6.1 & 6.2: Graph Searching

Frank Stajano

Thomas Sauerwald

Lent 2015


**Figure 22.1** Two representations of an undirected graph. (a) An undirected graph $G$ with 5 vertices and 7 edges. (b) An adjacency-list representation of $G$. (c) The adjacency-matrix representation of $G$. 

If $G$ is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$, since an edge of the form $(u; v)$ is represented by having $v$ appear in $\text{Adj}[u]$. If $G$ is
Figure 22.1 Two representations of an undirected graph. (a) An undirected graph $G$ with 5 vertices and 7 edges. (b) An adjacency-list representation of $G$. (c) The adjacency-matrix representation of $G$.

Figure 22.2 Two representations of a directed graph. (a) A directed graph $G$ with 6 vertices and 8 edges. (b) An adjacency-list representation of $G$. (c) The adjacency-matrix representation of $G$. 
Graph Searching

Overview

- **Graph searching** means traversing a graph via the edges in order to visit all vertices
- useful for identifying connected components, computing the diameter etc.
Graph searching means traversing a graph via the edges in order to visit all vertices. Useful for identifying connected components, computing the diameter etc. Two strategies: Breadth-First-Search and Depth-First-Search.
Graph Searching

Overview

- **Graph searching** means traversing a graph via the edges in order to visit all vertices.
- Useful for identifying connected components, computing the diameter etc.
- Two strategies: **Breadth-First-Search** and **Depth-First-Search**

Measure time complexity in terms of the size of $V$ and $E$ (often write just $V$ instead of $|V|$, and $E$ instead of $|E|$).
Outline

Breadth-First Search

Depth-First Search

Topological Sort

Minimum Spanning Tree Problem
Breadth-First Search: Basic Ideas

Given an undirected/directed graph $G = (V, E)$ and source vertex $s$, BFS sends out a wave from $s$ to compute distances/shortest paths.

Vertex Colours:
- **White** = Unvisited
- **Grey** = Visited, but not all neighbors (=adjacent vertices)
- **Black** = Visited and all neighbors

Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex $s$
Breadth-First Search: Basic Ideas

Basic Idea

- Given an undirected/directed graph \( G = (V, E) \) and source vertex \( s \)
- BFS sends out a wave from \( s \) \( \rightsquigarrow \) compute distances/shortest paths
Breadth-First Search: Basic Ideas

- **Given an undirected/directed graph** $G = (V, E)$ and source vertex $s$
- **BFS sends out a wave** from $s \rightarrow$ compute distances/shortest paths
- **Vertex Colours:**
  - **White** = Unvisited
  - **Grey** = Visited, but not all neighbors (=adjacent vertices)
  - **Black** = Visited and all neighbors
Breadth-First-Search: Pseudocode

```python
0: def bfs(G, s):
1:     # Run BFS on the given graph G
2:     # starting from source s
3:     assert(s in G.vertices())
4: # Initialize graph and queue
5:     for v in G.vertices():
6:         v.predecessor = None
7:         v.d = Infinity  # .d = distance from s
8:         v.colour = "white"
9:     Q = Queue()
10: # Visit source vertex
11:     s.d = 0
12:     s.colour = "grey"
13:     Q.insert(s)
14: # Visit the adjacents of each vertex in Q
15:     while not Q.isEmpty():
16:         u = Q.extract()
17:         assert (u.colour == "grey")
18:         for v in u.adjacent():
19:             if v.colour == "white"
20:                 v.colour = "grey"
21:                 v.d = u.d + 1
22:                 v.predecessor = u
23:                 Q.insert(v)
24:         u.colour = "black"
```

**From any vertex, visit all adjacent vertices before going any deeper**

**Vertex Colours:**
- **White** = Unvisited
- **Grey** = Visited, but not all neighbors
- **Black** = Visited and all neighbors

**Runtime**

Assuming that all executions of the FOR-loop for $u$ takes $O(|u$.adj$|)$ (adjacency list model),

$$\sum_{u \in V} |u$.adj$| = 2 |E|$$
Breadth-First-Search: Pseudocode

0: def bfs(G,s)
1:     Run BFS on the given graph G
2:     starting from source s
3:
4:     assert(s in G.vertices())
5:
6:     # Initialize graph and queue
7:     for v in G.vertices():
8:         v.predecessor = None
9:         v.d = Infinity # .d = distance from s
10:        v.colour = "white"
11:       Q = Queue()
12:
13:     # Visit source vertex
14:       s.d = 0
15:       s.colour = "grey"
16:       Q.insert(s)
17:
18:     # Visit the adjacents of each vertex in Q
19:     while not Q.isEmpty():
20:         u = Q.extract()
21:         assert (u.colour == "grey")
22:         for v in u.adjacent():
23:             if v.colour == "white"
24:                 v.colour = "grey"
25:                 v.d = u.d+1
26:                 v.predecessor = u
27:                 Q.insert(v)
28:         u.colour = "black"

- From any vertex, visit all adjacent vertices before going any deeper
Breadth-First-Search: Pseudocode

```python
0: def bfs(G,s)
1:     Run BFS on the given graph G
2:     starting from source s
3:     assert(s in G.vertices())
4: # Initialize graph and queue
5:     for v in G.vertices():
6:         v.predecessor = None
7:         v.d = Infinity # .d = distance from s
8:         v.colour = "white"
9:     Q = Queue()
10:    # Visit source vertex
11:    s.d = 0
12:    s.colour = "grey"
13:    Q.insert(s)
14:    # Visit the adjacents of each vertex in Q
15:    while not Q.isEmpty():
16:        u = Q.extract()
17:        assert (u.colour == "grey")
18:        for v in u.adjacent():
19:            if v.colour = "white"
20:                v.colour = "grey"
21:                v.d = u.d+1
22:                v.predecessor = u
23:                Q.insert(v)
24:        u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:
  - **White** = Unvisited
  - **Grey** = Visited, but not all neighbors
  - **Black** = Visited and all neighbors
**Breadth-First-Search: Pseudocode**

0: def bfs(G, s)
1:     Run BFS on the given graph G
2:     starting from source s
3:     assert(s in G.vertices())
4:     # Initialize graph and queue
5:     for v in G.vertices():
6:         v.predecessor = None
7:         v.d = Infinity # .d = distance from s
8:         v.colour = "white"
9:     Q = Queue()
10:     # Visit source vertex
11:     s.d = 0
12:     s.colour = "grey"
13:     Q.insert(s)
14:     # Visit the adjacent vertices of each vertex in Q
15:     while not Q.isEmpty():
16:         u = Q.extract()
17:         assert (u.colour == "grey")
18:         for v in u.adjacent():
19:             if v.colour = "white"
20:                 v.colour = "grey"
21:                 v.d = u.d+1
22:                 v.predecessor = u
23:                 Q.insert(v)
24:             else:
25:                 v.colour = "black"

- From any vertex, visit all adjacent vertices before going any deeper
- **Vertex Colours:**
  - **White** = Unvisited
  - **Grey** = Visited, but not all neighbors
  - **Black** = Visited and all neighbors
- **Runtime ???**
Breadth-First-Search: Pseudocode

