Elementary Graph Algorithms

Hosung Jo
Division of Computer and Engineering
Hanyang University
Contents

❖ Graphs
  ● Graphs basics
  ● Graph representation

❖ Searching a graph
  ● Breadth-first search
  ● Depth-first search

❖ Applications of depth-first search
  ● Topological sort
Graph basics

- A **graph** $G$ is a pair $(V, E)$ where $V$ is a **vertex** set and $E$ is an **edge** set.

- A **vertex** (node) is a stand-alone object.
  - Represented by a circle.

- An **edge** (link) is an object connecting two vertices.
  - Represented by either an arrow or a line.

A directed graph

A directed graph

An undirected graph
A *directed graph* (or *digraph*) is a graph with *directed edges*.
- Edges have directions so they are represented by *arrows*.
- Each edge *leaves* a vertex and *enters* a vertex.
- The blue edge leaves vertex 2 and enters vertex 4.
Graph basics

- An edge leaving a vertex $u$ and entering a vertex $v$ is said it is *incident from* $u$ and *incident to* $v$.
  - The blue edge is incident from vertex 2 and to vertex 4.

- In a digraph, *self-loops* (edges from a vertex to itself) are possible.
  - The red edge is a self-loop.
Graph basics

- Normally, each vertex is identified by a number or a name.
  - $V = \{1, 2, 3, 4, 5, 6\}$

- Each edge is identified by the *ordered pair of vertices* it leaves and enters.
  - $E = \{(1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3)\}$

- In a digraph, there are at most 2 edges between two vertices.
An *undirected graph* is a graph with *undirected edges*.
- Edges have no directions so they are represented by *lines*.
- Self-loops are forbidden.
- Edge \((u,v)\) is the same as edge \((v,u)\).
  - \((2,4) = (4,2)\)
  - The blue edge is *incident on* vertices 2 and 4.
**Graph basics**

**Adjacency**
- If \((u,v)\) is an edge, vertex \(v\) is *adjacent* to vertex \(u\).
- In an undirected graph, adjacency relation is symmetric.
  - If \(u\) is adjacent to \(v\), \(v\) is adjacent to \(u\).
- In a directed graph, it is not symmetric.
  - Vertex 2 is *adjacent* to 1.
  - But vertex 1 is *not adjacent* to 2.
**Degree**

- The **out-degree** of a vertex is the number of edges leaving it.
  - The out-degree of vertex 2 is 3.
- The **in-degree** of a vertex is the number of edges entering it.
  - The in-degree of vertex 2 is 2.
- \(\text{degree} = \text{out-degree} + \text{in-degree}\).
Graph basics

**Degree**
- In an undirected graph,
  - The out-degree and the in-degree are not defined.
  - Only the degree of a vertex is defined.
- The degree of vertex 2 is 3.
**Path**

- A path from vertex $u$ to vertex $v$ is a sequence of vertices $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ where
  - $v_0 = u$, $v_k = v$, and
  - every vertex $v_{i+1}$ ($0 \leq i \leq k-1$) is adjacent to $v_i$.
    - There is an edge $(v_i, v_{i+1})$ for all $i$.
- $\langle 1, 2, 4, 5 \rangle$ is a path.
- $\langle 1, 2, 4, 1, 2 \rangle$ is a path.
- $\langle 1, 2, 4, 2 \rangle$ is not a path.
**Path**
- The **length** of a path is the number of edges in the path.
  - The length of a path \(<1, 2, 4, 5>\) is 3.
  - If there is a path from vertex \(u\) to vertex \(v\), \(v\) is called **reachable** from \(u\).
- Vertex 5 is reachable from vertex 1.
- Vertex 3 is not reachable from vertex 1.
**Graph basics**

**Simple path**
- A path is simple if all vertices in the path are distinct.
- A path $<1, 2, 4, 5>$ is a simple path.
- A path $<1, 2, 4, 1, 2>$ is not a simple path.
**Graph basics**

**Cycle and simple cycle**

- A path \(<v_0, v_1, v_2, \ldots, v_k>\) is a cycle if \(v_0 = v_k\)
- A cycle \(<v_0, v_1, v_2, \ldots, v_k>\) is simple if \(v_1, v_2, \ldots, v_k\) are distinct.
- A path \(<1, 2, 4, 5, 4, 1>\) is a cycle but it is not a simple cycle.
- A path \(<1, 2, 4, 1>\) is a simple cycle.
Graph basics

- **An acyclic graph**
  - A graph without cycles

- **A connected graph**
  - An undirected graph is *connected* if every pair of vertices is connected by a path.

