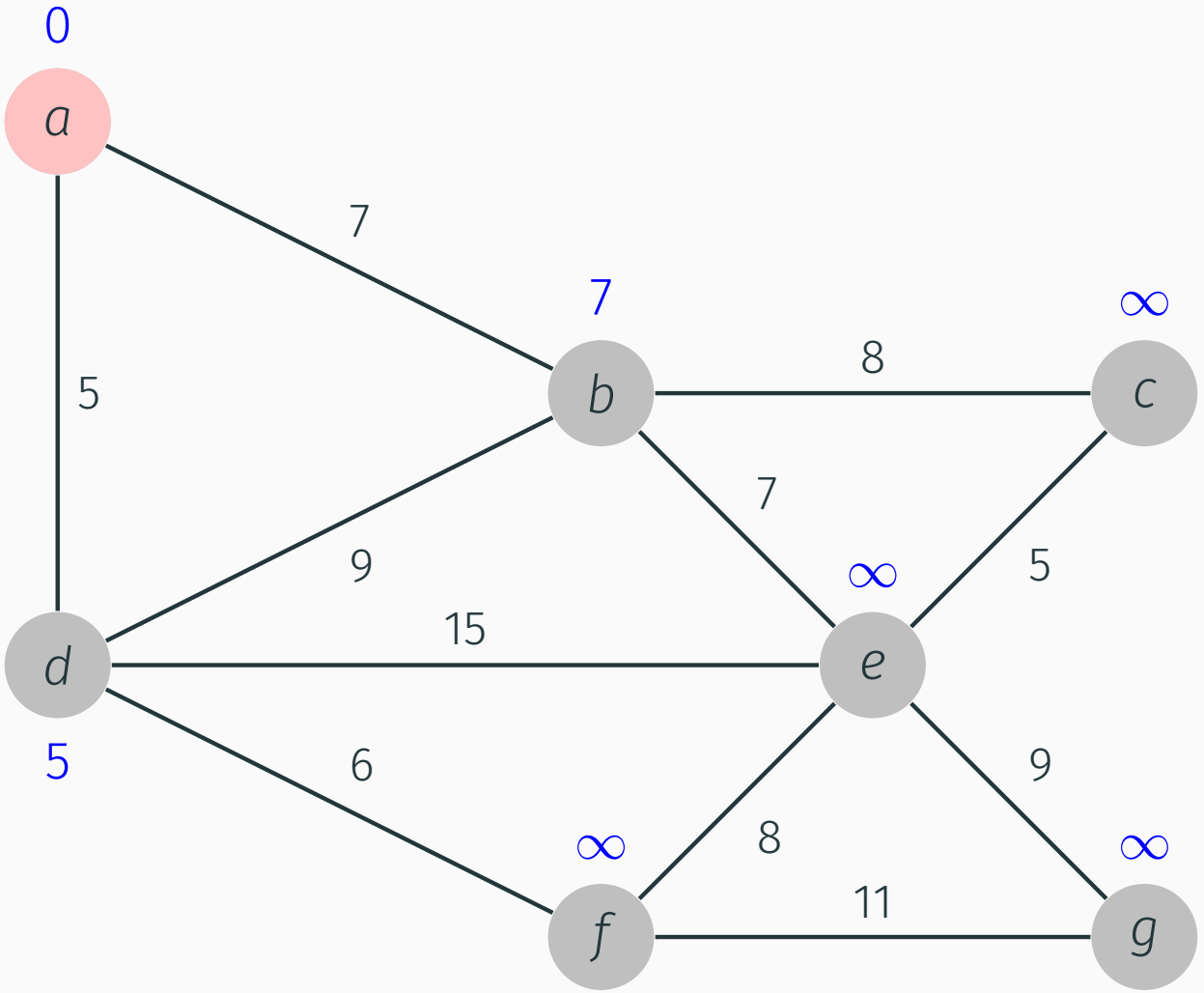
- Search as wide as possible
- **Use a Queue – FIFO**

```
1  while Q is not empty
2    u = Q.dequeue()
3    for each v in u.Adj
4      if v.color = GRAY
5        v.color = red
6        v.d = u.d + 1
7        v.p = u
8        Q.enqueue(v)
9    u.color = BLACK
```

