1 Course Information
2 Introduction
3 Divide-and-Conquer Algorithms
4 Greedy Algorithms
5 Dynamic Programming Algorithms
6 Graph Algorithms
7 Intractability and Undecidability
Table of Contents

Graph Algorithms
- Graphs
- Breadth-first Search
- Depth-first Search
- Topological Ordering
- Strongly Connected Components
- Minimum Spanning Trees
- Single Source Shortest Paths
- All-Pairs Shortest Paths
Graphs and Digraphs

A graph is a pair $G = (V, E)$. $V$ is a set whose elements are called vertices and $E$ is a set whose elements are called edges. Each edge joins two distinct vertices. An edge can be represented as a set of two vertices, e.g., $\{u, v\}$, where $u \neq v$. We may also write this edge as $uv$ or $vu$.

We often denote the number of vertices by $n$ and the number of edges by $m$. Clearly $m \leq \binom{n}{2}$.

A directed graph or digraph is also a pair $G = (V, E)$. The elements of $E$ are called directed edges or arcs in a digraph. Each arc joins two vertices, and an arc can be represented as an ordered pair, e.g., $(u, v)$. The arc $(u, v)$ is directed from $u$ (the tail) to $v$ (the head), and we allow $u = v$.

If we denote the number of vertices by $n$ and the number of arcs by $m$, then $m \leq n^2$. 
**Data Structures for Graphs: Adjacency Matrices**

There are two main data structures to represent graphs: an **adjacency matrix** and a set of **adjacency lists**.

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$. The **adjacency matrix** of $G$ is an $n$ by $n$ matrix $A = (a_{u,v})$, which is indexed by $V$, such that

$$a_{u,v} = \begin{cases} 
1 & \text{if}\ \{u, v\} \in E \\
0 & \text{otherwise.}
\end{cases}$$

There are exactly $2m$ entries of $A$ equal to 1.

If $G$ is a digraph, then

$$a_{u,v} = \begin{cases} 
1 & \text{if}\ \(u, v\) \in E \\
0 & \text{otherwise.}
\end{cases}$$

For a digraph, there are exactly $m$ entries of $A$ equal to 1.
Example

\( V = \{1, 2, 3, 4, 5, 6, 7, 8\} \) and \( E = \{12, 13, 23, 24, 25, 35, 37, 38, 56, 78\} \).

The adjacency matrix is

\[
\begin{pmatrix}
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
\end{pmatrix}
\]
Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$.

An adjacency list representation of $G$ consists of $n$ linked lists.

For every $u \in V$, there is a linked list (called an adjacency list) which is named $\text{Adj}[u]$.

For every $v \in V$ such that $uv \in E$, there is a node in $\text{Adj}[u]$ labelled $v$. (This definition is used for both directed and undirected graphs.)

In an undirected graph, every edge $uv$ corresponds to nodes in two adjacency lists: there is a node $v$ in $\text{Adj}[u]$ and a node $u$ in $\text{Adj}[v]$.

In a directed graph, every edge corresponds to a node in only one adjacency list.
Example

The adjacency lists for the previous graph are

\[
\begin{align*}
Adj[1] : & \quad 2 \rightarrow 3 \\
Adj[2] : & \quad 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \\
Adj[3] : & \quad 1 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 8 \\
Adj[4] : & \quad 2 \\
Adj[5] : & \quad 2 \rightarrow 3 \rightarrow 6 \\
Adj[6] : & \quad 5 \\
Adj[7] : & \quad 3 \rightarrow 8 \\
Adj[8] : & \quad 3 \rightarrow 7
\end{align*}
\]
Breadth-first Search of an Undirected Graph

A **breadth-first search** of an undirected graph begins at a specified vertex $s$.

The search “spreads out” from $s$, proceeding in **layers**.

First, all the neighbours of $s$ are **explored**.

Next, the neighbours of those neighbours are explored.

This process continues until all vertices have been explored.

A **queue** is used to keep track of the vertices to be explored.
Breadth-first Search

Algorithm: \(\text{BFS}(G, s)\)
for each \(v \in V(G)\)
do \(\begin{cases}
\text{colour}[v] \leftarrow \text{white} \\
\pi[v] \leftarrow \emptyset
\end{cases}\)
colour\([s]\) \leftarrow \text{gray}

\text{InitializeQueue}(Q)
\text{Enqueue}(Q, s)

while \(Q \neq \emptyset\)
do \(\begin{cases}
u \leftarrow \text{Dequeue}(Q) \\
\text{for each } v \in \text{Adj}[u]
do \begin{cases}
\text{if colour}[v] = \text{white}
do \begin{cases}
\text{then } \begin{cases}
colour[v] = \text{gray} \\
\pi[v] \leftarrow u \\
\text{Enqueue}(Q, v)
\end{cases}
colour[u] \leftarrow \text{black}
\end{cases}
\end{cases}
\end{cases}
Example