- **Connected components**
  - Maximally connected subsets of vertices of an undirected graph.
Graph basics

Strongly connected

- A directed graph is *strongly connected* if every pair of vertices is reachable from each other.

Strongly connected components

- Maximally strongly connected subsets of vertices in a directed graph.
Directed version of an undirected graph

- Replace each undirected edge $(u,v)$ by two directed edges $(u,v)$ and $(v,u)$. 
**Undirected version** of a directed graph
- Replace each directed edge \((u,v)\) by an undirected edge \((u,v)\)
Graph basics

Undirected graph $G \rightarrow$ directed ver. $G' \rightarrow$ undirected ver. $G''$
- Are $G$ and $G''$ the same?

Directed graph $G \rightarrow$ undirected ver. $G' \rightarrow$ directed ver. $G''$
- Are $G$ and $G''$ the same?
A complete graph

- An undirected graph in which every pair of vertices is adjacent.

- The number of edges with \( n \) vertices?
A bipartite graph

An undirected graph $G = (V,E)$ in which $V$ can be partitioned into two sets $V_1$ and $V_2$ such that for each edge $(u,v)$, either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. 

![Graph diagram](image-url)
Graph basics

- **Forest**
  - An acyclic, undirected graph

- **Tree**
  - A connected forest
  - A connected, acyclic, undirected graph
Graph basics

**Dag**
- A directed acyclic graph

![Diagram of a directed acyclic graph]

**Handshaking lemma**
- If $G = (V, E)$ is an undirected graph

\[
\sum_{v \in V} \text{degree}(v) = 2 \cdot |E|
\]
Graph basics

**Tree: connected, acyclic, and undirected graph**
- Any two vertices are connected by a unique simple path.
- If any edge is removed, the resulting graph is disconnected.
- If any edge is added, the resulting graph contains a cycle.
- $|E| = |V| - 1$
Graph basics

*G is a tree.*

= \( G \) is a connected, acyclic, and undirected graph

= In \( G \), any two vertices are connected by a unique simple path.

= \( G \) is connected, and if any edge is removed, the resulting graph is disconnected.

= \( G \) is connected, \( |E| = |V| - 1 \).

= \( G \) is acyclic, \( |E| = |V| - 1 \).

= \( G \) is acyclic, but if any edge is added, the resulting graph contains a cycle.
Graph basics

- The number of edges
  - Directed graph
    - $|E| \leq |V|^2$
  - Undirected graph
    - $|E| \leq |V| (|V|-1) / 2$
Contents

Graphs
- Graphs basics
- Graph representation

Searching a graph
- Breadth-first search
- Depth-first search

Applications of depth-first search
- Topological sort
Graph representation

Representations of graphs

- Adjacency-list representation
- Adjacency-matrix representation
Graph representation

**Adjacency-list representation**
- An array of $|V|$ lists, one for each vertex.
- For vertex $u$, its adjacency list contains all vertices adjacent to $u$. 

A directed graph

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Adjacency-list representation

- For an undirected graph, its directed version is stored.

An undirected graph

- $\Theta(V + E)$ space
Graph representation

**Adjacency-matrix representation**

- $|V| \times |V|$ matrix: $\Theta(V^2)$ space
- Entry $(i,j)$ is 1 if there is an edge and 0 otherwise.

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

A directed graph
Graph representation

**Adjacency-matrix representation**

- $|V| \times |V|$ matrix
- Entry $(i,j)$ is 1 if there is an edge and 0 otherwise.

An undirected graph
Adjacency-matrix representation

- For an undirected graph, there is a symmetry along the main diagonal of its adjacency matrix.
- Storing the lower matrix is enough.

An undirected graph
Comparison of adjacency list an adjacency matrix

- Storage
  - If $G$ is sparse, adjacency list is better.
    - because $|E| < |V|^2$.
  - If $G$ is dense, adjacency matrix is better.
    - because adjacency matrix uses only one bit for an entry.