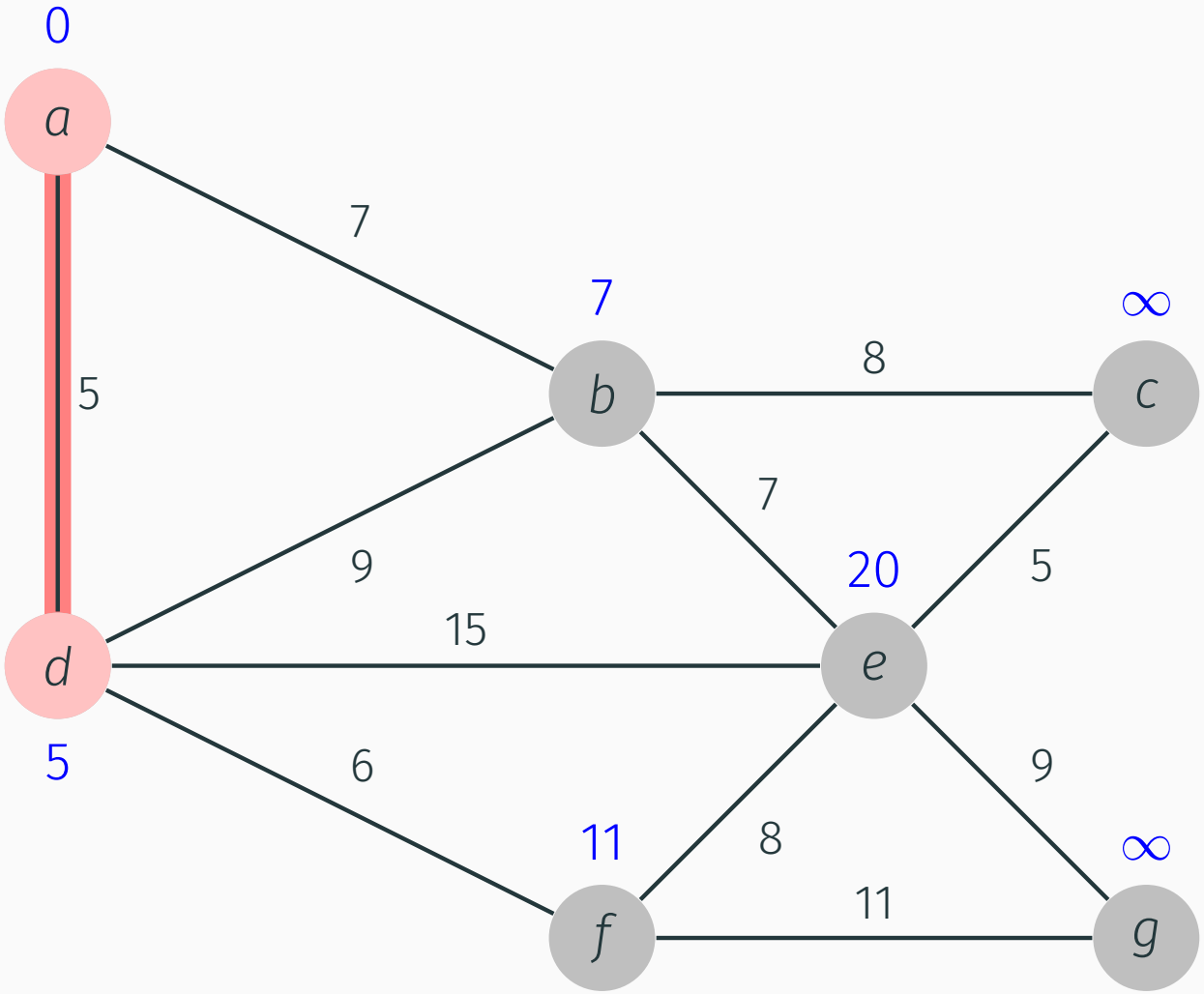
Dijkstra's algorithm: Example



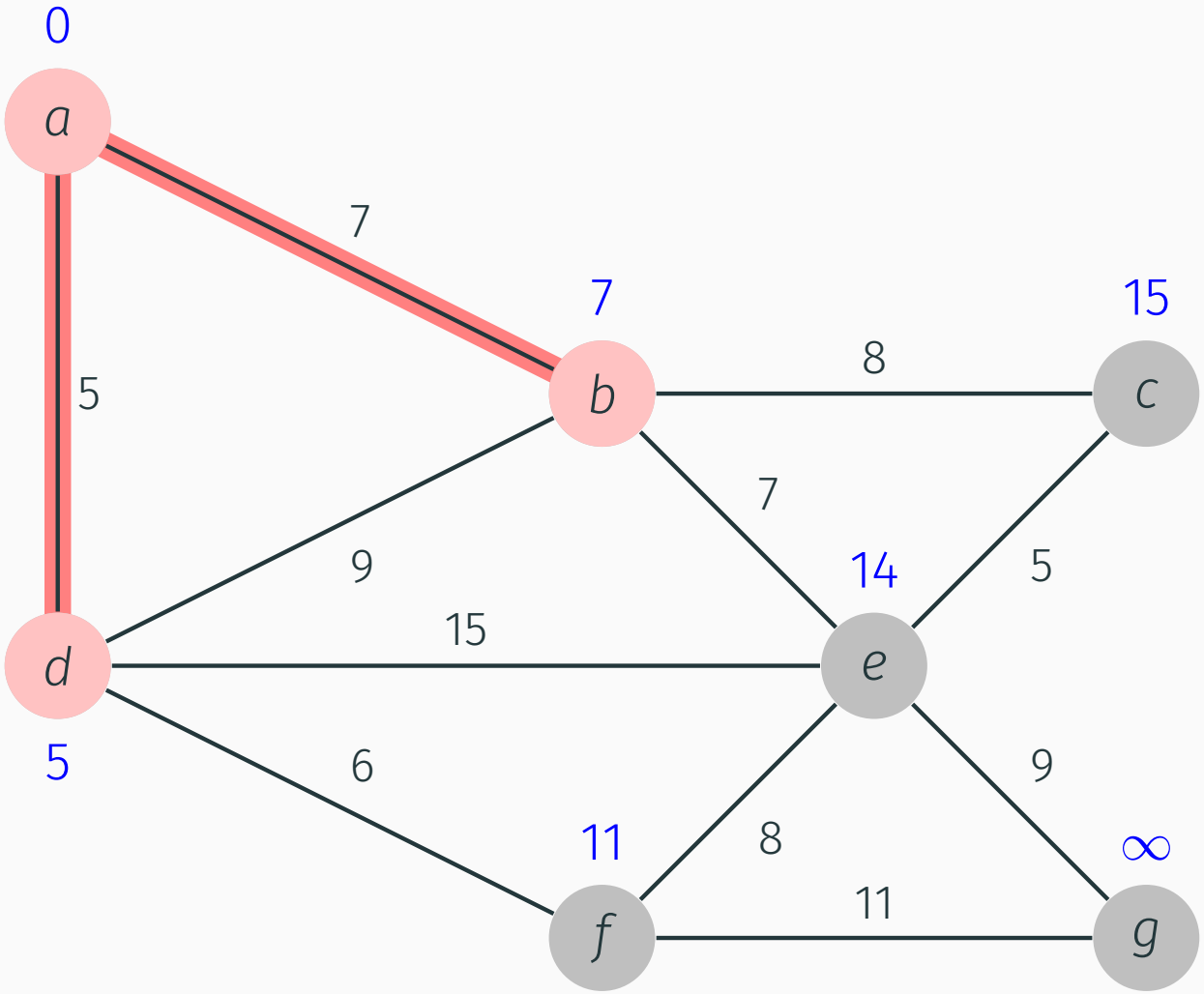
Dijkstra's algorithm: Example



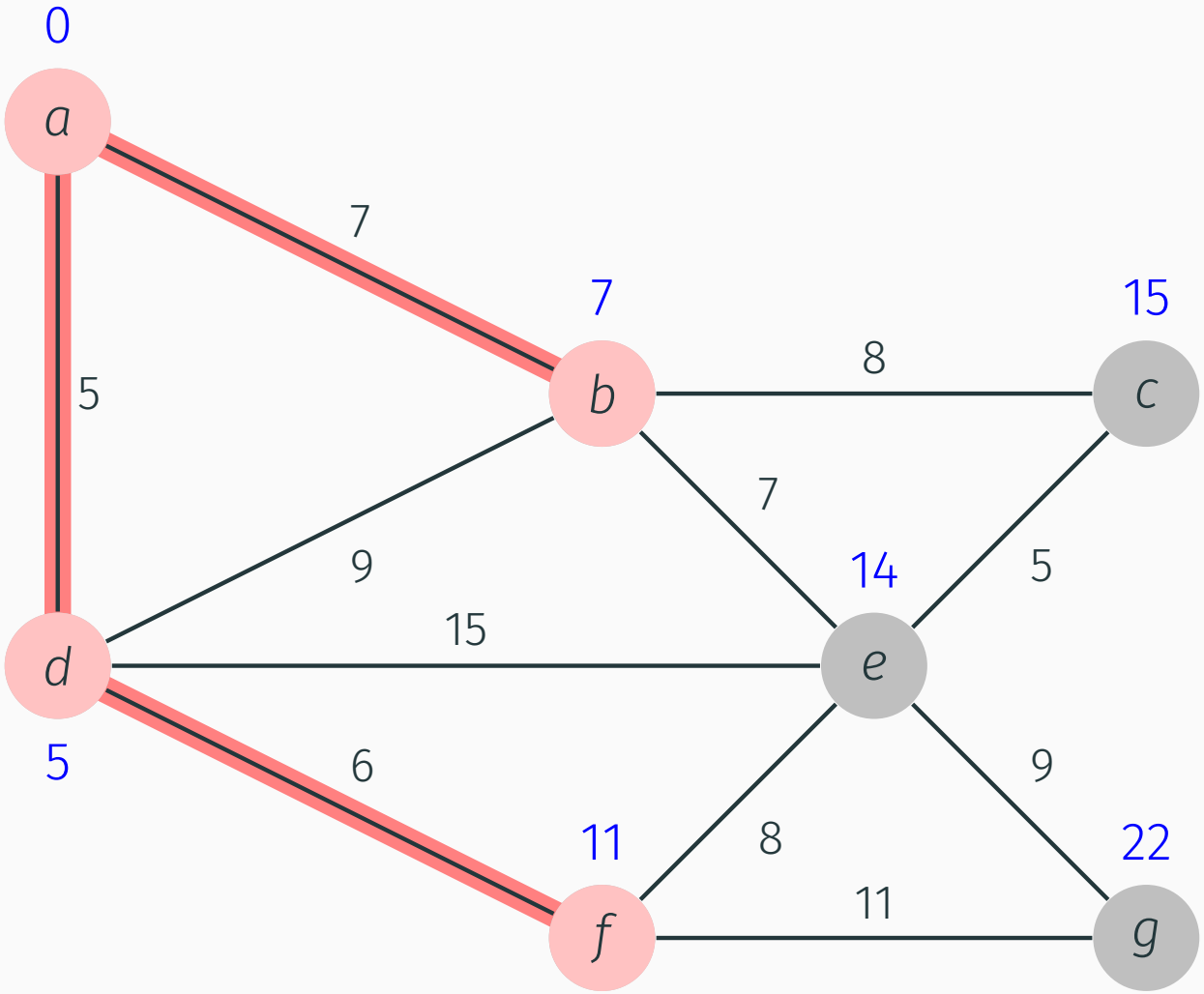
Dijkstra's algorithm: Example



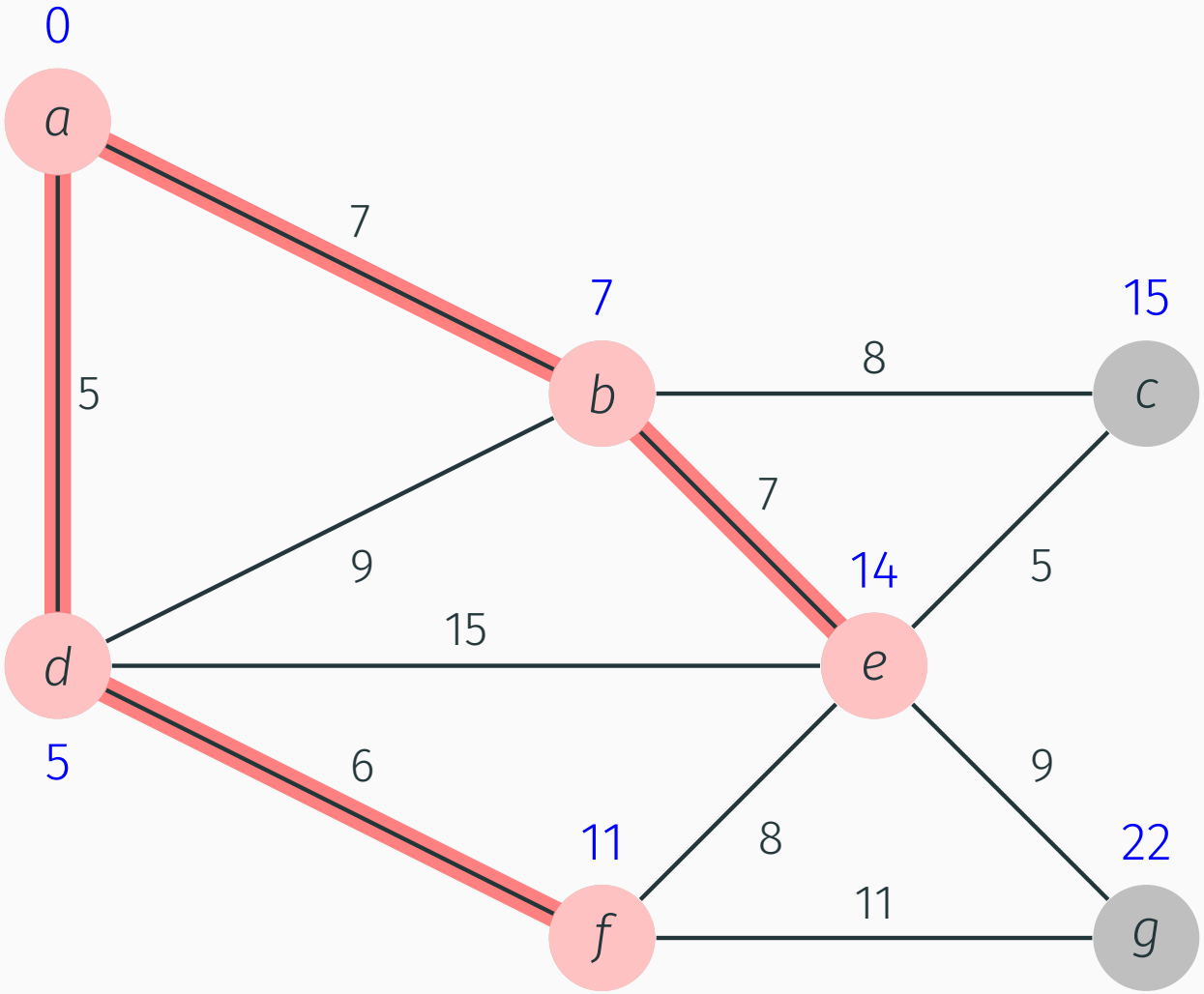
Dijkstra's algorithm: Example



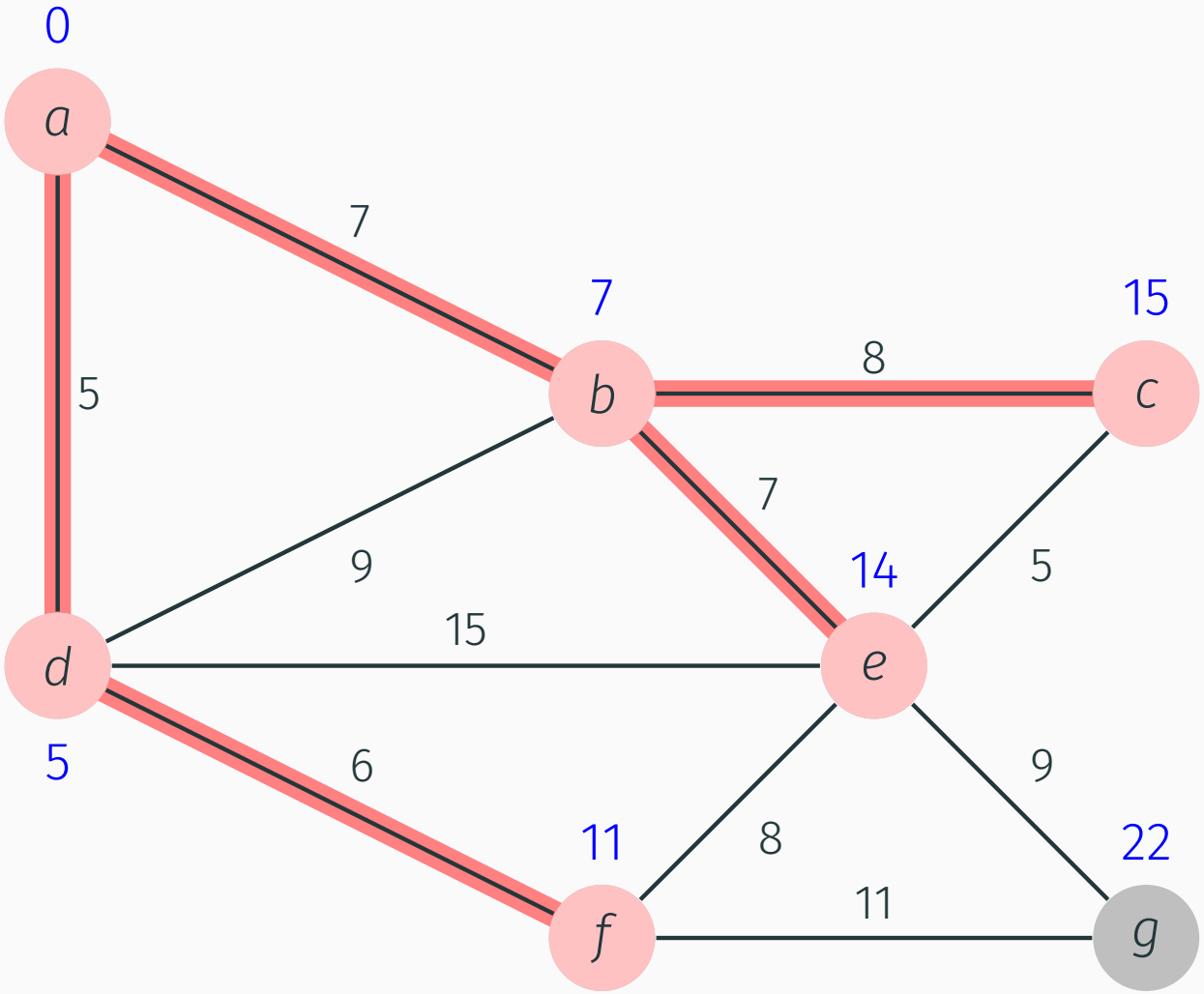
Dijkstra's algorithm: Example



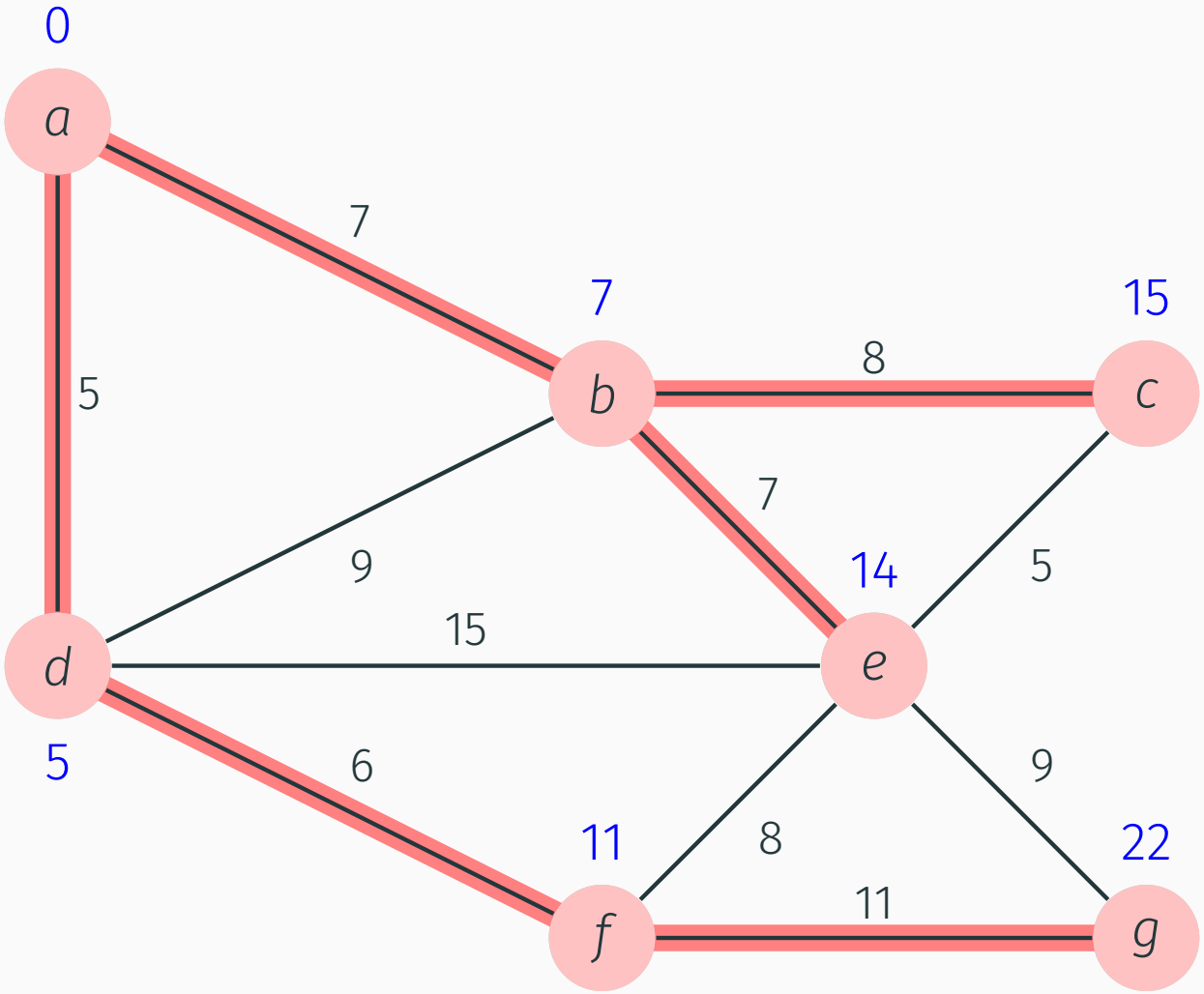
Dijkstra's algorithm: Example



Dijkstra's algorithm: Example



Dijkstra's algorithm: Example



Dijkstra's algorithm

```
initialize(G,s)

for each v in V
  v.d = ∞
  d.p = null
s.d = 0
```

```
relax(u, v, w)

if v.d > u.d + w(u,v)
  v.d = u.d + w(u,v)
  v.p = u
```

```
initialize(G,s)
S = ∅
Q = V
while Q is not empty
  u = Extract-Min(Q)
  S = S ∪ u
  for each v in u.Adj
    relax(u, v, w)
```

Dijkstra's algorithm

```
1  initialize(G,s)
2
3  for each v in V
4    v.d = ∞
5    d.p = null
6  s.d = 0
```

```
1  relax(u, v, w)
2
3  if v.d > u.d + w(u,v)
4    v.d = u.d + w(u,v)
5    v.p = u
```

```
1  initialize(G,s)
2  S = ∅
3  Q = V
4  while Q is not empty
5    u = Extract-Min(Q)
6    S = S ∪ u
7    for each v in u.Adj
8      relax(u, v, w)
```

Use a Priority Queue

Running time: $O(m \log n)$. **Why?**

Dijkstra's algorithm

```
1  initialize(G,s)
2
3  for each v in V
4    v.d = ∞
5    d.p = null
6  s.d = 0
```

```
1  relax(u, v, w)
2
3  if v.d > u.d + w(u,v)
4    v.d = u.d + w(u,v)
5    v.p = u
```

```
1  initialize(G,s)