We run *breadth-first search* with $s = 1$ on the previous graph:

\[
\begin{align*}
\text{colour}[1] & \leftarrow \text{grey}, Q = [1] \\
u & \leftarrow 1, Q = [] \\
v & \leftarrow 2, \text{colour}[2] \leftarrow \text{grey}, \pi[2] \leftarrow 1, Q = [2] \\
v & \leftarrow 3, \text{colour}[3] \leftarrow \text{grey}, \pi[3] \leftarrow 1, Q = [2, 3] \\
\text{colour}[1] & \leftarrow \text{black} \\
u & \leftarrow 2, Q = [3] \\
v & \leftarrow 4, \text{colour}[4] \leftarrow \text{grey}, \pi[4] \leftarrow 2, Q = [3, 4] \\
v & \leftarrow 5, \text{colour}[5] \leftarrow \text{grey}, \pi[5] \leftarrow 2, Q = [3, 4, 5] \\
\text{colour}[2] & \leftarrow \text{black} \\
u & \leftarrow 3, Q = [4, 5] \\
v & \leftarrow 7, \text{colour}[7] \leftarrow \text{grey}, \pi[7] \leftarrow 3, Q = [4, 5, 7] \\
v & \leftarrow 8, \text{colour}[8] \leftarrow \text{grey}, \pi[8] \leftarrow 3, Q = [4, 5, 7, 8] \\
\text{colour}[3] & \leftarrow \text{black}
\end{align*}
\]
Example (cont.)

\[
\begin{align*}
\text{u} & \leftarrow 4, Q = [5, 7, 8] \\
\text{colour}[4] & \leftarrow \text{black} \\
\text{u} & \leftarrow 5, Q = [7, 8] \\
\quad \text{v} & \leftarrow 6, \text{colour}[6] \leftarrow \text{grey}, \pi[6] \leftarrow 5, Q = [7, 8, 6] \\
\text{colour}[5] & \leftarrow \text{black} \\
\text{u} & \leftarrow 7, Q = [8, 6] \\
\text{colour}[7] & \leftarrow \text{black} \\
\text{u} & \leftarrow 8, Q = [6] \\
\text{colour}[8] & \leftarrow \text{black} \\
\text{u} & \leftarrow 6, Q = [] \\
\text{colour}[6] & \leftarrow \text{black}
\end{align*}
\]

The tree edges are 12, 13, 24, 25, 37, 38, 56.
Properties of Breadth-first Search

A vertex is **white** if it is **undiscovered**.

A vertex is **gray** if it has been **discovered**, but we are still processing its adjacent vertices.

A vertex becomes **black** when all the adjacent vertices have been processed.

If $G$ is **connected**, then every vertex eventually is coloured black and every vertex $v \neq s$ has a unique predecessor $\pi[v]$ in the BFS tree.

When we explore an edge $\{u, v\}$ starting from $u$:

- if $v$ is **white**, then $uv$ is a **tree edge** and $\pi[v] = u$ is the **predecessor** of $v$ in the **BFS tree**
- otherwise, $uv$ is a **cross edge**.

The BFS tree consists of all the tree edges.
Shortest Paths via Breadth-first Search

Algorithm: \( \text{BFS}(G, s) \)

for each \( v \in V(G) \) do

\[
\begin{align*}
\text{colour}[v] &\leftarrow \text{white} \\
\pi[v] &\leftarrow \emptyset
\end{align*}
\]

\( \text{colour}[s] \leftarrow \text{gray} \)

\( \text{dist}[s] \leftarrow 0 \)

\text{InitializeQueue}(Q)

\text{Enqueue}(Q, s)

while \( Q \neq \emptyset \)

\[
\begin{align*}
\text{Dequeue}(Q)\quad &\text{do}
\quad \text{for each } v \in \text{Adj}[u] \quad \text{do}
\quad \text{if } \text{colour}[v] = \text{white} \quad \text{then}
\quad \text{colour}[u] \leftarrow \text{black}
\end{align*}
\]

\[
\begin{align*}
\text{colour}[v] &\leftarrow \text{gray} \\
\pi[v] &\leftarrow u \\
\text{Enqueue}(Q, v) \\
\text{dist}[v] &\leftarrow \text{dist}[u] + 1
\end{align*}
\]
Distances in Breadth-first Search

Lemma 6.1

If \( u \) is discovered before \( v \), then \( \text{dist}[u] \leq \text{dist}[v] \).

Proof.

By contradiction. Let \( v \) be the first vertex such that \( \text{dist}[u] > \text{dist}[v] \) for some \( u \) that was discovered before \( v \). Denote \( d = \text{dist}[v] \); then \( \text{dist}[u] \geq d + 1 \). Let \( \pi[v] = v_1 \); then \( \text{dist}[v_1] = d - 1 \). Let \( \pi[u] = u_1 \); then \( \text{dist}[u_1] \geq d \). Note that \( v_1 \) was discovered before \( u_1 \) since \( v \) is the first “out-of-order” vertex. So, in order of discovery, we have \( v_1, u_1, u, v \). Vertex \( v \) was discovered while processing \( \text{Adj}[v_1] \) and vertex \( u \) was discovered while processing \( \text{Adj}[u_1] \). This means that \( v \) was discovered before \( u \), a contradiction.
Distances in Breadth-first Search (cont.)

Lemma 6.2

If \( \{u, v\} \) is any edge, then \(|dist[u] - dist[v]| \leq 1\).

Proof.

WLOG suppose \( u \) is discovered before \( v \).

(1) \( v \) is white when we process \( Adj[u] \). Then \( dist[v] = dist[u] + 1 \).