```python
0: def bfs(G,s)
1:     Run BFS on the given graph G
2:     starting from source s
3:     assert(s in G.vertices())
4: # Initialize graph and queue
5: for v in G.vertices():
6:     v.predecessor = None
7:     v.d = Infinity # .d = distance from s
8:     v.colour = "white"
9: Q = Queue()
10: # Visit source vertex
11: s.d = 0
12: s.colour = "grey"
13: Q.insert(s)
14: # Visit the adjacent vertices of each vertex in Q
15: while not Q.isEmpty():
16:     u = Q.extract()
17:     assert (u.colour == "grey")
18:     for v in u.adjacent():
19:         if v.colour = "white"
20:             v.colour = "grey"
21:             v.d = u.d+1
22:             v.predecessor = u
23:             Q.insert(v)
24:     u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:
  - **White** = Unvisited
  - **Grey** = Visited, but not all neighbors
  - **Black** = Visited and all neighbors
- Runtime ???

6.1 & 6.2: Graph Searching
Breadth-First-Search: Pseudocode

0: def bfs(G, s)
1:   Run BFS on the given graph G
2:   starting from source s
3:  
4:   assert(s in G.vertices())
5:  
6:   # Initialize graph and queue
7:   for v in G.vertices():
8:     v.predecessor = None
9:     v.d = Infinity # .d = distance from s
10:    v.colour = "white"
11:  Q = Queue()
12: 
13:   # Visit source vertex
14:   s.d = 0
15:   s.colour = "grey"
16:  Q.insert(s)
17: 
18:   # Visit the adjacents of each vertex in Q
19:   while not Q.isEmpty():
20:     u = Q.extract()
21:     assert (u.colour == "grey")
22:     for v in u.adjacent():
23:       if v.colour == "white"
24:         v.colour = "grey"
25:         v.d = u.d + 1
26:         v.predecessor = u
27:       Q.insert(v)
28:     u.colour = "black"

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:
  - **White** = Unvisited
  - **Grey** = Visited, but not all neighbors
  - **Black** = Visited and all neighbors
- Runtime $O(V + E)$
Breadth-First-Search: Pseudocode

```python
def bfs(G, s):
    # Run BFS on the given graph G starting from source s
    assert(s in G.vertices())
    # Initialize graph and queue
    for v in G.vertices():
        v.predecessor = None
        v.d = Infinity  # .d = distance from s
        v.colour = "white"
    Q = Queue()
    # Visit source vertex
    s.d = 0
    s.colour = "grey"
    Q.insert(s)
    # Visit the adjacents of each vertex in Q
    while not Q.isEmpty():
        u = Q.extract()
        assert (u.colour == "grey")
        for v in u.adjacent():
            if v.colour == "white"
                v.colour = "grey"
                v.d = u.d+1
                v.predecessor = u
                Q.insert(v)
        u.colour = "black"
```

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:
  - **White** = Unvisited
  - **Grey** = Visited, but not all neighbors
  - **Black** = Visited and all neighbors
- Runtime $O(V + E)$

Assuming that all executions of the FOR-loop for $u$ takes $O(|u.adj|)$ (adjacency list model!!)
Breadth-First-Search: Pseudocode

0: `def bfs(G, s)`
1: Run BFS on the given graph `G` starting from source `s`
3: 
4: `assert(s in G.vertices())`
5: 
6: Initialize graph and queue
7: `for v in G.vertices():`
8: `v.predecessor = None`
9: `v.d = Infinity` # `.d = distance from s`
10: `v.colour = "white"`
11: `Q = Queue()`
12: 
13: Visit source vertex
14: `s.d = 0`
15: `s.colour = "grey"`
16: `Q.insert(s)`
17: 
18: Visit the adjacent vertices of each vertex in Q
19: `while not Q.isEmpty():`
20: `u = Q.extract()`
21: `assert (u.colour == "grey")`
22: `for v in u.adjacent():`
23: `if v.colour == "white"`
24: `v.colour = "grey"`
25: `v.d = u.d + 1`
26: `v.predecessor = u`
27: `Q.insert(v)`
28: `u.colour = "black"`

- From any vertex, visit all adjacent vertices before going any deeper
- Vertex Colours:
  - **White** = Unvisited
  - **Grey** = Visited, but not all neighbors
  - **Black** = Visited and all neighbors
- Runtime $O(V + E)$

Assuming that all executions of the FOR-loop for `u` takes $O(|u\text{.adj}|)$ (adjacency list model!)

$$\sum_{u \in V} |u\text{.adj}| = 2|E|$$
Complete Execution of BFS (Figure 22.3)

Queue:

```
<table>
<thead>
<tr>
<th>r</th>
<th>∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>∞</td>
</tr>
<tr>
<td>s</td>
<td>0</td>
</tr>
<tr>
<td>w</td>
<td>∞</td>
</tr>
<tr>
<td>t</td>
<td>∞</td>
</tr>
<tr>
<td>x</td>
<td>∞</td>
</tr>
<tr>
<td>u</td>
<td>∞</td>
</tr>
<tr>
<td>v</td>
<td>∞</td>
</tr>
<tr>
<td>w</td>
<td>∞</td>
</tr>
<tr>
<td>x</td>
<td>∞</td>
</tr>
<tr>
<td>y</td>
<td>∞</td>
</tr>
</tbody>
</table>
```