2  S = ∅
3  Q = V
4  while Q is not empty
5    u = Extract-Min(Q)
6    S = S ∪ u
7    for each v in u.Adj
8      relax(u, v, w)
```

Use a Priority Queue

Running time: $O(m \log n)$. **Why?**

Proof of correctness?

Minimum Spanning Tree

Minimum Spanning Tree

A Minimum Spanning Tree (MST) is a subgraph of an undirected graph such that the subgraph spans (includes) all nodes, is connected, is acyclic, and has minimum total edge weight.

Minimum Spanning Tree

A Minimum Spanning Tree (MST) is a subgraph of an undirected graph such that the subgraph spans (includes) all nodes, is connected, is acyclic, and has minimum total edge weight.

Greedy idea?

Minimum Spanning Tree

A Minimum Spanning Tree (MST) is a subgraph of an undirected graph such that the subgraph spans (includes) all nodes, is connected, is acyclic, and has minimum total edge weight.

Greedy idea?

Prim's algorithm and **Kruskal's algorithm**

Both work with weighted and unweighted graphs even when edges have negative weight.

Minimum Spanning Tree

A Minimum Spanning Tree (MST) is a subgraph of an undirected graph such that the subgraph spans (includes) all nodes, is connected, is acyclic, and has minimum total edge weight.

Greedy idea?

Prim's algorithm and **Kruskal's algorithm**

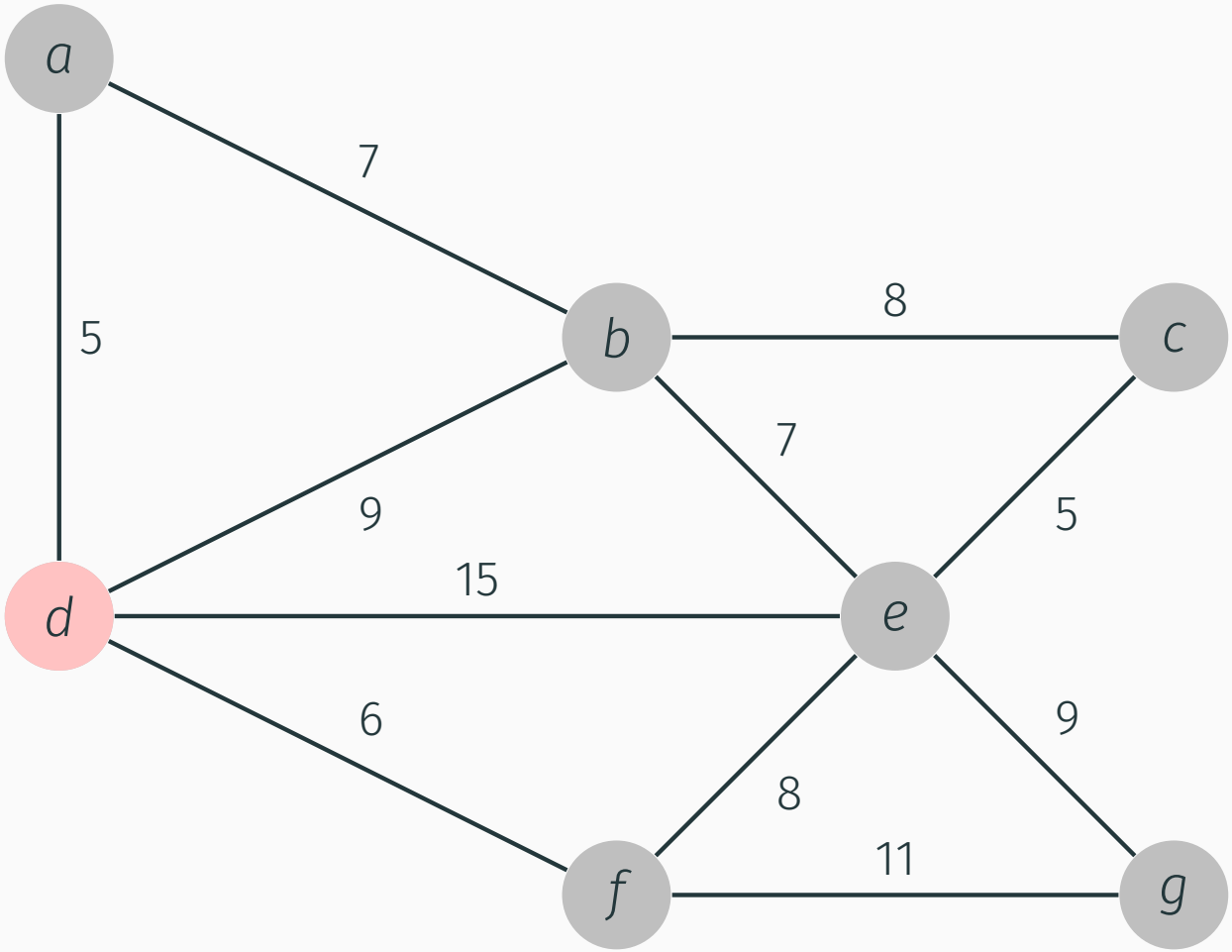
Both work with weighted and unweighted graphs even when edges have negative weight.

Note that: we do not fix the root of the spanning tree.