(2) \( v \) is grey when we process \( Adj[u] \). Let \( \pi[v] = v_1 \); then \( v \) was discovered when \( Adj[v_1] \) was being processed. So \( v_1 \) was discovered before \( u \). By Lemma 1, \( dist[v_1] \leq dist[u] \). Also, \( dist[v] = dist[v_1] + 1 \), so \( dist[u] \geq dist[v] - 1 \). Since \( u \) was discovered before \( v \), we have \( dist[u] \leq dist[v] \) by Lemma 1. Therefore, \( dist[u] \leq dist[v] \leq dist[u] + 1 \).

(3) \( v \) is black when we process \( Adj[u] \). Then \( Adj[v] \) has been completely processed and we would already have discovered \( u \) from \( v \); contradiction.
Theorem 6.3

\( \text{dist} [v] \) is the length of the shortest path from \( s \) to \( v \).

Proof.

Let \( \delta(v) \) denote the length of the shortest path from \( s \) to \( v \). Consider the path \( v \pi [v] \pi [\pi [v]] \cdots s \). This path has distance \( \text{dist} [v] \), so \( \delta(v) \leq \text{dist} [v] \). To complete the proof, we show that \( \delta(v) \geq \text{dist} [v] \); we will prove this by induction on \( \delta(v) \).

Base case: \( \delta(v) = 0 \). Then \( v = s \) and \( \text{dist} [v] = 0 = \delta(v) \).

Induction assumption: Assume \( \delta(v) \geq \text{dist} [v] \) if \( \delta(v) \leq d - 1 \). Now suppose \( \delta(v) = d \). Let \( s \ v_1 \ v_2 \ \cdots \ v_{d-1} \ v_d = v \) be a shortest path (having length \( d \)). Then \( \delta(v_{d-1}) = d - 1 = \text{dist} [v_{d-1}] \) by induction. We have that \( \text{dist} [v] \leq \text{dist} [v_{d-1}] + 1 \) (by Lemma 2). But \( \text{dist} [v_{d-1}] = d - 1 \), so \( \text{dist} [v] \leq d = \delta(v) \) and we’re done.
Bipartite Graphs and Breadth-first Search

A graph is **bipartite** if the vertex set can be partitioned as $V = X \cup Y$, in such a way that all edges have one endpoint in $X$ and one endpoint in $Y$.

A graph is bipartite if and only if it does not contain an **odd cycle**.

**BFS** can be used to test if a graph is bipartite:

- if we encounter an edge $\{u, v\}$ with $dist[u] = dist[v]$, then $G$ is not bipartite, whereas
- if no such edge is found, then define $X = \{u : dist[u] \text{ is even}\}$ and $Y = \{u : dist[u] \text{ is odd}\}$; then $X, Y$ forms a bipartition.
Bipartite Graphs

Theorem 6.4

A graph is bipartite if and only if it contains no cycle of odd length.

Proof.

(⇒): Suppose $G$ contains an odd cycle, $v_1, v_2, \ldots, v_{2k+1}, v_1$. WLOG colour $v_1$ red. Then we are forced to colour $v_2$ blue, $v_3$ red, $\ldots$, and $v_{2k+1}$ red. Then the edge $v_{2k+1}v_1$ joins two red vertices, so $G$ is not bipartite.

(⇐): Suppose $G$ is not bipartite. WLOG assume $G$ is connected. Let $s$ be any vertex. Define $X = \{v : \text{dist}[v] \text{ is even}\}$ and $Y = \{v : \text{dist}[v] \text{ is odd}\}$. Since $G$ is not bipartite, there is an edge $uv$ where $u, v \in X$ or $u, v \in Y$. $	ext{dist}[u]$ and $	ext{dist}[v]$ are both even or both odd, so $	ext{dist}[u] = \text{dist}[v]$ since $|\text{dist}[u] - \text{dist}[v]| \leq 1$ for any edge $uv$. Denote $d = \text{dist}[u] = \text{dist}[v]$. $u, u_1 = \pi[u], \ldots, u_d = \pi[u_{d-1}] = s$ and $v, v_1 = \pi[v], \ldots, v_d = \pi[v_{d-1}] = s$ are paths of length $d$ in $G$. Let $j = \min\{i : u_i = v_i\}$ ($j \leq d$ since $u_d = v_d = s$). Then $v_i, \ldots, v_1, v, u, u_1, \ldots, u_i = v_i$ is an odd cycle.
Depth-first Search of a Directed Graph

A depth-first search uses a stack (or recursion) instead of a queue. We define predecessors and colour vertices as in BFS. It is also useful to specify a discovery time $d[v]$ and a finishing time $f[v]$ for every vertex $v$. We increment a time counter every time a value $d[v]$ or $f[v]$ is assigned. We eventually visit all the vertices, and the algorithm constructs a depth-first forest.

The complexity of depth-first search is $\Theta(n + m)$. 
Depth-first Search

Algorithm: \( DFS(G) \)
for each \( v \in V(G) \)
do \{ 
  colour[v] \leftarrow \text{white}
  \pi[v] \leftarrow \emptyset
\}
time \leftarrow 0
for each \( v \in V(G) \)
do \{ 
  if colour[v] = \text{white} 
  then \text{DFSvisit}(v)
\}
Depth-first Search (cont.)