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: \( s \)

![Graph Diagram]
Complete Execution of BFS (Figure 22.3)

Queue:

$r \rightarrow \infty$
$s \rightarrow 0$
$t \rightarrow \infty$
$u \rightarrow \infty$
$v \rightarrow \infty$
$w \rightarrow \infty$
$x \rightarrow \infty$
$y \rightarrow \infty$
$\rightarrow \infty$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue:

\[ \infty \]

\[ r \]

\[ s \]

\[ t \]

\[ u \]

\[ v \]

\[ w \]

\[ x \]

\[ y \]

$\$
Complete Execution of BFS (Figure 22.3)

Queue: $\not\in \overset{r}{s} r$

Diagram: 

- Vertices: $v, w, x, y$
- Edges: $r \rightarrow s, t \rightarrow u$
- Queue: $r$
Complete Execution of BFS (Figure 22.3)

Queue: $r$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $s \ r \ w$

![Graph Diagram]

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $s \rightarrow r \rightarrow w$

![Graph diagram showing BFS execution]
Complete Execution of BFS (Figure 22.3)

Queues:

\[ s \quad \bar{x} \quad w \]
Complete Execution of BFS (Figure 22.3)

Queue: $s \times w$

```
1 r
∞ v

1 s
∞ w

∞ t
∞ x
∞ y

∞ u

1 t
∞ u
```
Complete Execution of BFS (Figure 22.3)

Queue: $s \ s \ w$

[Diagram showing a graph with nodes $r$, $s$, $t$, $u$, $v$, $w$, $x$, $y$ and edges connecting them.]
Complete Execution of BFS (Figure 22.3)

Queue: $s$ $x$ $w$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $\$ \ x \ w \ v$

![Graph](image)
Queue: $s \quad x \quad w \quad v$

Complete Execution of BFS (Figure 22.3)
Complete Execution of BFS (Figure 22.3)

Queue: $s, x, w, v$

Graph: Nodes labeled with distances, edges connecting nodes.
Complete Execution of BFS (Figure 22.3)

Queue: $s \times w \times v$

Diagram of a graph with nodes $r$, $s$, $t$, $u$, $v$, $w$, $x$, $y$, and edges connecting them.
Complete Execution of BFS (Figure 22.3)

Queue:

$$\begin{align*}
\text{Queue:} & \quad \{s, x, w, v\} \\
& \quad r \quad s \quad t \quad u
\end{align*}$$

```
1
2
v
```
```
0
1
w
```
```
\infty
\infty
```
```
\infty
x
y
\infty
```
Complete Execution of BFS (Figure 22.3)

Queue: $s$, $x$, $w$, $v$
Complete Execution of BFS (Figure 22.3)

Queue: $s \, \times \, \times \, v \, \, t$

Diagram: A graph with nodes labeled $r, s, t, u, v, w, x, y$ and edges connecting them in a specific pattern.
Complete Execution of BFS (Figure 22.3)

Queue:

$\infty \ r \ s \ \infty \ t \ \infty \ u \ \infty \ v \ \infty \ x \ \infty \ y$

Graph:

- $r$ to $1$
- $s$ to $0$
- $t$ to $2$
- $u$ to $\infty$
- $v$ to $2$
- $w$ to $1$
- $x$ to $\infty$
- $y$ to $\infty$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue:

\[ s \quad x \quad w \quad v \quad t \quad x \]
Complete Execution of BFS (Figure 22.3)

Queue: \( s \ x \ w \ v \ t \ x \)

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue:

$\text{Queue: } s \ x \ \cancel{w} \ \cancel{v} \ t \ x$

![Graph Diagram]

6.1 & 6.2: Graph Searching

T.S.
Complete Execution of BFS (Figure 22.3)

Queue:

$\$ s \ x \ w \ x \ t \ x$

---

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $r \ s \ w \ t \ x$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $\infty r s t u \infty v \infty x$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue:

\[ \infty r s \infty t \infty u \infty v \infty w \infty x \]

\[ \infty r 1 s 0 t 2 u \infty \]

\[ \infty v 2 w 1 x 3 u 3 y 3 \]

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $\infty r s t u \infty v \infty w \infty x y$
Complete Execution of BFS (Figure 22.3)

Queue: $s \ x \ \_\_\_ \ x \ x \ x \ x \ x \ x \ u$

![Graph Diagram]
Complete Execution of BFS (Figure 22.3)

Queue: $s \ x \ w \ x \ x \ x \ x \ u$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $s \ x \ w \ x \ x \ x \ x \ u$

![Graph Diagram](attachment:image.png)
Complete Execution of BFS (Figure 22.3)

Queue:  $s \ x \ w \ x \ x \ x \ u$

Graph:

- $r$ to $s$
- $s$ to $t$
- $t$ to $u$
- $s$ to $v$
- $s$ to $w$
- $v$ to $x$
- $w$ to $x$

Nodes and edges as shown in the figure.
Complete Execution of BFS (Figure 22.3)

Queue: $\infty s \infty x \infty w \infty x \infty x \infty u$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $s \ x \ w \ x \ x \ x \ u$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: \( \infty r \infty s \infty t \infty u \infty v \infty w \infty x \infty y \infty z \infty w \infty x \infty v \infty s \infty r \infty w \infty x \infty x \infty x \infty u \infty \)

Diagram with nodes and edges.
Complete Execution of BFS (Figure 22.3)

Queue: $s \times \times w \times \times x \times u$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: \( r \ s \ t \ u \)

![Graph diagram](image)
Complete Execution of BFS (Figure 22.3)

Queue: $\emptyset \times \times \times \times \times u$

![Graph Diagram]
Complete Execution of BFS (Figure 22.3)

Queue:

\[ s \quad x \quad \cancel{w} \quad \cancel{x} \quad \cancel{x} \quad \cancel{x} \quad u \]