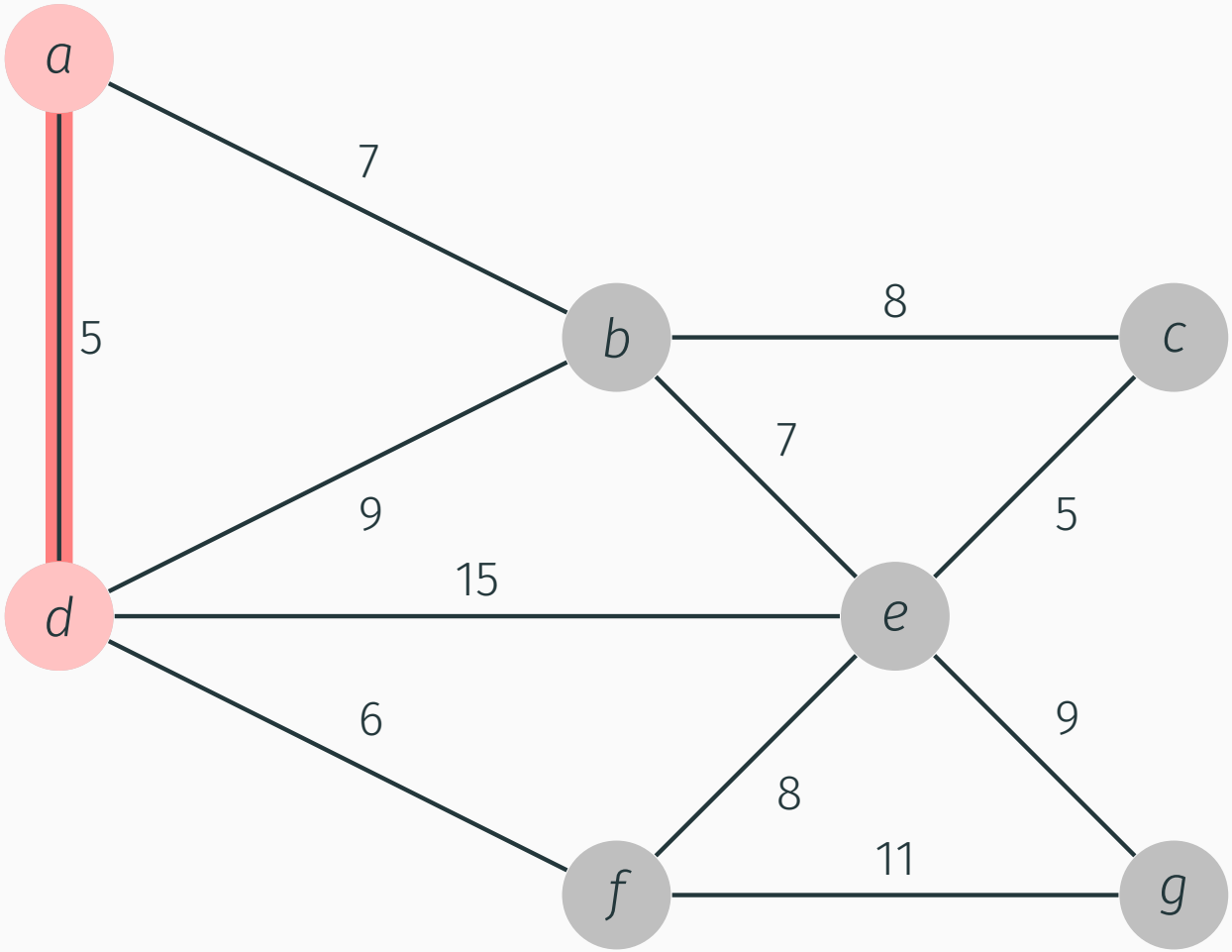
Minimum Spanning Tree

Prim's algorithm

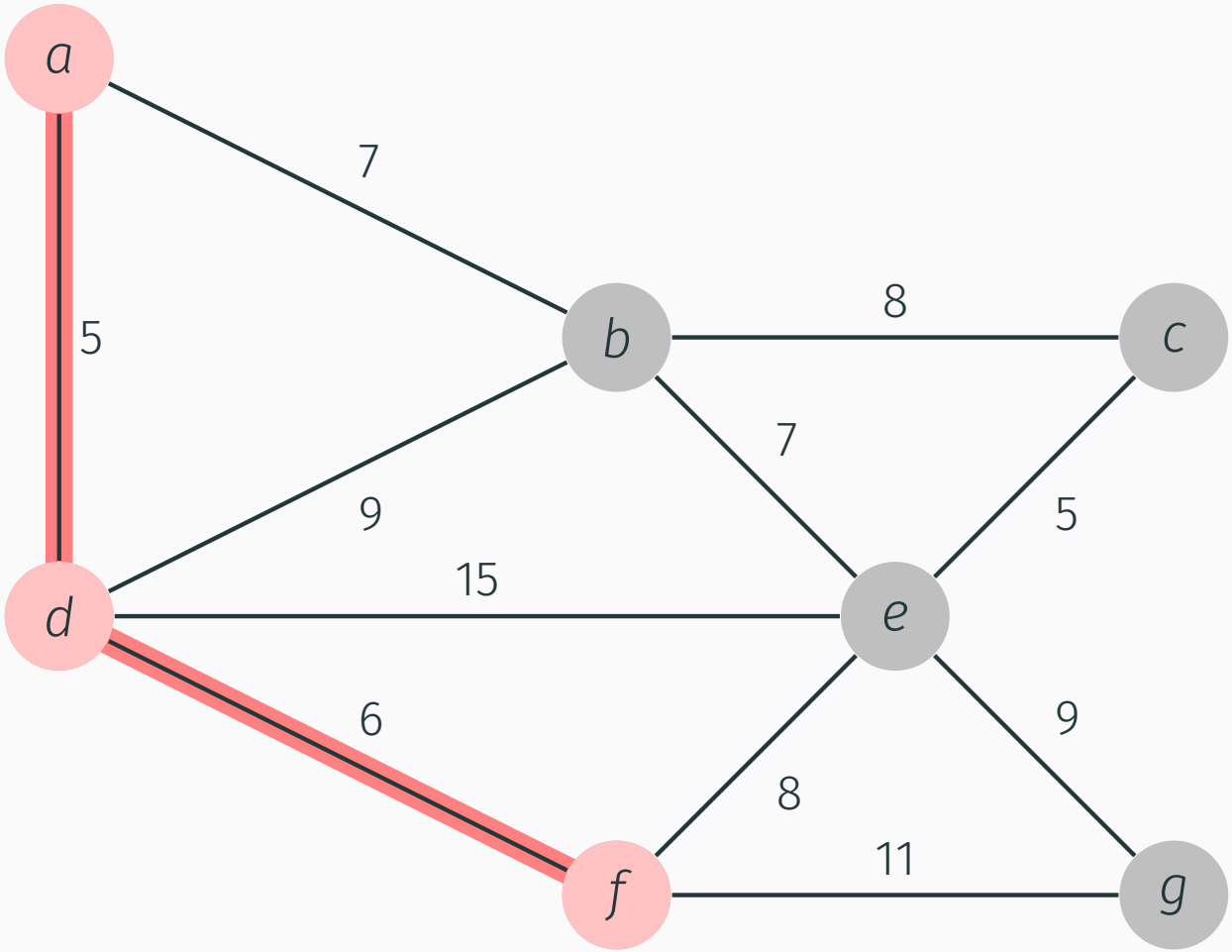
Prim’s algorithm



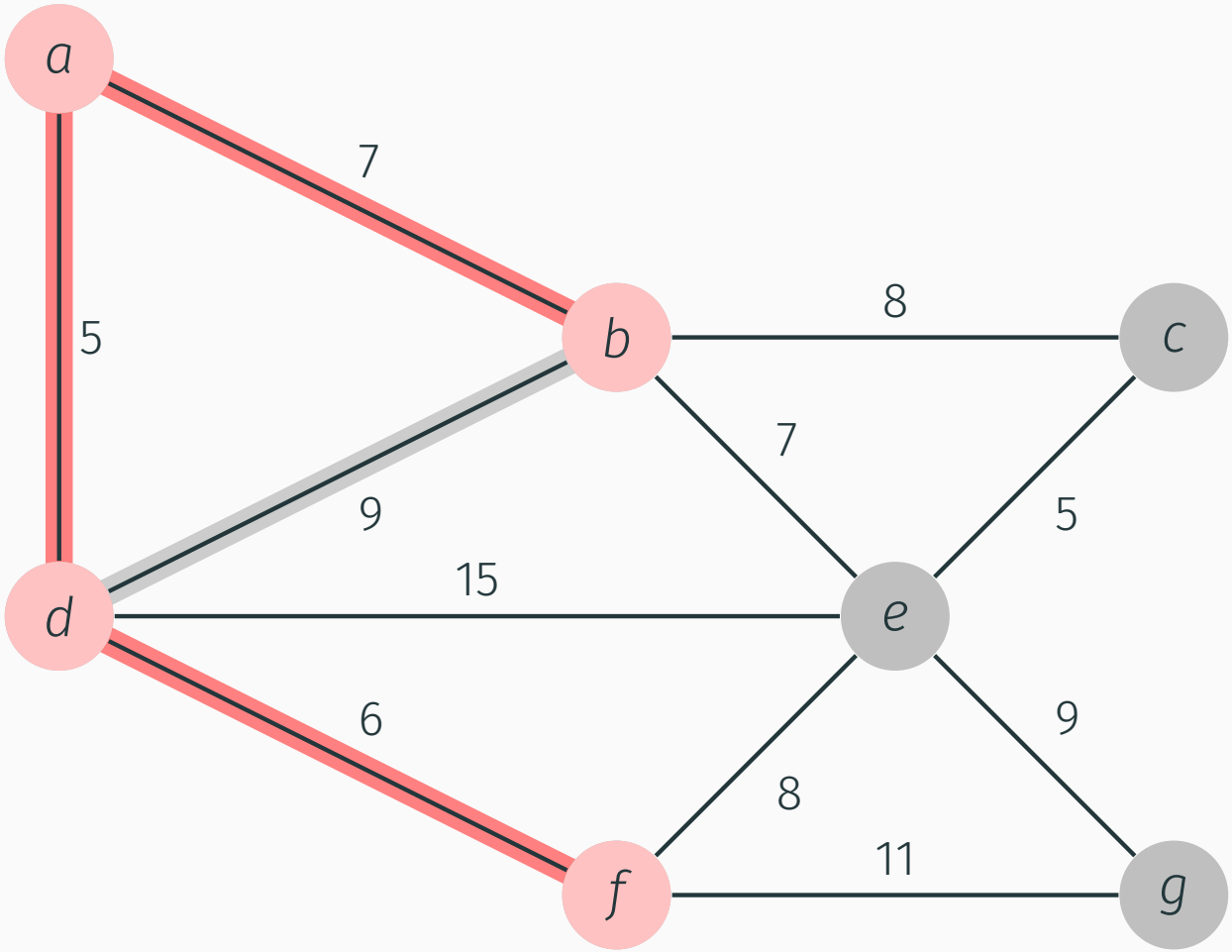
Prim's algorithm



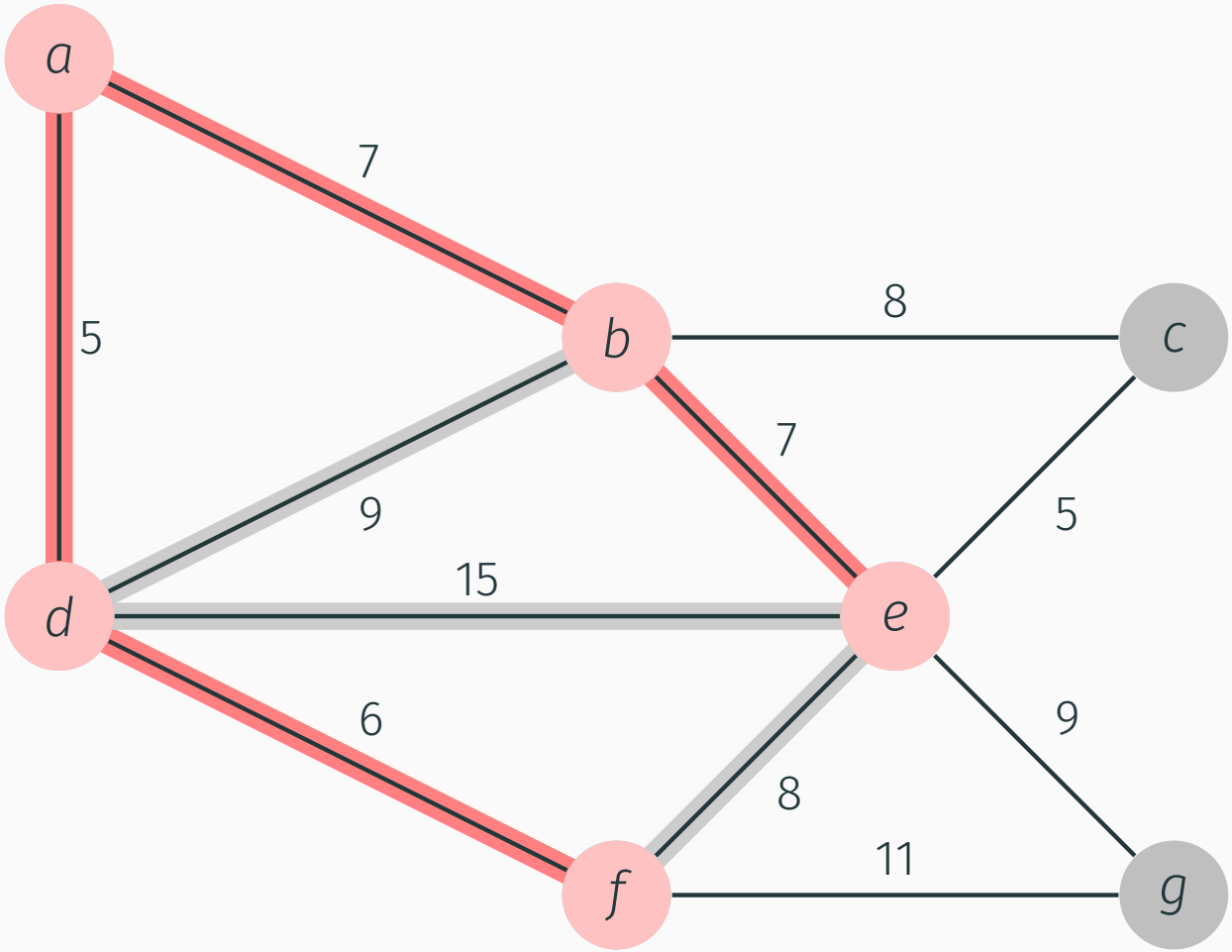
Prim's algorithm



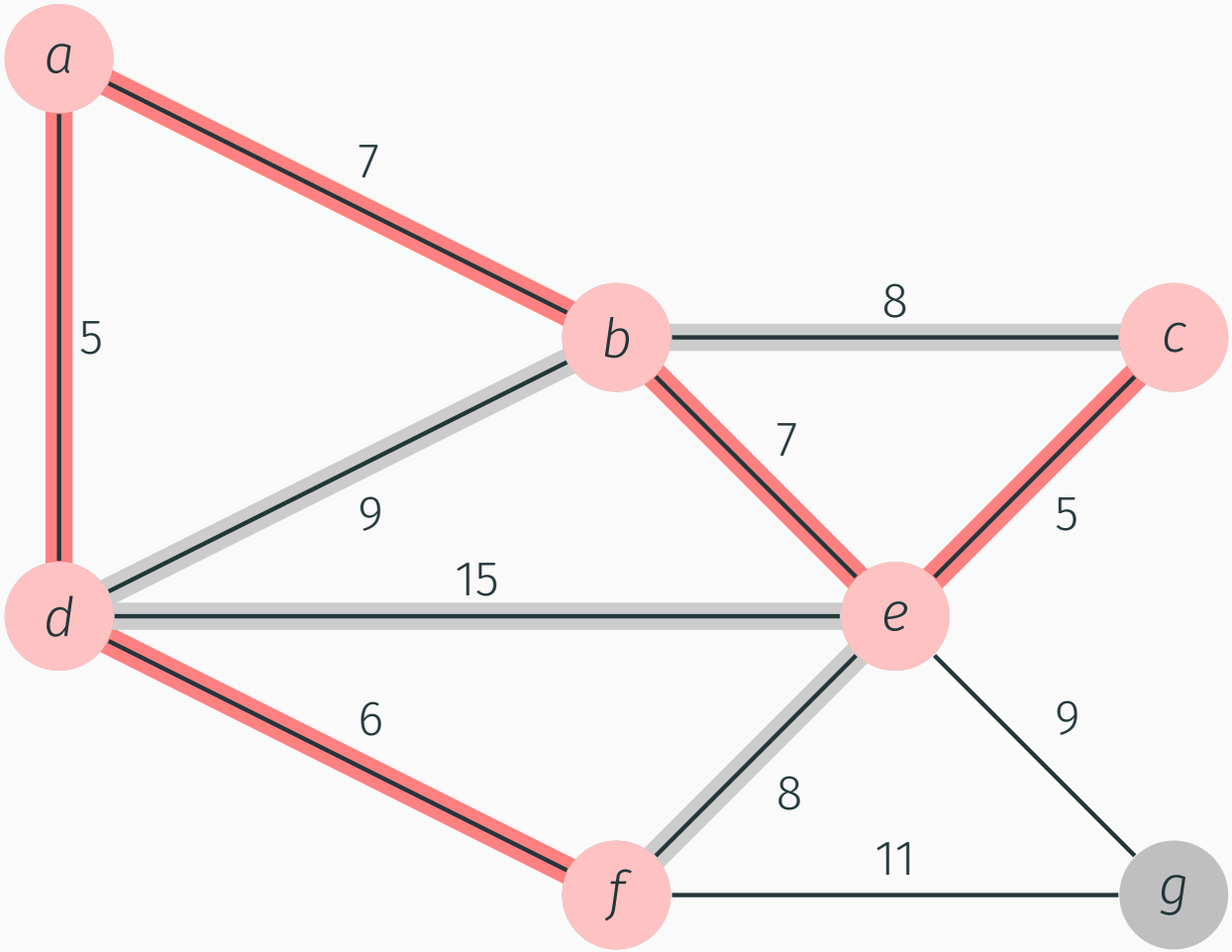
Prim's algorithm



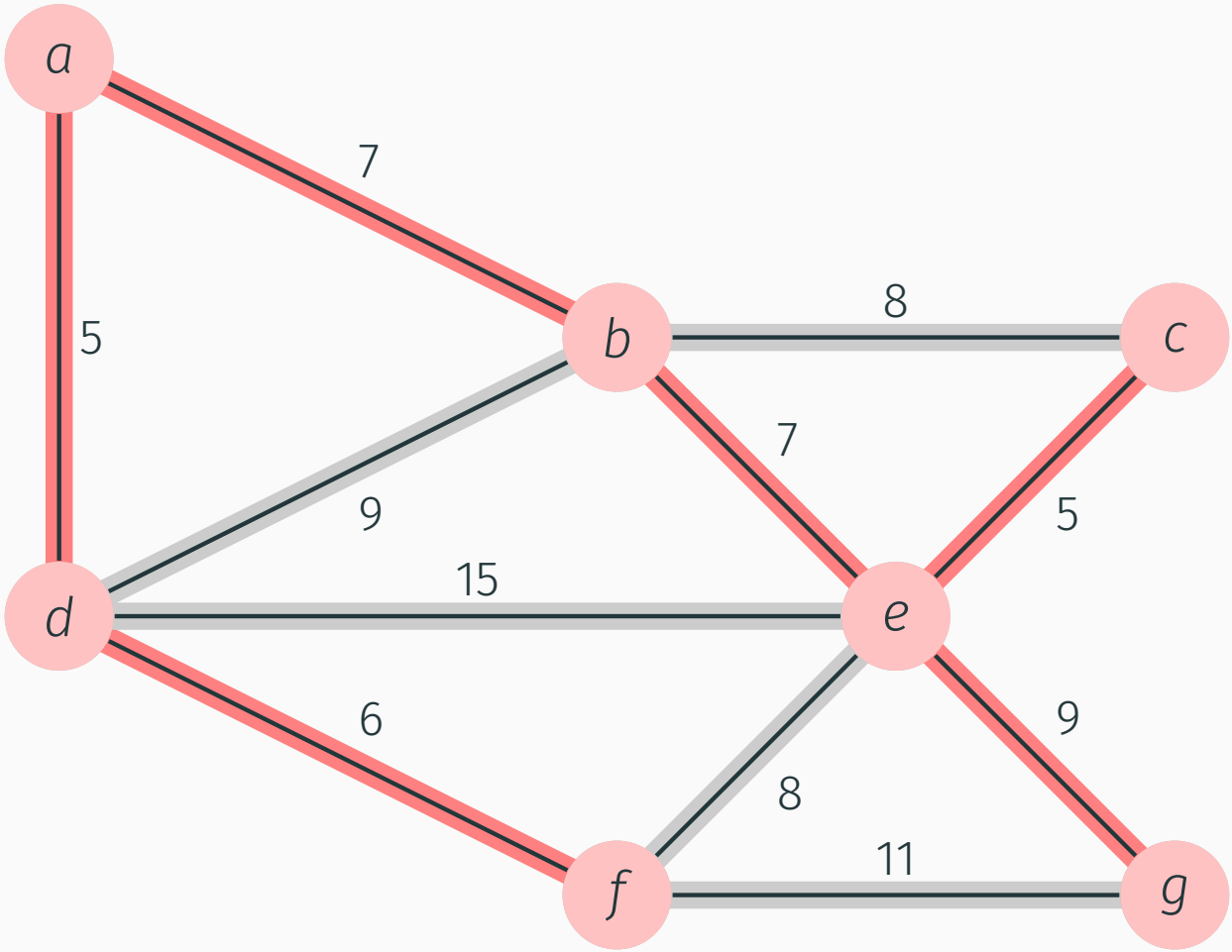
Prim's algorithm



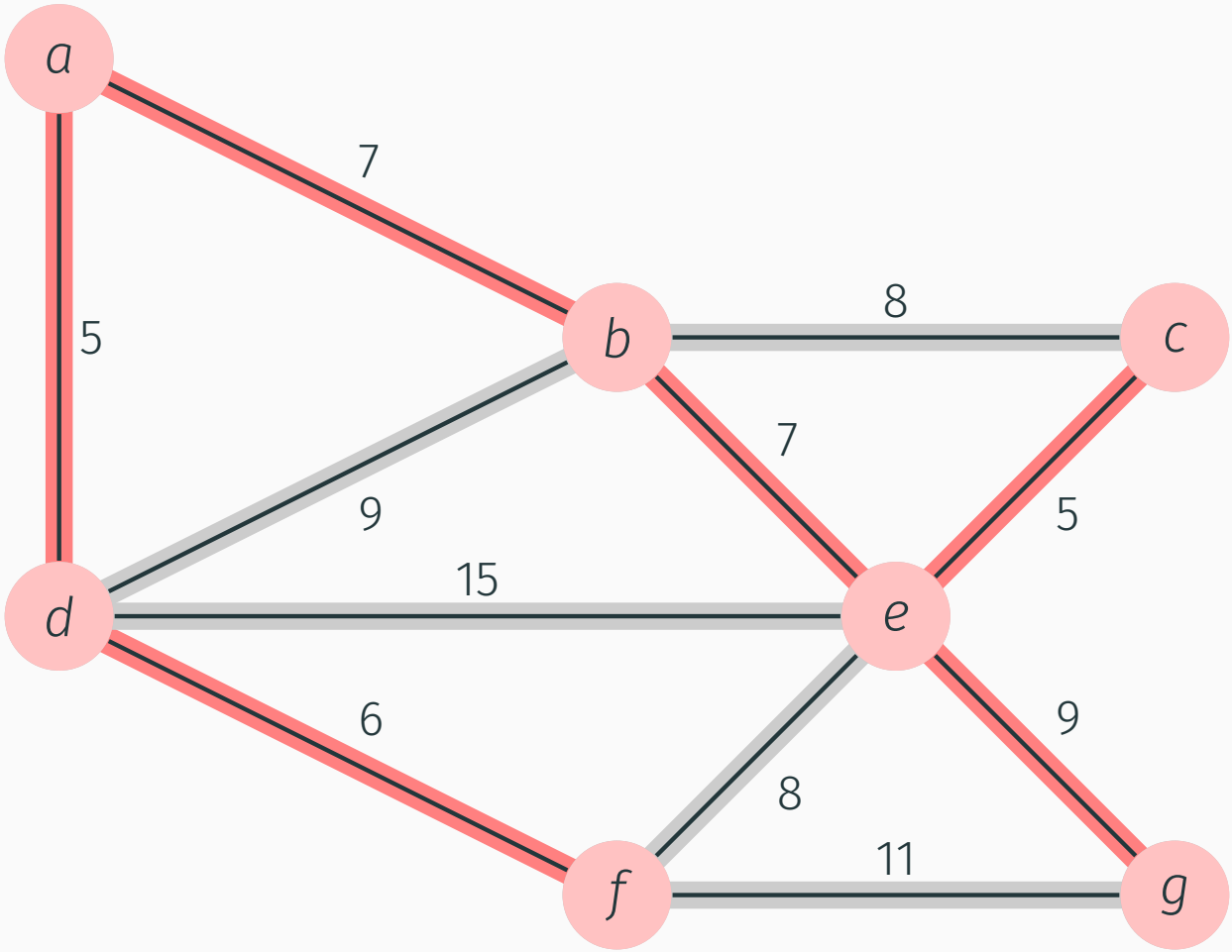
Prim's algorithm



Prim's algorithm