- Edge present test: does an edge $(i,j)$ exist?
  - Adjacency matrix: $\Theta(1)$ time.
  - Adjacency list: $O(|V|)$ time.
Graph representation

- Comparison of adjacency list and adjacency matrix
  - Listing or visiting all edges
    - Adjacency matrix: $\Theta(V^2)$ time.
    - Adjacency list: $\Theta(V + E)$ time.
Graph representation

- Weighted graph
  - Edges have weights.
**Graph representation**

- **Weighted graph representation**
  - adjacency list

```
  a  b  c  d  e
  a  | 2  |   |   |   |
  b  |   | 2  | 5  | 2  |
  c  | 4  | 5  | 1  | 1  |
  d  | 2  |   | 3  | 3  |
  e  | 5  |   | 3  |   |
```

```plaintext
Weighted graph representation
- adjacency list

```
```

```plaintext
```

```plaintext
```

```plaintext
```

```plaintext
```

```plaintext
```

```plaintext
```

```plaintext
```

```plaintext
```

```plaintext
```

```plaintext
```

```plaintext
```

```plaintext
```

```plaintext
```
Graph representation

- Weighted graph representation
  - adjacency matrix
  - $\Theta(V^2)$ space

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Graph representation

- **Transpose of a matrix**
  - The *transpose* of a matrix $A = (a_{ij})$ is
  - $A^T = (a_{ij}^T)$ where $a_{ij}^T = a_{ji}$
  - An undirected graph is its own transpose: $A = A^T$.

\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{bmatrix}^T =
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{12} & a_{22} & a_{23} \\
  a_{13} & a_{23} & a_{33}
\end{bmatrix}
\]
Self-study

Exercise 22.1-3
- The transpose of a directed graph

Exercise 22.1-4
- Removing duplicate edges in a multigraph in $O(V + E)$ time.

Exercise 22.1-6
- Universal sink detection in $O(V)$ time.
Contents

Graphs
- Graphs basics
- Graph representation

Searching a graph
- Breadth-first search
- Depth-first search

Applications of depth-first search
- Topological sort
Searching a tree

- Breadth-first search
- Depth-first search
Breadth-first search

**Distance**
- Distance from $u$ to $v$
  - The number of edges in the shortest path from $u$ to $v$.
  - The distance from $s$ to $v$ is 2.
Breadth-first search

- **Breadth-first search**
  - Given a graph $G = (V, E)$ and a source vertex $s$, it explores the edges of $G$ to "discover" every reachable vertex from $s$.
  - It discovers vertices in the increasing order of distance from the source. It first discovers all vertices at distance 1, then 2, and etc.
Breadth-first search

- **Breadth-first search**
  
  - Given a graph $G = (V, E)$ and a *source* vertex $s$, it explores the edges of $G$ to "discover" every reachable vertex from $s$.
  
  - It discovers vertices in the increasing order of distance from the source. It first discovers all vertices at distance 1, then 2, and etc.

![Diagram](image)
Breadth-first search

- **Breadth-first search**
  - Given a graph $G = (V, E)$ and a source vertex $s$, it explores the edges of $G$ to "discover" every reachable vertex from $s$.
  - It discovers vertices in the increasing order of distance from the source. It first discovers all vertices at distance 1, then 2, and etc.
Breadth-first search

- **Breadth-first search**
  - Given a graph \( G = (V, E) \) and a *source* vertex \( s \), it explores the edges of \( G \) to "discover" every reachable vertex from \( s \).
  - It discovers vertices in the increasing order of distance from the source. It first discovers all vertices at distance 1, then 2, and etc.
Breadth-first search

- It also computes
  - the distance of vertices from the source: \( u.d = 3 \)
  - the predecessor of vertices: \( u.\pi = t \)
The predecessor subgraph of $G$ as $G_\pi = (V_\pi, E_\pi)$,

- $V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$
- $E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$. 

Breadth-first search
The predecessor subgraph $G_{\pi}$ is a **breadth-first tree**.
- since it is connected and $|E_{\pi}| = |V_{\pi}| - 1$.
- The edges in $E_{\pi}$ are called **tree edges**.
Breadth-first search

\textbf{BFS}(G, s)

1. \textbf{for} each vertex \( u \in G.V - \{ s \} \)
2. \hspace{1em} \( u.color = \text{WHITE} \)