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $\infty r s \infty t \infty u \infty v \infty x \infty y \infty w \infty v \infty u$

![Graph Diagram]
Complete Execution of BFS (Figure 22.3)

Queue:

\[ r \quad s \quad x \quad w \quad x \quad x \quad x \quad u \]

![Graph diagram with nodes and edges showing BFS traversal]

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $s \ x \ \not w \ \not x \ \not x \ \not x \ u \ y$

Graph:

- Vertices: $r, s, t, u, v, w, x, y$
- Edges: $r \rightarrow s, s \rightarrow t, t \rightarrow u, u \rightarrow x, x \rightarrow y, v \rightarrow w$

Steps:
1. $r$ enters the queue
2. $s$ enters the queue
3. $t$ enters the queue
4. $u$ enters the queue
5. $x$ enters the queue
6. $y$ enters the queue

Notes:
- $w$ is not visited.
- $x$ is visited twice.
- $y$ is the last vertex to be visited.
Complete Execution of BFS (Figure 22.3)

Queue: $s \ x \ w \ y \ u \ y$
Complete Execution of BFS (Figure 22.3)

Queue: $s \times w \times u \times y$

Diagram showing nodes and edges with labels $r, s, t, u, v, w, x, y$.
Complete Execution of BFS (Figure 22.3)

Queue: $s \ x \ w \ x \ x \ x \ u \ y$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $s \ x \ w \ x \ x \ x \ u \ y$

Graph:

- Vertices: $r, s, t, u, v, w, x, y$
- Edges: connections between vertices

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue:

\[ \text{Queue: } s \quad x \quad w \quad x \quad x \quad x \quad u \quad y \]

\[ r \quad 1 \quad s \quad 0 \quad t \quad 2 \quad u \quad 3 \]

\[ v \quad w \quad x \quad y \]

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $s \ x \ w \ x \ x \ x \ u \ y$

The image shows a directed graph with nodes labeled $r, s, t, u, v, w, x, y$ and edges illustrating the breadth-first search (BFS) traversal. The queue $s, x, w, x, x, u, y$ indicates the order in which vertices are explored.
Complete Execution of BFS (Figure 22.3)
Complete Execution of BFS (Figure 22.3)

Queue: $s \ x \ w \ x \ x \ x \ x \ u \ y$

Graph representation: 

Vertices: $r, s, t, u, v, w, x, y$

Edges: 
- $r \rightarrow 1$
- $1 \rightarrow 2$
- $s \rightarrow 0$
- $0 \rightarrow 1$
- $2 \rightarrow 1$
- $t \rightarrow 2$
- $2 \rightarrow x$
- $u \rightarrow 3$
- $3 \rightarrow y$
- $x \rightarrow y$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue: $s \ x \ w \ x \ x \ x \ u \ y$

Diagram of BFS execution.
Complete Execution of BFS (Figure 22.3)

Queue: $s$, $x$, $w$, $y$

Graph:

- Vertices: $r$, $s$, $t$, $u$, $v$, $w$, $x$, $y$
- Edges: $r ightarrow s$, $s ightarrow t$, $t ightarrow u$, $u ightarrow v$, $v ightarrow w$, $w ightarrow x$, $x ightarrow y$

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue:

 Queue: [Nodes]
Complete Execution of BFS (Figure 22.3)

Queue:

```plaintext
<table>
<thead>
<tr>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
</tr>
</tbody>
</table>
```

---

6.1 & 6.2: Graph Searching
Complete Execution of BFS (Figure 22.3)

Queue:

![Graph Diagram]

Nodes: s, t, w, x, u, y

Edges:
- s → r
- r → 1
- 1 → 2
- 2 → 1
- 1 → 0
- 0 → s
- s → w
- w → t
- t → 2
- 2 → x
- x → 3
- 3 → y

Steps:
1. s is added to the queue.
2. s is removed from the queue.
3. s is marked as visited.
4. r is added to the queue.
5. r is removed from the queue.
6. r is marked as visited.
7. s is added to the queue.
8. s is removed from the queue.
9. s is marked as visited.
10. t is added to the queue.
11. t is removed from the queue.
12. t is marked as visited.
13. s is added to the queue.
14. s is removed from the queue.
15. s is marked as visited.
16. w is added to the queue.
17. w is removed from the queue.
18. w is marked as visited.
19. t is added to the queue.
20. t is removed from the queue.
21. t is marked as visited.
22. w is added to the queue.
23. w is removed from the queue.
24. w is marked as visited.
25. t is added to the queue.
26. t is removed from the queue.
27. t is marked as visited.
28. s is added to the queue.
29. s is removed from the queue.
30. s is marked as visited.
31. t is added to the queue.
32. t is removed from the queue.
33. t is marked as visited.
34. w is added to the queue.
35. w is removed from the queue.
36. w is marked as visited.
37. t is added to the queue.
38. t is removed from the queue.
39. t is marked as visited.
40. w is added to the queue.
41. w is removed from the queue.
42. w is marked as visited.
43. t is added to the queue.
44. t is removed from the queue.
45. t is marked as visited.
46. w is added to the queue.
47. w is removed from the queue.
48. w is marked as visited.
49. t is added to the queue.
50. t is removed from the queue.
51. t is marked as visited.
52. w is added to the queue.
53. w is removed from the queue.
54. w is marked as visited.
55. t is added to the queue.
56. t is removed from the queue.
57. t is marked as visited.
58. w is added to the queue.
59. w is removed from the queue.
60. w is marked as visited.
61. t is added to the queue.
62. t is removed from the queue.
63. t is marked as visited.
64. w is added to the queue.
65. w is removed from the queue.
66. w is marked as visited.
67. t is added to the queue.
68. t is removed from the queue.
69. t is marked as visited.
70. w is added to the queue.
71. w is removed from the queue.
72. w is marked as visited.
73. t is added to the queue.
74. t is removed from the queue.
75. t is marked as visited.
76. w is added to the queue.
77. w is removed from the queue.
78. w is marked as visited.
79. t is added to the queue.
80. t is removed from the queue.
81. t is marked as visited.
82. w is added to the queue.
83. w is removed from the queue.
84. w is marked as visited.
85. t is added to the queue.
86. t is removed from the queue.
87. t is marked as visited.
88. w is added to the queue.
89. w is removed from the queue.
90. w is marked as visited.
91. t is added to the queue.
92. t is removed from the queue.
93. t is marked as visited.
94. w is added to the queue.
95. w is removed from the queue.
96. w is marked as visited.
97. t is added to the queue.
98. t is removed from the queue.
99. t is marked as visited.
100. w is added to the queue.
101. w is removed from the queue.
102. w is marked as visited.
103. t is added to the queue.
104. t is removed from the queue.
105. t is marked as visited.
106. w is added to the queue.
107. w is removed from the queue.
108. w is marked as visited.
109. t is added to the queue.
110. t is removed from the queue.
111. t is marked as visited.
112. w is added to the queue.
113. w is removed from the queue.
114. w is marked as visited.
115. t is added to the queue.
116. t is removed from the queue.
117. t is marked as visited.
118. w is added to the queue.
119. w is removed from the queue.
120. w is marked as visited.
121. t is added to the queue.
122. t is removed from the queue.
123. t is marked as visited.
124. w is added to the queue.
125. w is removed from the queue.
126. w is marked as visited.
127. t is added to the queue.
128. t is removed from the queue.
129. t is marked as visited.
130. w is added to the queue.
131. w is removed from the queue.
132. w is marked as visited.
133. t is added to the queue.
134. t is removed from the queue.
135. t is marked as visited.
Complete Execution of BFS (Figure 22.3)