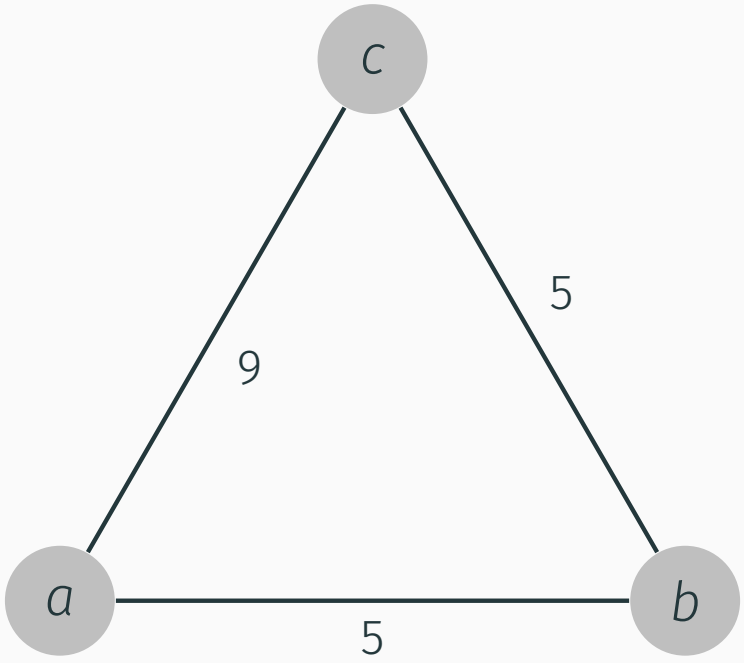


Prim's algorithm



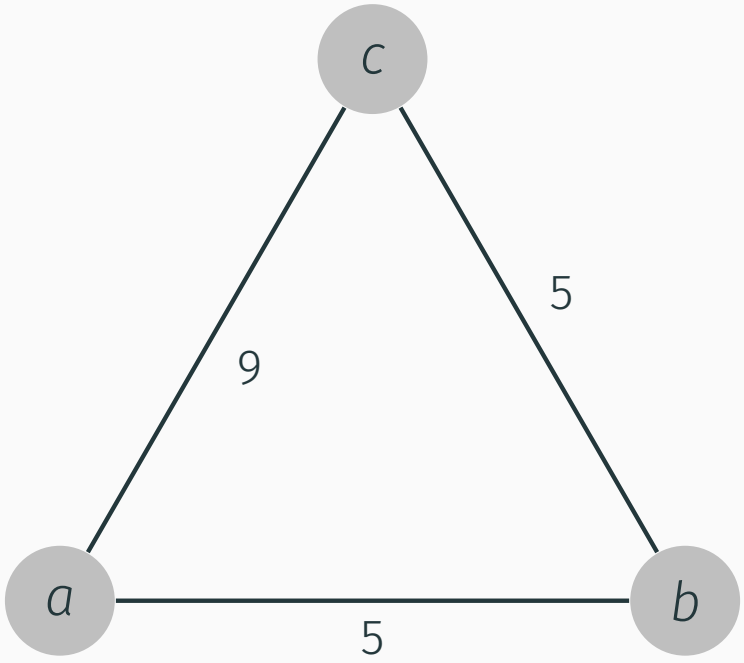
Similar to Dijkstra's Algorithm except that v.d records edge weights, not path lengths. Therefore, Prim's algorithm also has running time $O(m \log n)$

Prim's Algorithm vs Dijkstra's Algorithm



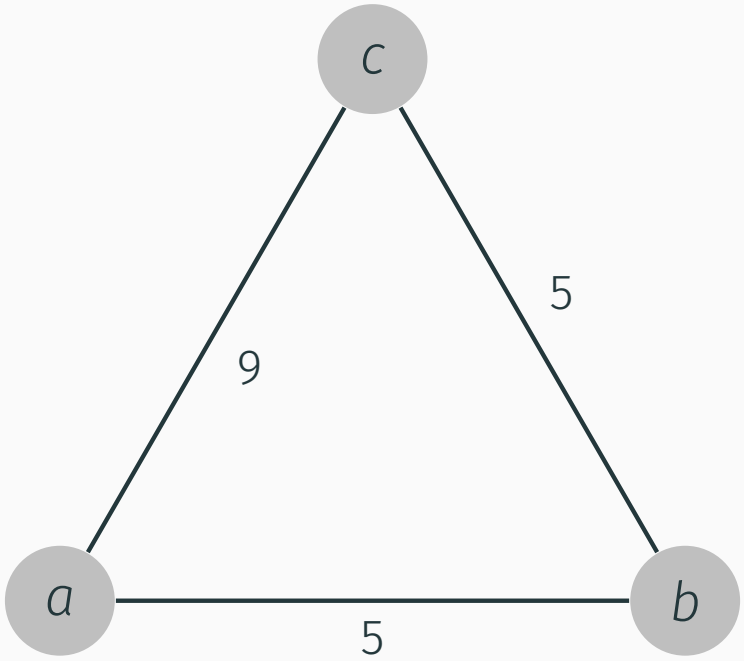
- Shortest path tree rooted at a:
- Minimum spanning tree:

Prim's Algorithm vs Dijkstra's Algorithm