**Algorithm**: \( DFSvisit(v) \)
- \( colour[v] \leftarrow \text{gray} \)
- \( time \leftarrow time + 1 \)
- \( d[v] \leftarrow time \)
  
  **comment**: \( d[v] \) is the discovery time for vertex \( v \)

  **for each** \( w \in Adj[v] \)
  - \( \begin{cases} 
    \text{if} \ colour[w] = \text{white} \\
    \text{then} \ \{ \pi[w] \leftarrow v \\
    DFSvisit(w) \} 
  \end{cases} \)
  
  \( colour[v] \leftarrow \text{black} \)
- \( time \leftarrow time + 1 \)
- \( f[v] \leftarrow time \)
  
  **comment**: \( f[v] \) is the finishing time for vertex \( v \)
Example of Depth-first Search

Consider the directed graph on vertex set \( \{1, 2, 3, 4, 5, 6\} \) with the following adjacency lists:

\[
\begin{align*}
Adj[1] & : 2 \rightarrow 3 \\
Adj[2] & : 3 \\
Adj[3] & : 4 \\
Adj[4] & : 2 \\
Adj[5] & : 4 \rightarrow 6 \\
Adj[6] & : \\
\end{align*}
\]

Initial call: \( \text{DFSvisit}(1) \), recursive calls: \( \text{DFSvisit}(2), \text{DFSvisit}(3), \text{DFSvisit}(4) \).

Initial call: \( \text{DFSvisit}(5) \), recursive call: \( \text{DFSvisit}(6) \).

The depth-first forest consists of two trees. One tree has arcs 12, 23, 34 (initial call from \( \text{DFSvisit}(1) \)) and the other tree has arc 56 (initial call from \( \text{DFSvisit}(5) \)).
Classification of Edges in Depth-first Search

- $uv$ is a **tree edge** if $u = \pi[v]$
- $uv$ is a **forward edge** if it is not a tree edge, and $v$ is a descendant of $u$ in a tree in the depth-first forest
- $uv$ is a **back edge** if $u$ is a descendant of $v$ in a tree in the depth-first forest
- any other edge is a **cross edge**.
Properties of Edges in Depth-first Search

In the following table, we indicate the colour of a vertex $v$ when an edge $uv$ is discovered, and the relation between the start and finishing times of $u$ and $v$, for each possible type of edge $uv$.

<table>
<thead>
<tr>
<th>edge type</th>
<th>colour of $v$</th>
<th>discovery/finish times</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>white</td>
<td>$d[u] &lt; d[v] &lt; f[v] &lt; f[u]$</td>
</tr>
<tr>
<td>forward</td>
<td>black</td>
<td>$d[u] &lt; d[v] &lt; f[v] &lt; f[u]$</td>
</tr>
<tr>
<td>back</td>
<td>gray</td>
<td>$d[v] &lt; d[u] &lt; f[u] &lt; f[v]$</td>
</tr>
<tr>
<td>cross</td>
<td>black</td>
<td>$d[v] &lt; f[v] &lt; d[u] &lt; f[u]$</td>
</tr>
</tbody>
</table>

Observe that two intervals $(d[u], f[u])$ and $(d[v], f[v])$ never overlap. Two intervals are either disjoint or nested. This is sometimes called the parenthesis theorem.
Topological Orderings and DAGs

A directed graph \( G \) is a **directed acyclic graph**, or **DAG**, if \( G \) contains no directed cycle.

A directed graph \( G = (V, E) \) has a **topological ordering**, or **topological sort**, if there is a linear ordering \(<\) of all the vertices in \( V \) such that \( u < v \) whenever \( uv \in E \).

Here is a topological ordering—all edges are directed from left to right:
Some Interesting/useful Facts

Lemma 6.5

A DAG contains a vertex of indegree 0.

Proof.

Suppose we have a directed graph in which every vertex has positive indegree. Let $v_1$ be any vertex. For every $i \geq 1$, let $v_{i+1}v_i$ be an arc. In the sequence $v_1, v_2, v_3, \ldots$, consider the first repeated vertex, $v_i = v_j$ where $j > i$. Then $v_j, v_{j-1}, \ldots, v_i, v_j$ is a directed cycle.
Some Interesting/useful Facts (cont.)

Theorem 6.6

A directed graph $D$ has a topological sort if and only if it is a DAG.

Proof.

($\Rightarrow$): Suppose $D$ has a directed cycle $v_1, v_2, \ldots, v_j, v_1$. Then $v_1 < v_2 < \cdots < v_j < v_1$, so a topological ordering does not exist.

($\Leftarrow$): Suppose $D$ is a DAG. Then the algorithm below constructs a topological ordering.

Algorithm: \textit{TopOrdering}($D$)

\begin{align*}
D_1 & \leftarrow D \\
\text{for } i & \leftarrow 1 \text{ to } n \\
\text{do} & \begin{cases} 
\text{let } v_i \text{ be a vertex in } D_i \text{ having indegree 0} \\
\text{construct } D_{i+1} \text{ from } D_i \text{ by deleting } v_i 
\end{cases} \\
\text{return } & (v_1, v_2, \ldots, v_n)
\end{align*}
Developing an Algorithm based on DFS

**Lemma 6.7**

A directed graph is a DAG if and only if a depth-first search encounters no back edges.

**Proof.**

(⇒): Any back edge creates a directed cycle.