Queue:

\[
\begin{align*}
\text{Queue: } & \quad r \quad s \quad w \quad x \quad y \quad u \quad x \\
\end{align*}
\]
Outline

Breadth-First Search

Depth-First Search

Topological Sort

Minimum Spanning Tree Problem
Depth-First Search: Basic Ideas

- Given an undirected/directed graph $G = (V, E)$ and source vertex $s$
Depth-First Search: Basic Ideas

Given an undirected/directed graph $G = (V, E)$ and source vertex $s$

As soon as we discover a vertex, explore from it

Basic Idea

- Given an undirected/directed graph $G = (V, E)$ and source vertex $s$
- As soon as we discover a vertex, explore from it

Solving Mazes
Depth-First Search: Basic Ideas

Basic Idea
- Given an undirected/directed graph \( G = (V, E) \) and source vertex \( s \)
- As soon as we discover a vertex, explore from it
- Two time stamps for every vertex: Discovery Time, Finishing Time

Solving Mazes
Depth-First-Search: Pseudocode

0: def dfs(G,s):
  1: Run DFS on the given graph G
  2: starting from the given source s
  3:
  4: assert(s in G.vertices())
  5:
  6: # Initialize graph
  7: for v in G.vertices():
  8:   v.predecessor = None
  9:   v.colour = "white"
10: dfsRecurse(G,s)

0: def dfsRecurse(G,s):
  1: s.colour = "grey"
  2: s.d = time() # .d = discovery time
  3: for v in s.adjacent():
  4:   if v.colour = "white"
  5:     v.predecessor = s
  6:     dfsRecurse(G,v)
  7: s.colour = "black"
  8: s.f = time() # .f = finish time
Depth-First-Search: Pseudocode

**Depth-First-Search**

- We always go deeper before visiting other neighbors

---

```python
0: def dfs(G, s):
1:   Run DFS on the given graph G
2:   starting from the given source s
3:
4:   assert(s in G.vertices())
5:
6:   # Initialize graph
7:   for v in G.vertices():
8:     v.predecessor = None
9:     v.colour = "white"
10:   dfsRecurse(G, s)

0: def dfsRecurse(G, s):
1:   s.colour = "grey"
2:   s.d = time() # .d = discovery time
3:   for v in s.adjacent():
4:     if v.colour = "white"
5:       v.predecessor = s
6:       dfsRecurse(G, v)
7:   s.colour = "black"
8:   s.f = time() # .f = finish time
```

---

6.1 & 6.2: Graph Searching
Depth-First-Search: Pseudocode

```python
0: def dfs(G, s):
    1: Run DFS on the given graph G
    2: starting from the given source s
    3:
    4: assert(s in G.vertices())
    5:
    6: # Initialize graph
    7: for v in G.vertices():
    8:     v.predecessor = None
    9:     v.colour = "white"
10: dfsRecurse(G, s)
```

- We always go deeper before visiting other neighbors
- **Discovery and Finish times**, \( .d \) and \( .f \)

```python
0: def dfsRecurse(G, s):
    1: s.colour = "grey"
    2: s.d = time() # .d = discovery time
    3: for v in s.adjacent():
    4:     if v.colour = "white"
7:         v.predecessor = s
6:         dfsRecurse(G, v)
7: s.colour = "black"
8: s.f = time() # .f = finish time
```
Depth-First-Search: Pseudocode

0: def dfs(G,s):
1:    Run DFS on the given graph G
2:    starting from the given source s
3:
4:    assert(s in G.vertices())
5:
6:    # Initialize graph
7:    for v in G.vertices():
8:        v.predecessor = None
9:        v.colour = "white"
10:   dfsRecurse(G,s)

0: def dfsRecurse(G,s):
1:    s.colour = "grey"
2:    s.d = time() # .d = discovery time
3:    for v in s.adjacent():
4:        if v.colour = "white"
5:           v.predecessor = s
6:           dfsRecurse(G,v)
7:    s.colour = "black"
8:    s.f = time() # .f = finish time

- We always go deeper before visiting other neighbors
- Discovery and Finish times, \( d \) and \( f \)
- Vertex Colours:
  - **White** = Unvisited
  - **Grey** = Visited, but not all neighbors
  - **Black** = Visited and all neighbors