- Shortest path tree rooted at *a*: a-b, a-c
- Minimum spanning tree:

Prim's Algorithm vs Dijkstra's Algorithm



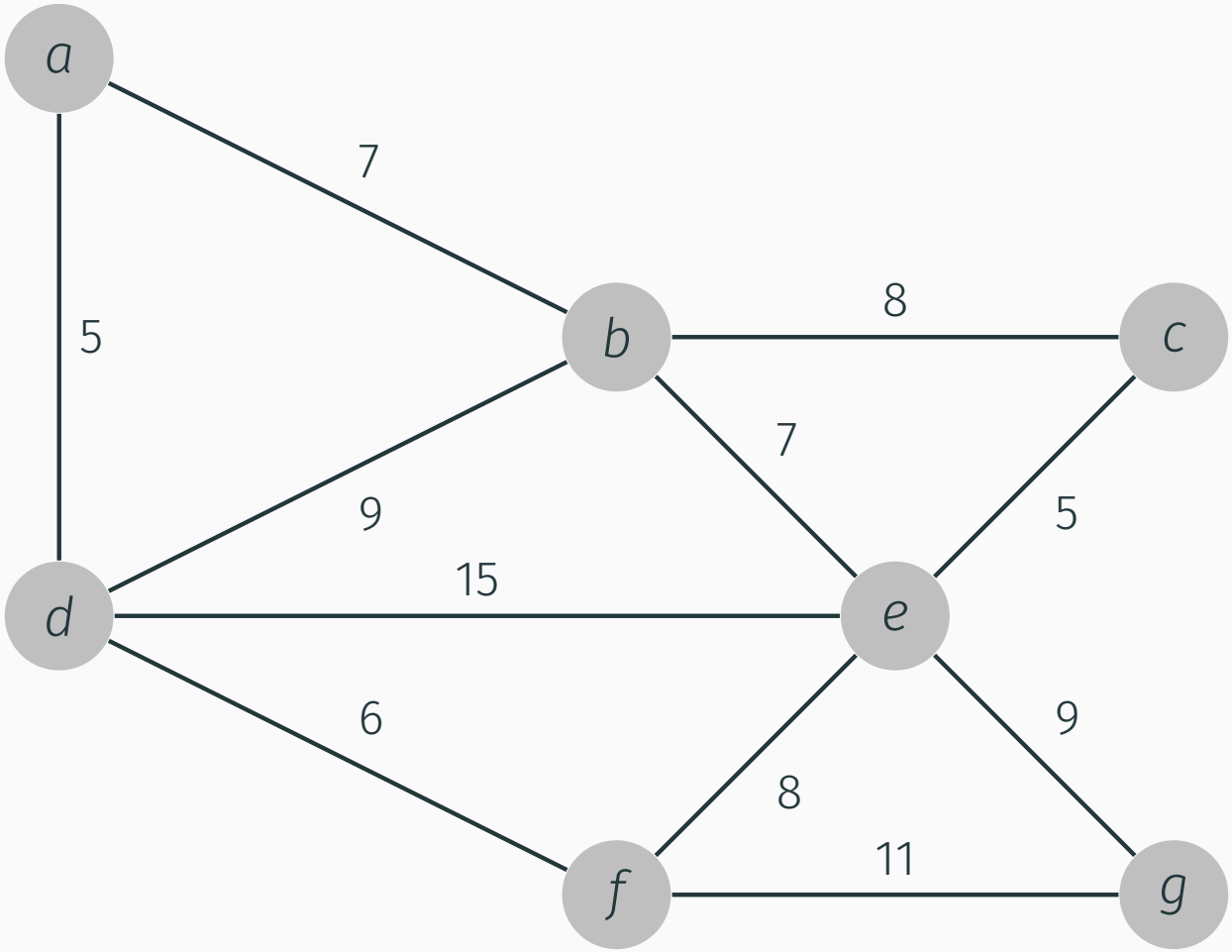
- **Shortest path tree rooted at** *a*: a-b, a-c
- **Minimum spanning tree**: a-b-c

Minimum Spanning Tree

Kruskal's algorithm

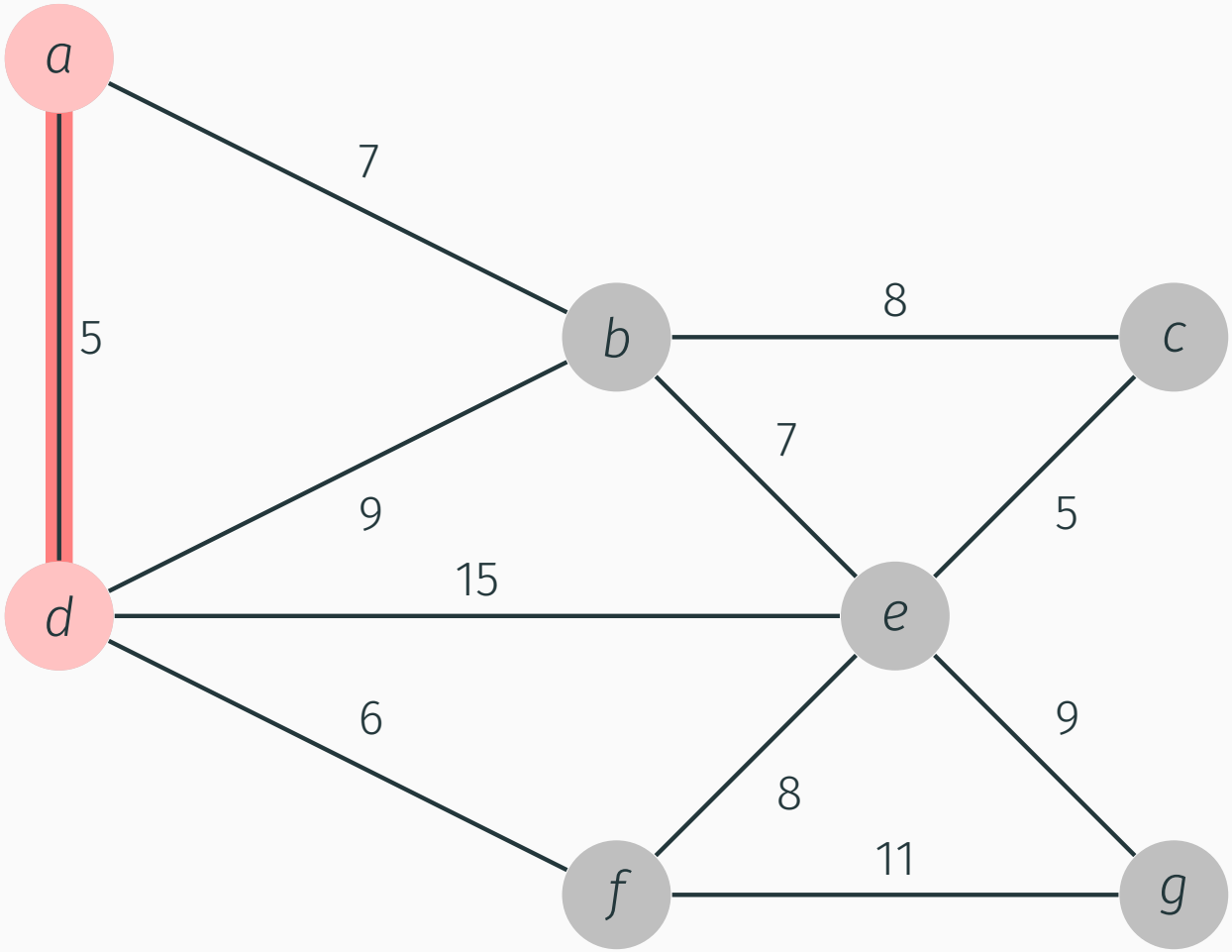
Kruskal's algorithm

- Select edges as light as possible
- If meeting an edge that may induce a cycle, ignore it
- Repeat the above steps