(⇐): Suppose $C = v_1, v_1, \ldots, v_\ell$ is a directed cycle. WLOG suppose that $v_1$ is the vertex in $C$ having the lowest discovery time. Consider the arc $v_\ell v_1$. We will prove that $v_\ell v_1$ is a back edge. First, since $d[v_\ell] > d[v_1]$, this arc must be a cross edge or a back edge (see slide #142). Suppose $v_\ell v_1$ is a cross edge. Then $v_1$ is black and $v_\ell$ is grey when the arc $v_\ell v_1$ is processed. But $v_1$ is not coloured black until all vertices reachable from $v_1$ are black. This is a contradiction, and hence $v_\ell v_1$ is a back edge. 

D.R. Stinson (SCS)
Lemma 6.8

Suppose $D$ is a DAG. Then $f[v] < f[u]$ for every arc $uv$.

Proof.

Look at the classification on slide # 203. In a DAG, there are no back edges. For any other type of arc $uv$, it holds that $f[v] < f[u]$.

Therefore, if $D$ is a DAG and we order the vertices in reverse order of finishing time, then we get a topological ordering.
Topological Ordering via Depth-first Search

Algorithm: $DFS(G)$

1. **InitializeStack($S$)**
2. $DAG \leftarrow true$
3. for each $v \in V(G)$
   - do $\{$
     - $colour[v] \leftarrow \text{white}$
     - $\pi[v] \leftarrow \emptyset$
   - $time \leftarrow 0$
4. for each $v \in V(G)$
   - do $\{$
     - if $colour[v] = \text{white}$
       - then $DFSvisit(v)$
   - $if DAg \ then \ return \ (S) \ else \ return \ (DAG)$
Topological Ordering via Depth-first Search (cont.)

Algorithm: $DFSvisit(v)$

$\text{colour}[v] \leftarrow \text{gray}$

$\text{time} \leftarrow \text{time} + 1$

$d[v] \leftarrow \text{time}$

\textbf{comment:} $d[v]$ is the discovery time for vertex $v$

\textbf{for each} $w \in \text{Adj}[v]$

\begin{align*}
\text{if } \text{colour}[w] = \text{white} & \text{ do} \\
& \text{then } \begin{cases} \\
\pi[w] \leftarrow v \\
DFSvisit(w) \\
\text{if } \text{colour}[w] = \text{gray} \text{ then } \text{DAG} \leftarrow \text{false} \\
\end{cases}
\end{align*}

$\text{colour}[v] \leftarrow \text{black}$

$\text{Push}(S, v)$

$\text{time} \leftarrow \text{time} + 1$

$f[v] \leftarrow \text{time}$

\textbf{comment:} $f[v]$ is the finishing time for vertex $v$
Example

We consider the graph from slide # 204.

It has the the following adjacency lists:

\[ \text{Adj}_1 : 6 \]
\[ \text{Adj}_2 : 1 \rightarrow 4 \rightarrow 5 \]
\[ \text{Adj}_3 : 2 \]
\[ \text{Adj}_4 : 1 \]
\[ \text{Adj}_5 : 4 \rightarrow 6 \]
\[ \text{Adj}_6 : \]

The initial calls are \( \text{DFSvisit}(1) \), \( \text{DFSvisit}(2) \) and \( \text{DFSvisit}(3) \).

The discovery/finish times are as follows:

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d[v] )</th>
<th>( f[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>( v )</td>
<td>( d[v] )</td>
<td>( f[v] )</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

The topological ordering is 3, 2, 5, 4, 1, 6 (reverse order of finishing time).
Connectivity and Strong Connectivity

An undirected graph is **connected** if there is a path between any two vertices.

How do we determine if a graph is connected using **DFS**?

In a directed graph, a vertex \( v \) is **reachable** from a vertex \( w \) if there is a directed path from \( v \) to \( w \).

A directed graph \( G \) is **strongly connected** if any vertex is reachable from any other vertex.

Prove that a directed graph \( G \) is strongly connected if all vertices are reachable from \( s \) and \( s \) is reachable from all vertices, for an arbitrary vertex \( s \).
Testing a Directed Graph to see if it is Strongly Connected

1. Pick any vertex $s$ in the directed graph $G$.
2. Run $DFSvisit(s)$ on $G$.
3. If there exists a white vertex, then QUIT ($G$ is not strongly connected).
4. Otherwise, reverse the direction of all edges in $G$ to construct another digraph $G_1$.
5. Run $DFSvisit(s)$ on $G_1$.
6. $G$ is strongly connected if and only if there is no white vertex in $G_1$.

What is the complexity of this algorithm?
(Strongly) Connected Components

For two vertices $x$ and $y$ of $G$, define $x \sim y$ if $x = y$; or if $x \neq y$ and there exist directed paths from $x$ to $y$ and from $y$ to $x$.

The relation $\sim$ is an equivalence relation.

The strongly connected components of $G$ are the equivalence classes of vertices defined by the relation $\sim$.

A strongly connected component of a digraph $G$ is a maximal strongly connected subgraph of $G$.

For undirected graphs, the definition is similar, except we define $x \sim y$ if $x = y$; or if $x \neq y$ and there exists a path joining $x$ and $y$.

Exercise: How do we determine the connected components of an undirected graph?
Strongly Connected Components of a Digraph $G$

The following directed graph has strongly connected components as indicated.

The **component graph** of a directed graph $G$ is a directed graph whose vertices are the strongly connected components of $G$. There is an arc from $C_i$ to $C_j$ if and only if there is an arc in $G$ from some vertex of $C_i$ to some vertex of $C_j$.