6.1 & 6.2: Graph Searching
Depth-First-Search: Pseudocode

```python
0: def dfs(G,s):
1:     Run DFS on the given graph G
2:     starting from the given source s
3:
4:     assert(s in G.vertices())
5:
6:     # Initialize graph
7:     for v in G.vertices():
8:         v.predecessor = None
9:         v.colour = "white"
10:    dfsRecurse(G,s)
```

```python
0: def dfsRecurse(G,s):
1:     s.colour = "grey"
2:     s.d = time() # .d = discovery time
3:     for v in s.adjacent():
4:         if v.colour = "white"
5:             v.predecessor = s
6:             dfsRecurse(G,v)
7:     s.colour = "black"
8:     s.f = time() # .f = finish time
```

- We always go deeper before visiting other neighbors
- **Discovery** and **Finish times**, .\(d\) and .\(f\)
- **Vertex Colours:**
  - **White** = Unvisited
  - **Grey** = Visited, but not all neighbors
  - **Black** = Visited and all neighbors

6.1 & 6.2: Graph Searching
Depth-First-Search: Pseudocode

0: def dfs(G,s):
1: Run DFS on the given graph G
2: starting from the given source s
3: assert(s in G.vertices())
4: # Initialize graph
5: for v in G.vertices():
6: v.predecessor = None
7: v.colour = "white"
8: dfsRecurse(G,s)

0: def dfsRecurse(G,s):
1: s.colour = "grey"
2: s.d = time() # .d = discovery time
3: for v in s.adjacent():
4: if v.colour = "white"
5: v.predecessor = s
6: dfsRecurse(G,v)
7: s.colour = "black"
8: s.f = time() # .f = finish time

- We always go deeper before visiting other neighbors
- Discovery and Finish times, .d and .f
- Vertex Colours:
  - **White** = Unvisited
  - **Grey** = Visited, but not all neighbors
  - **Black** = Visited and all neighbors
- Runtime \( O(V + E) \)
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS
Complete Execution of DFS

1. Start with vertex S.
2. Visit vertex V.
3. Visit vertex Y.
5. Visit vertex 3/.
6. Visit vertex 2/.
7. Visit vertex 1/.
8. Visit vertex x.
10. Visit vertex z.
11. Visit vertex w.
12. Visit vertex s.
14. Visit vertex w.
15. Visit vertex z.
16. Visit vertex r.

6.1 & 6.2: Graph Searching
Complete Execution of DFS
Complete Execution of DFS
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching

T.S. 11
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS
Complete Execution of DFS
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS
Complete Execution of DFS
Complete Execution of DFS
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Complete Execution of DFS

6.1 & 6.2: Graph Searching
Paranthesis Theorem (Theorem 22.7)

\[
\begin{align*}
4/5 \times 6/9 \\
7/8 \times 6/9 \\
3/10 \times 14/15 \\
2/11 \times 13/16
\end{align*}
\]

6.1 & 6.2: Graph Searching
Outline

Breadth-First Search

Depth-First Search

Topological Sort

Minimum Spanning Tree Problem
Topological Sort

Given: a directed acyclic graph (DAG)
Goal: Output a linear ordering of all vertices

Problem 6.1 & 6.2: Graph Searching T.S. 14
Topological Sort

Problem

- **Given**: a directed acyclic graph (DAG)
- **Goal**: Output a linear ordering of all vertices
Topological Sort

Given: a directed acyclic graph (DAG)
Goal: Output a linear ordering of all vertices
Topological Sort

Given: a directed acyclic graph (DAG)
Goal: Output a linear ordering of all vertices
Topological Sort

Problem

- **Given**: a directed acyclic graph (DAG)
- **Goal**: Output a linear ordering of all vertices

Diagram:

- Undershorts → Pants → Belt → Shirt → Tie → Jacket
- Socks → Shoes → Watch
Perform DFS's so that all vertices are visited
Output vertices in decreasing order of their finishing time
Solving Topological Sort

Knuth’s Algorithm (1968)
- Perform DFS’s so that all vertices are visited
- Output vertices in decreasing order of their finishing time

Runtime $O(V + E)$
Solving Topological Sort

Knuth’s Algorithm (1968)
- Perform DFS’s so that all vertices are visited
- Output vertices in decreasing order of their finishing time

Runtime $O(V + E)$

Don’t need to sort the vertices – use DFS directly!
Execution of Knuth’s Algorithm

6.1 & 6.2: Graph Searching
Execution of Knuth’s Algorithm

6.1 & 6.2: Graph Searching
 Execution of Knuth’s Algorithm

\[
\begin{align*}
S & \rightarrow 1/12 \\
& \downarrow \\
4/5 & \rightarrow X
\end{align*}
\]

\[
\begin{align*}
V & \rightarrow 2/11 \\
& \downarrow \\
3/10 & \rightarrow y
\end{align*}
\]

\[
\begin{align*}
W & \rightarrow 13/16 \\
& \downarrow \\
14/15 & \rightarrow z
\end{align*}
\]

\[
\begin{align*}
& \rightarrow 7/8 \\
6/9 & \rightarrow r
\end{align*}
\]
Execution of Knuth’s Algorithm

6.1 & 6.2: Graph Searching
Execution of Knuth’s Algorithm

6.1 & 6.2: Graph Searching
Execution of Knuth’s Algorithm

\[ \frac{1}{12} \xrightarrow{4/5} \frac{3}{10} \xrightarrow{7/8} \frac{6}{9} \xrightarrow{13/16} \frac{14/15}{13/16} \xrightarrow{3/10} \frac{2/11}{14/15} \xrightarrow{13/16} \frac{1}{12} \]

6.1 & 6.2: Graph Searching
Execution of Knuth’s Algorithm

6.1 & 6.2: Graph Searching T.S. 16
Execution of Knuth’s Algorithm

4/5 6/9 r 7/8 u 14/15 z

6.1 & 6.2: Graph Searching
Execution of Knuth’s Algorithm

6.1 & 6.2: Graph Searching
Execution of Knuth’s Algorithm

\[ \frac{4}{5} \times \frac{1}{5} = \frac{3}{10} \]

\[ \frac{7}{8} \times \frac{6}{9} = \frac{13}{16} \]

\[ \frac{14}{15} \times \frac{2}{11} = \frac{1}{12} \]
Execution of Knuth’s Algorithm

\begin{align*}
S & \quad 1/12 \quad \rightarrow \quad V \quad 2/11 \quad \rightarrow \quad W \quad 13/16 \\
& \quad 4/5 \quad \rightarrow \quad X \quad 3/10 \quad \rightarrow \quad Y \quad 7/8 \quad \rightarrow \quad U \quad 6/9 \quad \rightarrow \quad Z \quad 14/15 \\
W & \quad 13/16 \quad \rightarrow \quad Z \quad 14/15 \quad \rightarrow \quad S \quad 1/12 \quad \rightarrow \quad V \quad 2/11
\end{align*}
Execution of Knuth’s Algorithm