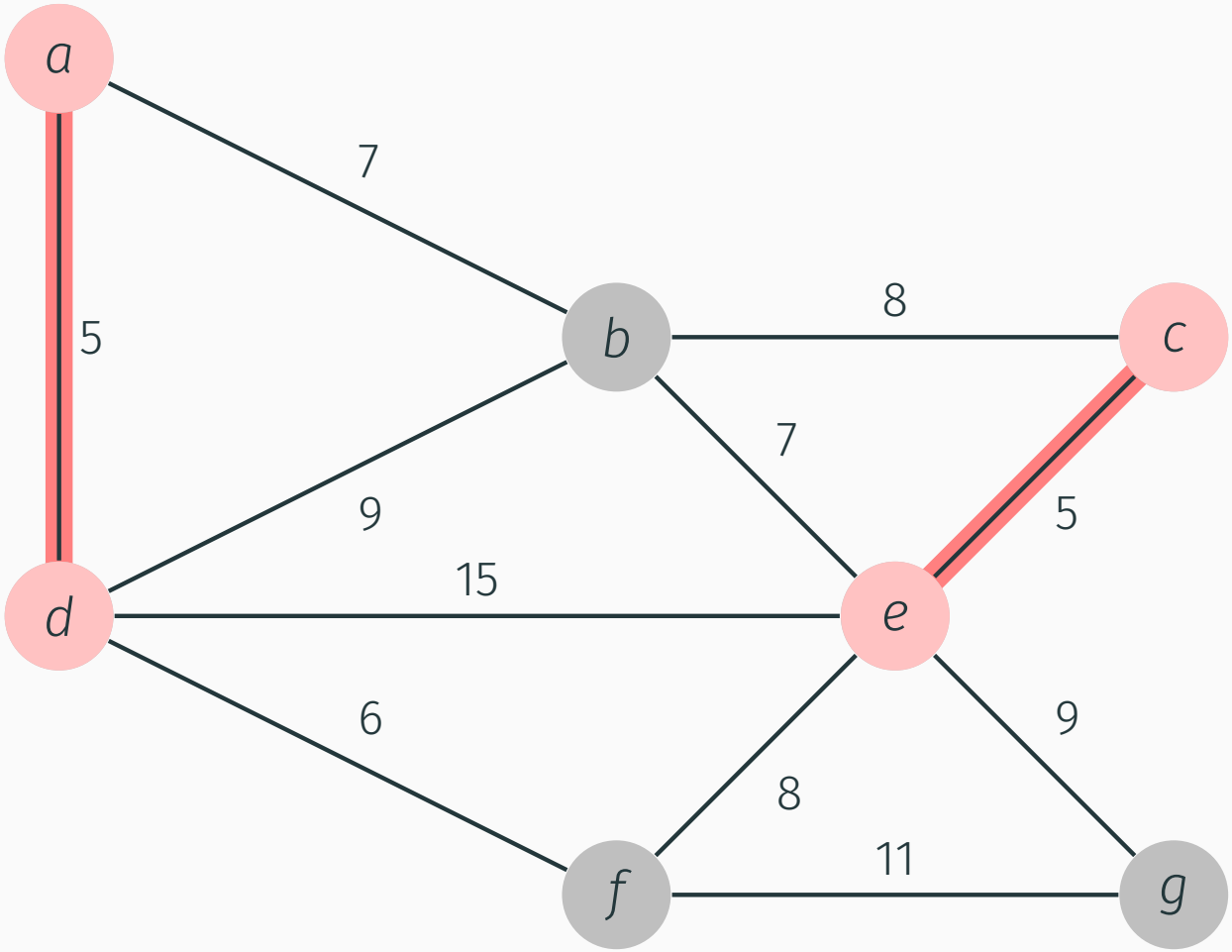
Kruskal's algorithm

- Select edges as light as possible
- If meeting an edge that may induce a cycle, ignore it
- Repeat the above steps



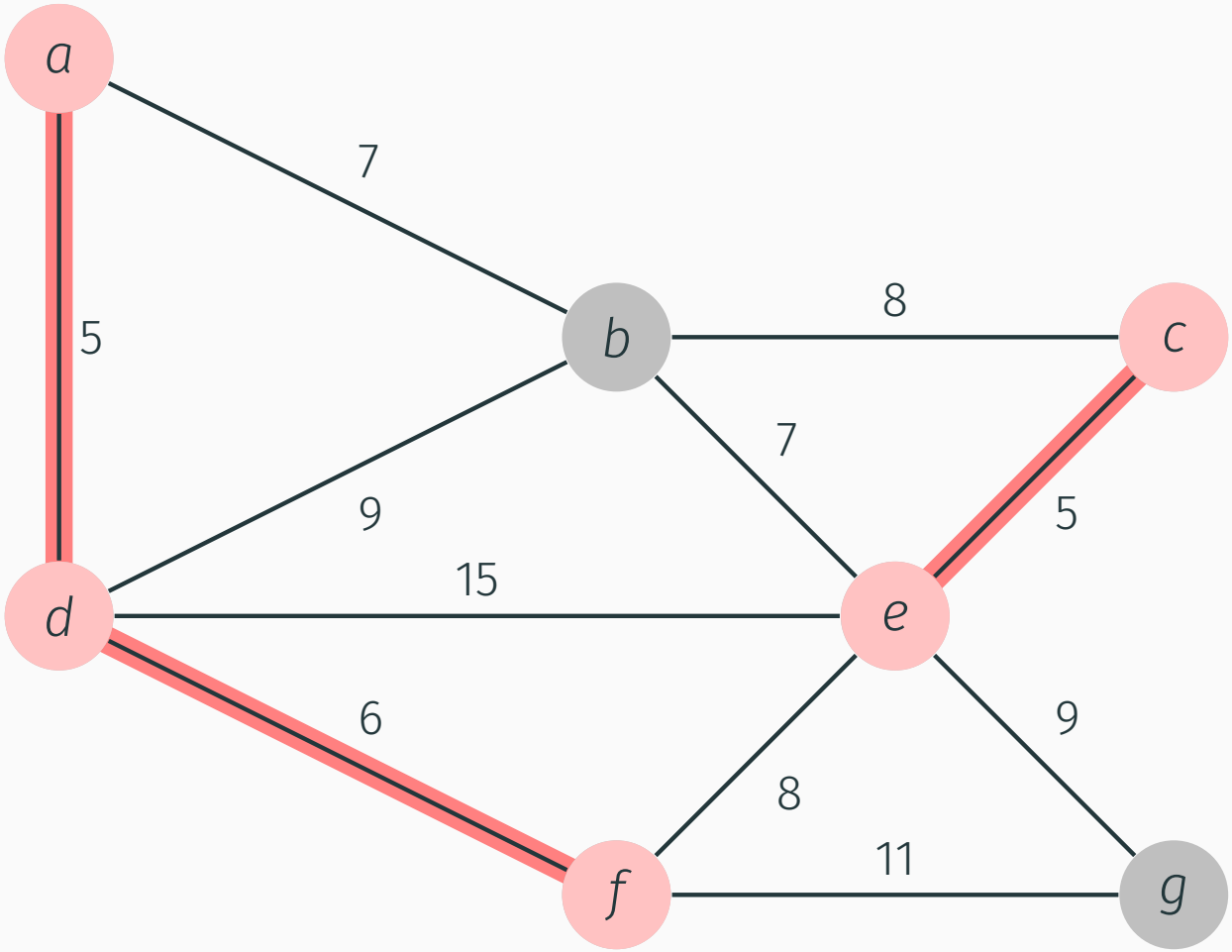
Kruskal's algorithm

- Select edges as light as possible
- If meeting an edge that may induce a cycle, ignore it
- Repeat the above steps



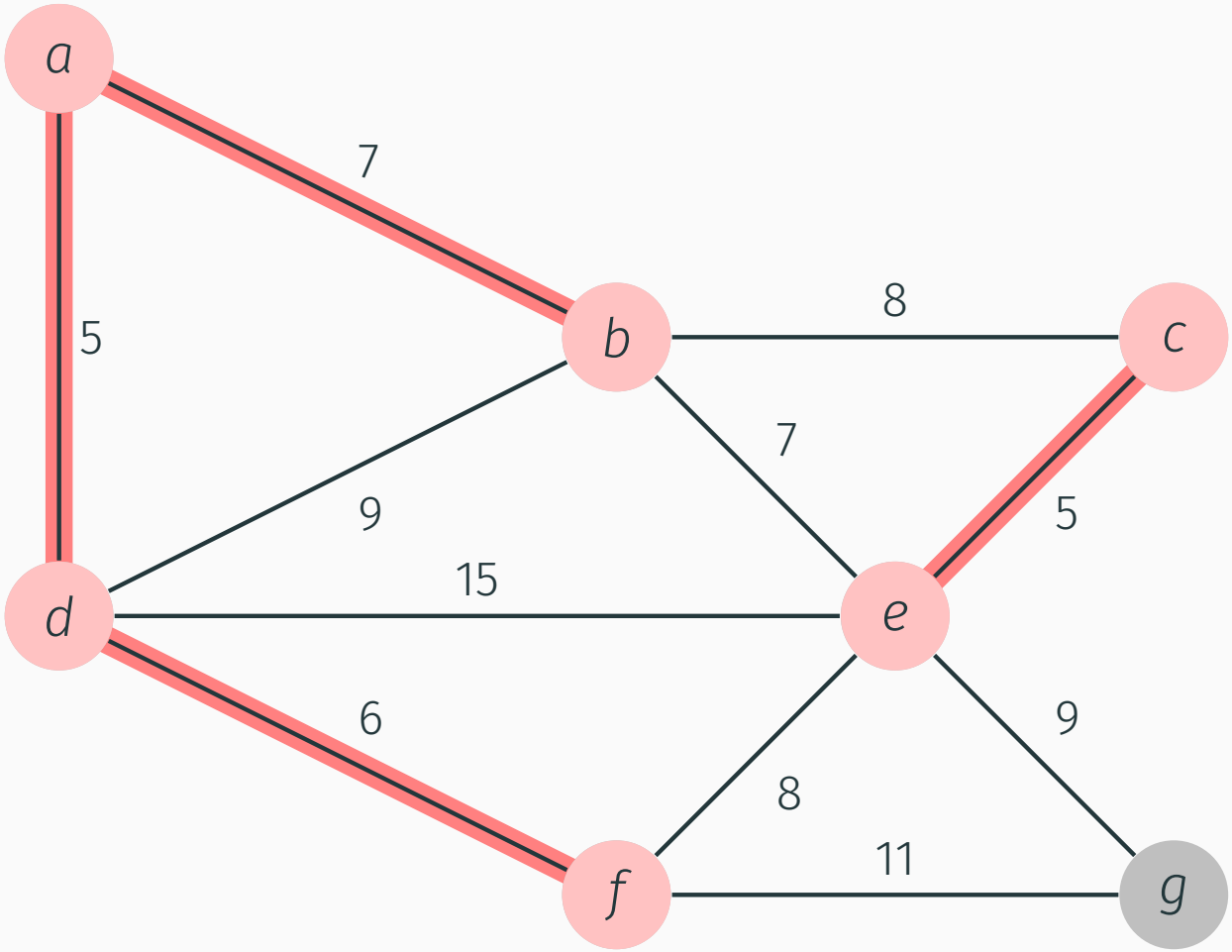
Kruskal's algorithm

- Select edges as light as possible
- If meeting an edge that may induce a cycle, ignore it
- Repeat the above steps