3. \hspace{1em} \( u.d = \infty \)
4. \hspace{1em} \( u.\pi = \text{NIL} \)
5. \( s.color = \text{GRAY} \)
6. \( s.d = 0 \)
7. \( s.\pi = \text{NIL} \)
8. \( Q = \emptyset \)
9. \text{ENQUEUE}(Q, s)
10. \textbf{while} \( Q \neq \emptyset \)
11. \hspace{1em} \( u = \text{DEQUEUE}(Q) \)
12. \hspace{1em} \textbf{for} each \( v \in G.Adj[u] \)
13. \hspace{2em} \textbf{if} \( v.color = \text{WHITE} \)
14. \hspace{3em} \( v.color = \text{GRAY} \)
15. \hspace{3em} \( v.d = u.d + 1 \)
16. \hspace{3em} \( v.\pi = u \)
17. \hspace{3em} \text{ENQUEUE}(Q, v) \)
18. \hspace{1em} \( u.color = \text{BLACK} \)
Breadth-first search

\[ r \quad s \quad t \quad u \]

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

\[ \infty \quad 0 \quad \infty \quad \infty \]

\[ \infty \quad \infty \quad \infty \quad \infty \]

\[ Q \quad s \]

\[ 0 \]
Breadth-first search

- white: not discovered (not entered the Q)
- gray: discovered (in the Q)
- black: finished (out of the Q)
Breadth-first search
Breadth-first search

Q

1 2 2

r s t u

1 0 2

v w x y
Breadth-first search

<table>
<thead>
<tr>
<th>r</th>
<th>s</th>
<th>t</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>∞</td>
</tr>
</tbody>
</table>

Q: [t, x]

Q: [t, x]

<table>
<thead>
<tr>
<th>r</th>
<th>s</th>
<th>v</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>r</td>
<td>w</td>
<td>/</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t</th>
<th>u</th>
<th>w</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>t</td>
<td>x</td>
<td>y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>r</th>
<th>/</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>w</th>
<th>s</th>
<th>t</th>
<th>/</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>t</th>
<th>u</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>x</td>
<td>/</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>y</th>
<th>u</th>
<th>/</th>
<th>y</th>
</tr>
</thead>
</table>

56
Breadth-first search
Breadth-first search

1. Breadth-first search is a graph traversal algorithm that explores the graph level by level, starting from the root node.

2. It visits every node at a given depth before moving on to the nodes at the next depth.

3. The algorithm uses a queue to keep track of the nodes to be visited. Initially, the queue contains only the root node.

4. At each step, the algorithm removes the first node from the queue and visits it. Then, it adds all undiscovered neighbors of the node to the queue.

5. This process continues until the queue is empty, indicating that all reachable nodes have been visited.

Example:

Consider the graph shown on the left. The Breadth-first search algorithm would visit the nodes in the order: r → u → s → t → v → w → x → y → z.

The queue Q would be updated as follows:

- Initial: Q = [r]
- After r: Q = [u, s]
- After u: Q = [t, v]
- After v: Q = [w, x]
- After w: Q = [y, z]

This demonstrates how Breadth-first search explores the graph layer by layer, ensuring that all nodes at a given level are visited before moving on to the next level.
Breadth-first search
Breadth-first search
Breadth-first search

**BFS**(*G*, *s*)
1. **for** each vertex *u* ∈ *G*.*V* − {*s*}
2. \( u.\text{color} = \text{WHITE} \)
3. \( u.d = \infty \)
4. \( u.\pi = \text{NIL} \)
5. \( s.\text{color} = \text{GRAY} \)
6. \( s.d = 0 \)
7. \( s.\pi = \text{NIL} \)
8. \( Q = \emptyset \)
9. **ENQUEUE**(\( Q \), *s*)
10. **while** \( Q \neq \emptyset \)
11. \( u = \text{DEQUEUE}(Q) \)
12. **for** each *v* ∈ *G*.\( \text{Adj}[u] \)
13. \[ \text{if } v.\text{color} == \text{WHITE} \]
14. \( v.\text{color} = \text{GRAY} \)
15. \( v.d = u.d + 1 \)
16. \( v.\pi = u \)
17. **ENQUEUE**(\( Q \), *v*)
18. \( u.\text{color} = \text{BLACK} \)
Breadth-first search

- **Running time**
  - **Initialization**: $\Theta(V)$

  - **Exploring the graph**: $O(V + E)$
    - A vertex is examined at most once.
    - An edge is explored at most twice.

  - **Overall**: $O(V + E)$
Self-study

- **Exercise 22.2-4  (22.2-3 in the 2nd ed.)**
  - The running time of BFS with adjacency matrix representation.

- **Exercise 22.2-6  (22.2-5 in the 2nd ed.)**