6.1 & 6.2: Graph Searching
Execution of Knuth’s Algorithm

\[ \frac{4}{5} \times \frac{6}{9} - \frac{7}{8} \]

\[ \frac{3}{10} \]

\[ \frac{14}{15} \]

\[ \frac{13}{16} \]

\[ \frac{2}{11} \]

\[ \frac{3}{10} \]

\[ \frac{1}{12} \]

\[ \frac{4}{5} \]

\[ \frac{3}{10} \]
Execution of Knuth’s Algorithm
Execution of Knuth’s Algorithm

6.1 & 6.2: Graph Searching
Execution of Knuth’s Algorithm

\begin{center}
\begin{tikzpicture}
  \node [fill=black,circle,draw] (S) at (0,0) {$S$};
  \node [fill=black,circle,draw] (V) at (2,0) {$V$};
  \node [fill=black,circle,draw] (W) at (4,0) {$W$};
  \node [fill=black,circle,draw] (X) at (0,-1) {$X$};
  \node [fill=black,circle,draw] (Y) at (2,-1) {$Y$};
  \node [fill=black,circle,draw] (Z) at (4,-1) {$Z$};
  \node [fill=black,circle,draw] (U) at (2,-2) {$U$};
  \node [fill=black,circle,draw] (R) at (4,-2) {$R$};

  \path [->]
  (S) edge node [above] {1/12} (V)
  (V) edge node [above] {2/11} (W)
  (W) edge node [below] {13/16} (R)
  (X) edge node [below] {4/5} (Y)
  (Y) edge node [below] {3/10} (Z)
  (Z) edge node [below] {14/15} (R)
  (S) edge node [above] {13/16} (W)
  (W) edge node [below] {13/16} (R)
  (V) edge node [above] {2/11} (W)
  (W) edge node [below] {13/16} (R)
  (X) edge node [below] {4/5} (Y)
  (Y) edge node [below] {3/10} (Z)
  (Z) edge node [below] {14/15} (R)
  (S) edge node [above] {13/16} (W)
  (W) edge node [below] {13/16} (R)
  (V) edge node [above] {2/11} (W)
  (W) edge node [below] {13/16} (R)
  (X) edge node [below] {4/5} (Y)
  (Y) edge node [below] {3/10} (Z)
  (Z) edge node [below] {14/15} (R)
  (S) edge node [above] {13/16} (W)
  (W) edge node [below] {13/16} (R)
  (V) edge node [above] {2/11} (W)
  (W) edge node [below] {13/16} (R)
  (X) edge node [below] {4/5} (Y)
  (Y) edge node [below] {3/10} (Z)
  (Z) edge node [below] {14/15} (R)

\end{tikzpicture}
\end{center}

6.1 & 6.2: Graph Searching
Execution of Knuth’s Algorithm

6.1 & 6.2: Graph Searching
Execution of Knuth’s Algorithm

6.1 & 6.2: Graph Searching
Execution of Knuth’s Algorithm

6.1 & 6.2: Graph Searching
If the input graph is a DAG, then the algorithm computes a linear order.

Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.
Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:
Correctness of Topological Sort using DFS

Theorem 22.12
If the input graph is a DAG, then the algorithm computes a linear order.

Proof:
- Consider any edge \((u, v) \in E(G)\) being explored,

\[
\text{If } v \text{ is grey, then there is a cycle (can't happen, because } G \text{ is acyclic!).}
\]
\[
\text{If } v \text{ is black, then } v < u.
\]
\[
\text{If } v \text{ is white, we call DFS on } v \text{ and } v < u.
\]

\[
\Rightarrow \text{In all cases } v < u, \text{ so } v \text{ appears after } u.
\]
Correctness of Topological Sort using DFS

Theorem 22.12
If the input graph is a DAG, then the algorithm computes a linear order.

Proof:
- Consider any edge \((u, v) \in E(G)\) being explored,
  \[ \Rightarrow u \text{ is grey and we have to show that } v.f < u.f \]
Correctness of Topological Sort using DFS

Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge \((u, v) \in E(G)\) being explored, 
  \[ \Rightarrow u \text{ is grey and we have to show that } v.f < u.f \]

1. If \(v\) is grey,

   - If \(v\) is grey,
     - there is a cycle (can't happen, because \(G\) is acyclic!).
   - If \(v\) is black,
     - then \(v.f < u.f\).
   - If \(v\) is white,
     - we call DFS \((v)\) and \(v.f < u.f\).

\[ \Rightarrow \text{in all cases } v.f < u.f, \] so \(v\) appears after \(u.f\).
Correctness of Topological Sort using DFS

Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge \((u, v) \in E(G)\) being explored,
  \(\Rightarrow u\) is grey and we have to show that \(v.f < u.f\)

  1. If \(v\) is grey,

  2. If \(v\) is black,

  3. If \(v\) is white, we call DFS \((v)\) and \(v.f < u.f\)

\(\Rightarrow\) In all cases \(v.f < u.f\), so \(v\) appears after \(u.f\).
Theorem 22.12
If the input graph is a DAG, then the algorithm computes a linear order.