**Exercise:** Prove that the component graph of $G$ is a DAG.
Strongly Connected Components of a Digraph $G$ (cont.)

For a strongly connected component $C$, define $f[C] = \max\{f[v] : v \in C\}$ and $d[C] = \min\{d[v] : v \in C\}$.

**Lemma 6.9**

If $C_i$, $C_j$ are strongly connected components, and there is an arc from $C_i$ to $C_j$ in the component graph, then $f[C_i] > f[C_j]$.

**Proof.**

Suppose $d(C_i) < d(C_j)$. Let $u \in C_i$ be the first discovered vertex. All vertices in $C_i \cup C_j$ are reachable from $u$, so they are descendants of $u$ in the DFS tree. Hence $f(v) < f(u)$ for all $v \in C_i \cup C_j$, $v \neq u$. Therefore $f(C_i) > f(C_j)$.

Suppose $d(C_i) > d(C_j)$. In this case, no vertices in $C_i$ are reachable from $C_j$, so $f(C_j) < d(C_i) < f(C_i)$.
Sharir’s Algorithm to Find the Strongly Connected Components

1. Perform a depth-first search of $G$, recording the finishing times $f[v]$ for all vertices $v$.
2. Construct a directed graph $H$ from $G$ by reversing the direction of all edges in $G$.
3. Perform a depth-first search of $H$, considering the vertices in decreasing order of the values $f[v]$ computed in step 1.
4. The strongly connected components of $G$ are the trees in the depth-first forest constructed in step 3.
### Example

Here are the discovery and finish times for each vertex:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>24</td>
<td>7</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>23</td>
<td>8</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>20</td>
<td>9</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>22</td>
<td>10</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>21</td>
<td>11</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>17</td>
<td>12</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
Depth-first Search of $H$

Assume that $f[v_1] > f[v_2] > \cdots > f[v_n]$.

Algorithm: \textit{DFS}(H)

\begin{algorithmic}
\For{$j \leftarrow 1 \text{ to } n$}
\State $\text{colour}[v_i] \leftarrow \text{white}$
\State $\text{scc} \leftarrow 0$
\EndFor
\For{$j \leftarrow 1 \text{ to } n$}
\If{$\text{colour}[v_i] = \text{white}$}
\State $\text{scc} \leftarrow \text{scc} + 1$
\State $\text{DFSvisit}(H, v_i, \text{scc})$
\EndIf
\EndFor
\Return $(\text{comp})$
\end{algorithmic}

\textbf{comment:} $\text{comp}[v]$ is the strongly connected component containing $v$
DFSvisit for $H$

**Algorithm:** $\text{DFSvisit}(H, v, \text{scc})$

- $\text{colour}[v] \leftarrow \text{gray}$
- $\text{comp}[v] \leftarrow \text{scc}$

for each $w \in \text{Adj}[v]$

- do { if $\text{colour}[w] = \text{white}$ then $\text{DFSvisit}(H, w, \text{scc})$
- $\text{colour}[v] \leftarrow \text{black}$
Example

Here is the previous directed graph with edge directions reversed:

Recall that we process the vertices in the order

\[1, 2, 4, 5, 3, 6, 7, 8, 9, 10, 11, 12.\]

\(\text{DFSvisit}(1)\) explores the s.c.c having vertices \(\{1, 2, 3, 4, 5\}\).

\(\text{DFSvisit}(6)\) explores the s.c.c having vertices \(\{6\}\).

\(\text{DFSvisit}(7)\) explores the s.c.c having vertices \(\{7, 8, 9\}\).

\(\text{DFSvisit}(10)\) explores the s.c.c having vertices \(\{10, 11, 12\}\).
Proof of Correctness of Sharir’s Algorithm

First, note that $G$ and $H$ have the same strongly connected components.

Let $u = v_{i_1}$ be the first vertex visited in step 3. Let $C$ be the s.c.c. containing $u$ and let $C'$ be any other s.c.c.

$f(C) > f(C')$, so there is no edge from $C'$ to $C$ in $G$ (by the Lemma). Therefore there is no edge from $C$ to $C'$ in $H$.

Hence no vertex in $C'$ is reachable from $u$ in $H$.

Therefore, $\text{DFSvisit}(u)$ explores the vertices in $C$ (and only those vertices); this forms one DFS tree in $H$.

Next, $\text{DFSvisit}(v_{i_2})$ explores the vertices in the s.c.c. containing $v_{i_2}$, etc.

Every time we make an initial call to $\text{DFSvisit}$, we are exploring a new s.c.c.

We increment $scc$, which is used to label the various s.c.c.

$\text{comp}[v]$ denotes the label of the s.c.c. containing $v$. 
Minimum Spanning Trees

A spanning tree in a connected, undirected graph $G = (V, E)$ is a subgraph $T$ that is a tree which contains every vertex of $V$.

$T$ is a spanning tree of $G$ if and only if $T$ is an acyclic subgraph of $G$ that has $n - 1$ edges (where $n = |V|$).

Problem 6.10

Minimum Spanning Tree

Instance: A connected, undirected graph $G = (V, E)$ and a weight function $w : E \rightarrow \mathbb{R}$.