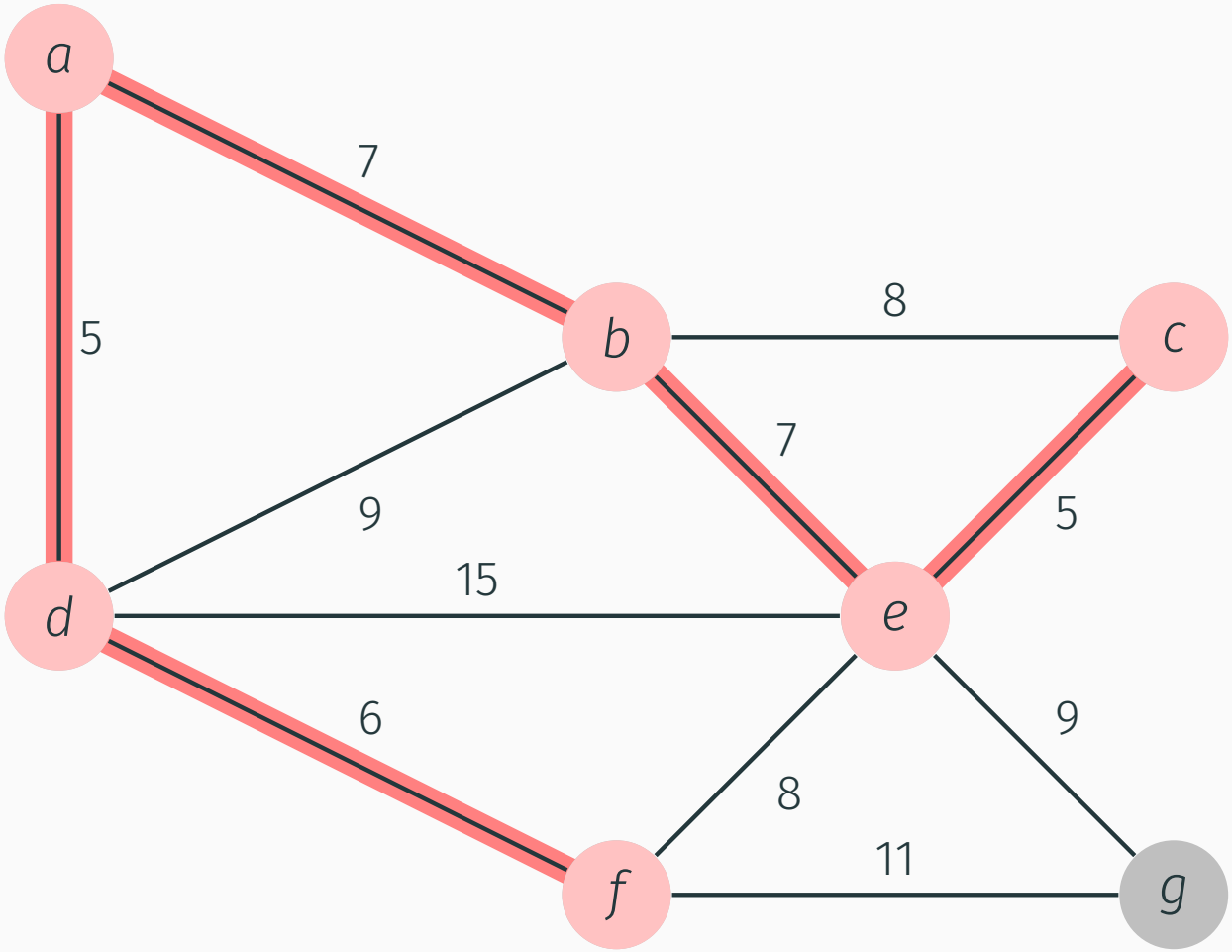
Kruskal's algorithm

- Select edges as light as possible
- If meeting an edge that may induce a cycle, ignore it
- Repeat the above steps



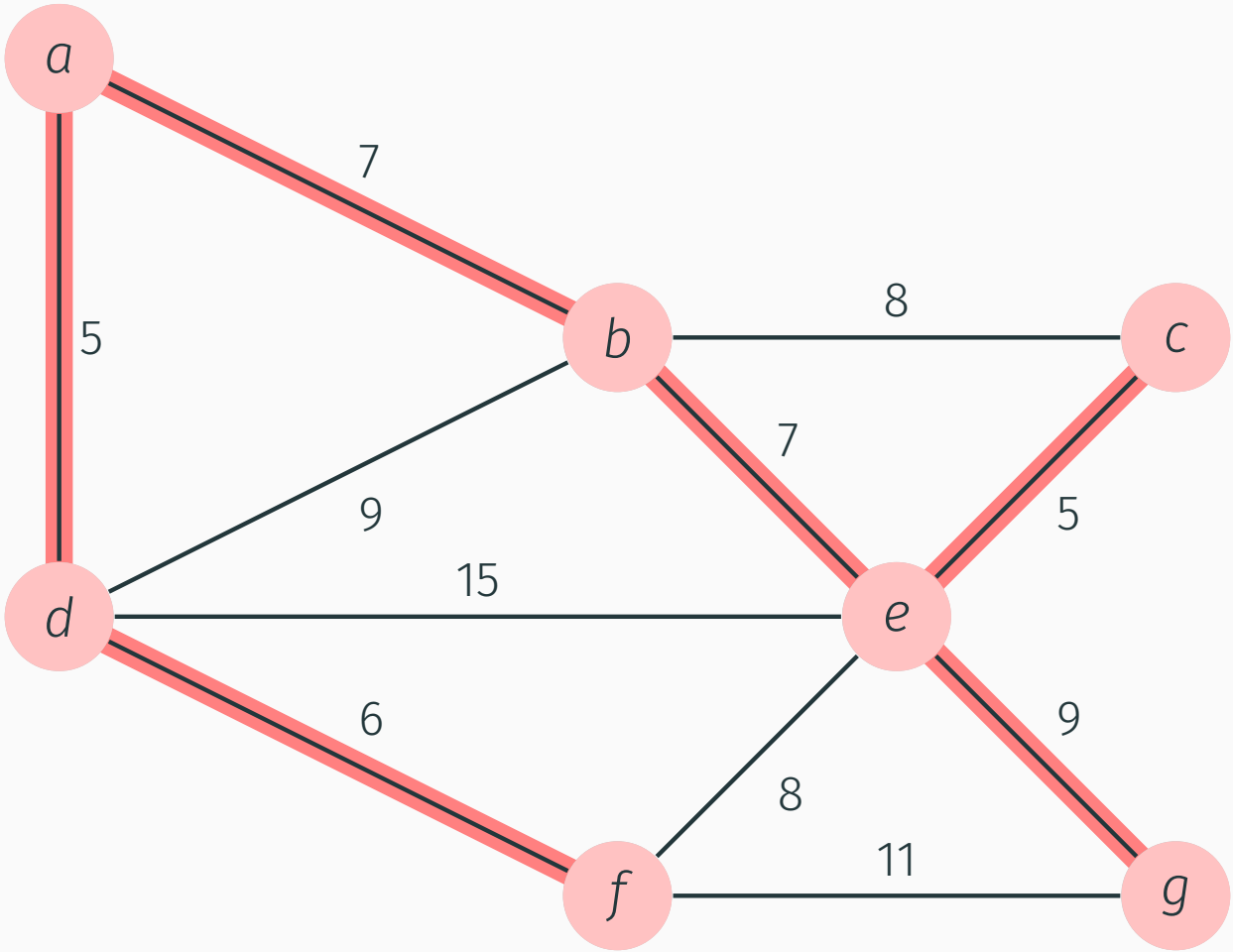
Kruskal's algorithm

- Select edges as light as possible
- If meeting an edge that may induce a cycle, ignore it
- Repeat the above steps



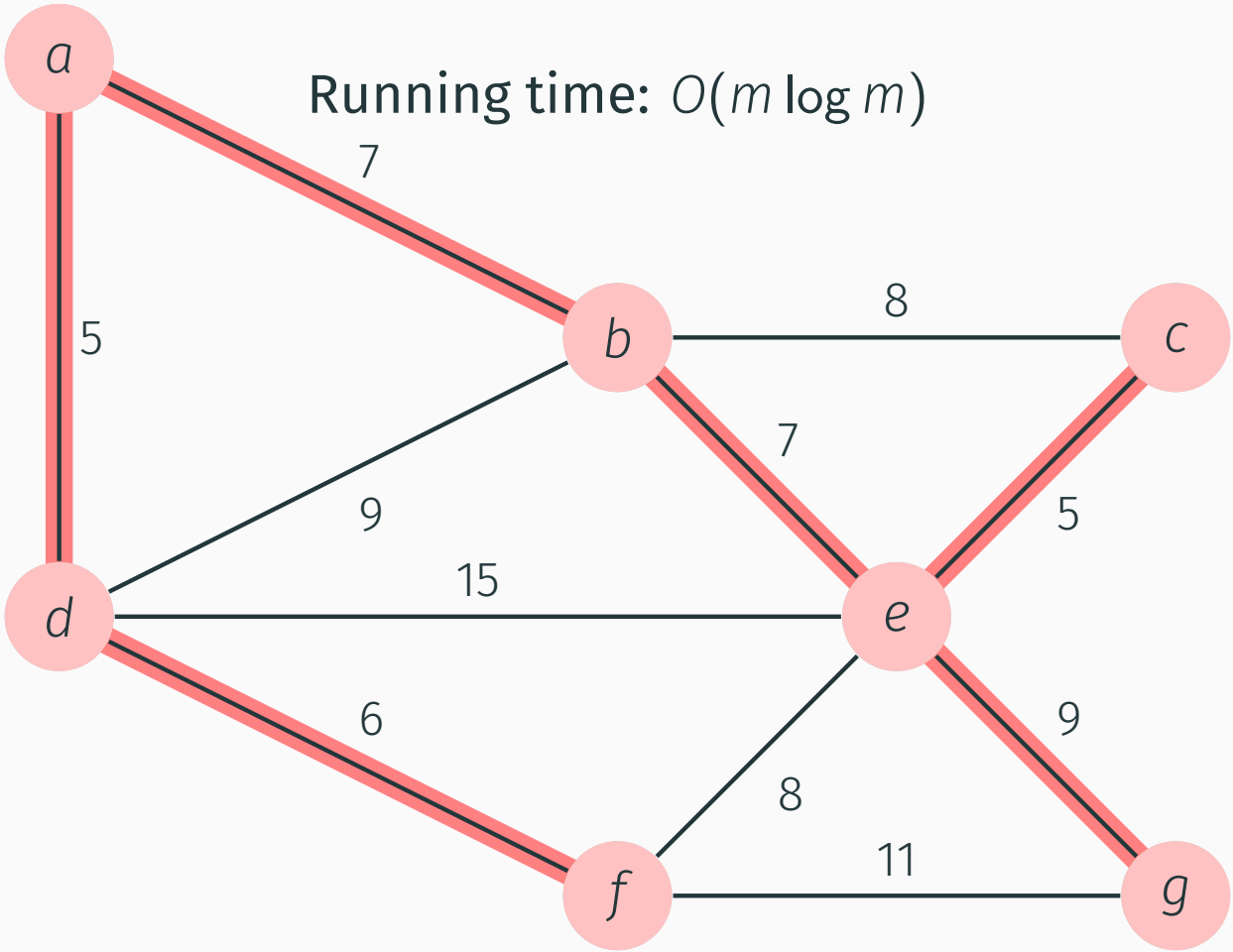
Kruskal's algorithm

- Select edges as light as possible
- If meeting an edge that may induce a cycle, ignore it
- Repeat the above steps



Kruskal's algorithm

- Select edges as light as possible
- If meeting an edge that may induce a cycle, ignore it
- Repeat the above steps



Running time: $O(m \log m)$

Thank you!

Questions?