Proof:
- Consider any edge \((u, v) \in E(G)\) being explored,
  \(\Rightarrow u\) is grey and we have to show that \(v.f < u.f\)

  1. *If \(v\) is grey, then there is a cycle*  
     *(can’t happen, because \(G\) is acyclic!).*
Correctness of Topological Sort using DFS

Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge \((u, v) \in E(G)\) being explored,
  \[ \Rightarrow u \text{ is grey and we have to show that } v.f < u.f \]

1. If \(v\) is grey, then there is a cycle  
   *(can’t happen, because \(G\) is acyclic!)*.
2. If \(v\) is black,
Correctness of Topological Sort using DFS

Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge \((u, v) \in E(G)\) being explored,
  \[ u \text{ is grey and we have to show that } v.f < u.f \]
  
  1. If \( v \) is grey, then there is a cycle
     (can’t happen, because \( G \) is acyclic!).
  2. If \( v \) is black, then \( v.f < u.f \).
Correctness of Topological Sort using DFS

Theorem 22.12

If the input graph is a DAG, then the algorithm computes a linear order.

Proof:

- Consider any edge \((u, v) \in E(G)\) being explored,
  \[ u \text{ is grey and we have to show that } v.f < u.f \]
  
  1. If \(v\) is grey, then there is a cycle
     *can’t happen, because \(G\) is acyclic!*
  2. If \(v\) is black, then \(v.f < u.f\).
  3. If \(v\) is white,
Theorem 22.12
If the input graph is a DAG, then the algorithm computes a linear order.

Proof:
- Consider any edge \((u, v) \in E(G)\) being explored,
  \(\Rightarrow u\) is grey and we have to show that \(v.f < u.f\)
  1. If \(v\) is grey, then there is a cycle (can’t happen, because \(G\) is acyclic!).
  2. If \(v\) is black, then \(v.f < u.f\).
  3. If \(v\) is white, we call DFS\((v)\) and \(v.f < u.f\).
Correctness of Topological Sort using DFS

Theorem 22.12
If the input graph is a DAG, then the algorithm computes a linear order.

Proof:
- Consider any edge \((u, v) \in E(G)\) being explored,
  \(\Rightarrow u\) is grey and we have to show that \(v.f < u.f\)
  1. If \(v\) is grey, then there is a cycle (can’t happen, because \(G\) is acyclic!).
  2. If \(v\) is black, then \(v.f < u.f\).
  3. If \(v\) is white, we call DFS\((v)\) and \(v.f < u.f\).

  \(\Rightarrow\) In all cases \(v.f < u.f\), so \(v\) appears after \(u\).
Correctness of Topological Sort using DFS

Theorem 22.12
If the input graph is a DAG, then the algorithm computes a linear order.

Proof:
- Consider any edge \((u, v) \in E(G)\) being explored,
  \[\Rightarrow u\text{ is grey and we have to show that } v.f < u.f\]
  1. If \(v\) is grey, then there is a cycle (can’t happen, because \(G\) is acyclic!).
  2. If \(v\) is black, then \(v.f < u.f\).
  3. If \(v\) is white, we call DFS\((v)\) and \(v.f < u.f\).

\[\Rightarrow\text{ In all cases } v.f < u.f, \text{ so } v\text{ appears after } u.\]
Summary of Graph Searching

Breadth-First-Search

- vertices are processed by a queue
- computes distances and shortest paths
  ✈ similar idea used later in Prim’s and Dijkstra’s algorithm
- Runtime $\mathcal{O}(V + E)$
Summary of Graph Searching

**Breadth-First-Search**
- vertices are processed by a queue
- computes distances and shortest paths
  - similar idea used later in Prim’s and Dijkstra’s algorithm
- Runtime $O(V + E)$

**Depth-First-Search**
- vertices are processed by recursive calls ($\approx$ stack)
- discovery and finishing times
- application: Topological Sorting of DAGs
- Runtime $O(V + E)$
Outline

Breadth-First Search

Depth-First Search

Topological Sort

Minimum Spanning Tree Problem
Minimum Spanning Tree Problem

- Given: undirected, connected graph $G = (V, E, w)$ with non-negative edge weights

Applications

6.1 & 6.2: Graph Searching
**Minimum Spanning Tree Problem**

- **Given:** undirected, connected graph \( G = (V, E, w) \) with non-negative edge weights
- **Goal:** Find a subgraph \( \subseteq E \) of minimum total weight that links all vertices

**Applications**

- Street Networks, Wiring Electronic Components, Laying Pipes
- Weights may represent distances, costs, travel times, capacities, resistance etc.
Minimum Spanning Tree Problem

- **Given**: undirected, connected graph $G = (V, E, w)$ with non-negative edge weights
- **Goal**: Find a subgraph $\subseteq E$ of minimum total weight that links all vertices

Must be necessarily a tree!
Minimum Spanning Tree Problem

- **Given**: undirected, connected graph \( G = (V, E, w) \) with non-negative edge weights
- **Goal**: Find a subgraph \( \subseteq E \) of minimum total weight that links all vertices

**Applications**

- Street Networks, Wiring Electronic Components, Laying Pipes
- **Weights** may represent distances, costs, travel times, capacities, resistance etc.
Generic Algorithm

0: def minimum spanningTree(G)
1:     A = empty set of edges
2:     while A does not span all vertices yet:
3:         add a safe edge to A
Generic Algorithm

0: def minimum spanningTree(G)
1:   A = empty set of edges
2:   while A does not span all vertices yet:
3:     add a safe edge to A

Definition

An edge of $G$ is safe if by adding the edge to $A$, the resulting subgraph is still a subset of a minimum spanning tree.
0: def minimum spanningTree(G)
1: A = empty set of edges
2: while A does not span all vertices yet:
3: add a safe edge to A

Definition
An edge of $G$ is safe if by adding the edge to $A$, the resulting subgraph is still a subset of a minimum spanning tree.

How to find a safe edge?
Finding safe edges

Definitions

- a cut is a partition of $V$ into at least two disjoint sets
Finding safe edges

Definitions

- a cut is a partition of $V$ into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of $A$ goes across the cut
Definitions

- a cut is a partition of $V$ into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of $A$ goes across the cut
Finding safe edges

**Definitions**

- A cut is a partition of $V$ into at least two disjoint sets.
- A cut respects $A \subseteq E$ if no edge of $A$ goes across the cut.
Finding safe edges

Definitions

- a cut is a partition of $V$ into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of $A$ goes across the cut

Theorem

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.