Find: A spanning tree $T$ of $G$ such that

$$\sum_{e \in T} w(e)$$

is minimized (this is called a minimum spanning tree, or MST).
Kruskal’s Algorithm

Assume that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$, where $m = |E|$.

Algorithm: \textit{Kruskal}(G, w)

\begin{align*}
A &\leftarrow \emptyset \\
\text{for } j &\leftarrow 1 \text{ to } m \\
\quad \text{do } &\begin{cases}
\text{if } A \cup \{e_j\} \text{ does not contain a cycle} \\
\quad \quad \text{then } A \leftarrow A \cup \{e_j\}
\end{cases} \\
\text{return } (A)
\end{align*}
An Example of Kruskal’s Algorithm
## Implementation Details for Kruskal’s Algorithm

We use a **union-find** data structure to determine if an edge $uv$ has vertices in two different trees.

Every tree $T$ will contain a **leader vertex**.

To find the leader vertex from a vertex $v$, we use an auxiliary array $L$.

From $v$, follow a directed path $v \rightarrow L[v] \rightarrow L[L[v]] \cdots$ until we reach a vertex $w$ with $L[w] = w$; then $w = \text{find}(v)$ is the leader vertex for the tree containing $v$.

Two vertices $u$ and $v$ are in the same tree if and only if $\text{find}(u) = \text{find}(v)$.

Initially, there are $n$ one-vertex trees and $L[v] = v$ for all $v$.

When we use an edge $uv$ to merge two trees, we perform the following **union** operation:

1. $u' \leftarrow \text{find}(u)$
2. $v' \leftarrow \text{find}(v)$
3. $L[u'] \leftarrow v'$.
Implementation Details and Complexity

Suppose we also keep track of the depth of each tree. In step 3, we always take $u'$ to be the leader of the tree having smaller depth.

If we merge two trees of depth $d$, we get a tree of depth $d + 1$. If we merge a tree of depth $d$ and one of depth $< d$, we have a tree of depth $d$.

Then union and find each run in $O(\log n)$ time (this is because a tree of depth $d$ has at least $2^d$ vertices, a fact that can be proven by induction on $d$).

This leads to an algorithm for MST having complexity $O(m \log n)$ (the pre-sort has complexity $O(m \log m)$, and the iterative part of the algorithm has complexity $O(m \log n)$).
Proof of Correctness

Let’s assume that all edge weights are distinct. Let $A$ be the spanning tree constructed by *Kruskal’s algorithm* and let $A'$ be an arbitrary MST.

Suppose the edges in $A$ are named $f_1, f_2, \ldots, f_{n-1}$, where $w(f_1) < w(f_2) \cdots < w(f_{n-1})$. Suppose $A \neq A'$ and let $f_j$ be the first edge in $A \setminus A'$.

$A' \cup \{f_j\}$ contains a unique cycle, say $C$. Let $e'$ be the first (i.e, lowest weight) edge of $C$ that is not in $A$ (such an edge exists because $C \not\subseteq A$). Define $A'' = A' \cup \{f_j\} \setminus \{e'\}$. Then $w(A'') = w(A') + w(f_j) - w(e')$.

Since $A'$ is an MST, we must have $w(A'') \geq w(A')$. Therefore, $w(f_j) \geq w(e')$. The edge weights are all distinct, so $w(f_j) > w(e')$.

What happened when *Kruskal’s algorithm* considered the edge $e'$? This occurred before it considered the edge $f_j$, because $w(f_j) > w(e')$. Since *Kruskal’s algorithm* rejected the edge $e'$, the edges $f_1, \ldots, f_{j-1}, e'$ must contain a cycle. However, $A'$ contains all these edges and $A'$ is a tree, so we have a contradiction. Therefore $A = A'$ and $A$ is an MST.
**Prim’s Algorithm (idea)**

We initially choose an arbitrary vertex \( u_0 \) and define \( A = \{ e \} \), where \( e \) is the *minimum weight* edge incident with \( u_0 \).

\( A \) is always a *single tree*, and at each step we select the minimum weight edge that joins a vertex in \( V_A \) to a vertex not in \( V_A \).

**Remark:** \( V_A \) denotes the set of vertices in the tree \( A \).

For a vertex \( v \not\in V_A \), define

\[
N[v] = u, \text{ where } \{u, v\} \text{ is a minimum weight edge such that } u \in V_A \\
W[v] = w(N[v], v).
\]

Assume \( w(u, v) = \infty \) if \( \{u, v\} \not\in E \).
Prim’s Algorithm

Algorithm: \( \text{Prim}(G, w) \)

\[ \begin{align*}
A & \leftarrow \emptyset \\
V_A & \leftarrow \{u_0\}, \text{ where } u_0 \text{ is arbitrary} \\
\text{for all } & \ v \in V \setminus \{u_0\} \\
\text{do} & \left\{ \\
W[v] & \leftarrow w(u_0, v) \\
N[v] & \leftarrow u_0 \\
\right. \\
\text{while } & \ |A| < n - 1 \\
\text{do} & \left\{ \\
\text{choose } & \ v \in V \setminus V_A \text{ such that } W[v] \text{ is minimized} \\
V_A & \leftarrow V_A \cup \{v\} \\
u & \leftarrow N[v] \\
A & \leftarrow A \cup \{uv\} \\
\text{for all } & \ v' \in V \setminus V_A \\
\text{do} & \left\{ \\
\text{if } & \ w(v, v') < W[v'] \\
\text{do} & \left\{ \\
W[v'] & \leftarrow w(v, v') \\
N[v'] & \leftarrow v \\
\text{then } & \right. \\
\right. \\
\right. \\
\text{return } & (A)
\end{align*} \]
Implementation and Complexity

Simple implementation:

- There are \( n - 1 \) iterations of the “while” loop.
- Finding \( v \) takes time \( O(n) \).
- Updating \( W \)-values takes time \( O(n) \).
- The algorithm has complexity \( O(n^2) \).