  - Impossible breadth-first trees.

- **Exercise 22.2-7  (22.2-6 in the 2nd ed.)**
  - Rivalry
Contents

- **Graphs**
  - Graphs basics
  - Graph representation

- **Searching a graph**
  - Breadth-first search
  - Depth-first search

- **Applications of depth-first search**
  - Topological sort
Colors of vertices

- Each vertex is initially *white* (not discovered).
- The vertex is *grayed* when it is *discovered*.
- The vertex is *blackened* when it is *finished*, that is, when its adjacency list has been examined completely.
Depth-first search

**Timestamps**

- Each vertex $v$ has two timestamps.
  - $v.d$: *discovery time* (when $v$ is grayed)
  - $v.f$: *finishing time* (when $v$ is blacken)
Depth-first search

(a)

(b)

(c)

(d)
Depth-first search

Diagram (e):
- Node u
- Node v
- Node w
- Node x
- Node y
- Node z

Diagram (f):
- Node u
- Node v
- Node w
- Node x
- Node y
- Node z

Diagram (g):
- Node u
- Node v
- Node w
- Node x
- Node y
- Node z

Diagram (h):
- Node u
- Node v
- Node w
- Node x
- Node y
- Node z

68
Depth-first search

(i)

(ii)

(iii)

(iv)
Depth-first search

(m)

(n)

(o)

(p)
Depth-first search

- The *predecessor subgraph* is a *depth-first forest*.
Depth-first search

**Parenthesis theorem (for gray interval)**

- **Inclusion**: The ancestor’s includes the descendants’.
- **Disjoint**: Otherwise.
Depth-first search

Classification of edges

- Tree edges
- Back edges
- Forward edges
- Cross edges
Classification of edges

- **Tree edges**: Edges in the depth-first forest.

- **Back edges**: Those edges \((u, v)\) connecting a vertex \(u\) to an ancestor \(v\) in a depth-first tree. Self-loops are considered to be back edges.

- **Forward edges**: Those edges \((u, v)\) connecting a vertex \(u\) to a descendant \(v\) in a depth-first tree.

- **Cross edges**: All other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.
Classification by the DFS algorithm

- Each edge \((u, v)\) can be classified by the color of the vertex \(v\) that is reached when the edge is first explored:
  - **white** indicates a tree edge,
  - **gray** indicates a back edge, and
  - **black** indicates a forward or cross edge.

- Forward and cross edges are classified by the *inclusion of gray intervals of \(u\) and \(v\).*
Depth-first search
Depth-first search
Depth-first search

(i)

(ii)

(iii)

(iv)
Depth-first search
Depth-first search

- In a depth-first search of an **undirected graph**, every edge of G is either a **tree edge** or a **back edge**.

  - Forward edge?

  - Cross edge?

- **Running Time**
  - $\Theta(V+E)$
Self-study

Exercise 22.3-5 (22.3-4 in the 2\textsuperscript{nd} ed.)
- Edge classification

Problem 22-2 \textit{a-d}
- Articulation points
Contents

- **Graphs**
  - Graphs basics
  - Graph representation

- **Searching a graph**
  - Breadth-first search
  - Depth-first search

- **Applications of depth-first search**
  - Topological sort
**Definition**
- Given a DAG (directed acyclic graph), generate a linear ordering of all its vertices such that all edges go from left to right.
Topological sort
**Main ideas**

- Successively place a node from the *left* with 0 *in-degree*.
- Successively place a node from the *right* with 0 *out-degree.*
- Run DFS on G and place the nodes from the *right* in the *increasing order of the finishing time*.
- $\Theta(V+E)$ time
Correctness

- If there is an edge from $u$ to $v$, then $v.f < u.f$.

- A directed graph $G$ is **acyclic** if and only if a depth-first search of $G$ yields *no back edges*. 
Self-study

**Exercise 22.4-2**
- Computing the number of simple paths from $s$ to $t$ in linear time.

**Exercise 22.4-3**
- Cycle detection in an undirected graph.

**Exercise 22.4-5**
- Another topological sort algorithm.
Programming Assignment

- **Depth-first search and its applications**
  - Exercise 22.3-10 (22.3-9, 2nd ed.) (#1)
    - Depth-first search with edge classification
  - Exercise 22.3-12 (22.3-11, 2nd ed.) (#2)
    - Connected component identification
  - **Topological sort (#3)**
    - The program should detect whether the input is a DAG or not.