Priority queue implementation: Use a priority queue (implemented as a min-heap) to store the \( W \)-values.

Fibonacci heap implementation: Use a Fibonacci heap to store the \( W \)-values.
A General Algorithm: Definitions

Let $G = (V, E)$ be a graph. A **cut** is a partition of $V$ into two non-empty (disjoint) sets, i.e., a pair $(S, V \setminus S)$, where $S \subseteq V$ and $1 \leq |S| \leq n - 1$.

Let $(S, V \setminus S)$ be a cut in a graph $G = (V, E)$. An edge $e \in E$ is a **crossing edge** with respect to the cut $(S, V \setminus S)$ if $e$ has one endpoint in $S$ and one endpoint in $V \setminus S$.

Let $A \subseteq E$. A cut $(S, V \setminus S)$ **respects** the set of edges $A$ provided that no edge in $A$ is a crossing edge.
A General Greedy Algorithm to Find an MST

Algorithm: \textit{GreedyMST}(G, w)

\begin{align*}
A & \leftarrow \emptyset \\
\text{while } & \quad |A| < n - 1 \\
\quad \text{do } & \quad \begin{cases} \\
\quad \text{let } (S, V \setminus S) \text{ be a cut that respects } A \\
\quad \text{let } e \text{ be a minimum weight crossing edge} \\
\quad A & \leftarrow A \cup \{e\} \\
\end{cases} \\
\text{return } & \quad (A)
\end{align*}
Correctness Proof of the General Greedy Algorithm to Find an MST

We prove that the spanning tree $A$ constructed by the general greedy algorithm is a MST, assuming all edge weights are distinct.

Let $e_1, \ldots, e_{n-1}$ be the edges in $A$ in order that they are added to $A$. We prove by induction on $j$ that $\{e_1, \ldots, e_j\}$ is contained in an MST. $j = 0$ is a trivial base case.

**Induction assumption**: suppose that $A_{j-1} = \{e_1, \ldots, e_{j-1}\} \subseteq T$, where $T$ is an MST, and consider $e_j$. If $e_j \in T$, we’re done, so assume $e_j \notin T$.

There is a cut $(S, V \setminus S)$ respecting $A_{j-1}$ for which $e_j$ is the minimum crossing edge. $T \cup \{e_j\}$ contains a unique cycle $C$. There is an edge $e' \neq e_j$ such that $e' \in C$ and $e'$ is a crossing edge for the cut $(S, V \setminus S)$.

Let $T' = T \cup \{e_j\} \setminus \{e'\}$. $w(T') = w(T) + w(e_j) - w(e')$. $T'$ is a spanning tree and $T$ is an MST, so $w(T') \geq w(T)$; hence $w(e_j) \geq w(e')$. But $e_j$ is the minimum weight crossing edge, so $w(e_j) < w(e')$; contradiction.
Single Source Shortest Paths

Problem 6.11

Single Source Shortest Paths

Instance: A directed graph $G = (V, E)$, a non-negative weight function $w : E \rightarrow \mathbb{R}^+ \cup \{0\}$, and a source vertex $u_0 \in V$.

Find: For every vertex $v \in V$, a directed path $P$ from $u_0$ to $v$ such that

$$w(P) = \sum_{e \in P} w(e)$$

is minimized.

The term shortest path really means minimum weight path.

We are asked to find $n$ different shortest paths, one for each vertex $v \in V$. 
Dijkstra’s Algorithm (Main Ideas)

Dijkstra’s algorithm requires that the graph have no edge weights < 0; it works for directed or undirected graphs.

$S$ is a subset of vertices such that the shortest paths from $u_0$ to all vertices in $S$ are known; initially, $S = \{u_0\}$.

For all vertices $v \in S$, $D[v]$ is the weight of the shortest path $P_v$ from $u_0$ to $v$, and all vertices on $P_v$ are in the set $S$.

For all vertices $v \not\in S$, $D[v]$ is the weight of the shortest path $P_v$ from $u_0$ to $v$ in which all interior vertices are in $S$.

For $v \neq u_0$, $\pi[v]$ is the predecessor of $v$ on the path $P_v$.

At each stage of the algorithm, we choose $v \in V \setminus S$ so that $D[v]$ is minimized, and then we add $v$ to $S$ (see the Lemma on the next slide). Then the arrays $D$ and $\pi$ are updated appropriately.
Dijkstra’s Algorithm (Main Ideas, cont.)

Lemma 6.12

Suppose \( v \) has the smallest \( D \)-value of any vertex not in \( S \). Then \( D[v] \) equals the weight of the shortest \((u_0, v)\)-path.

Proof.

Suppose there is a \((u_0, v)\)-path \( P' \) with weight less than \( D[v] \). Let \( v' \) be the first vertex of \( P' \) not in \( S \). Observe that \( v' \neq v \). Decompose \( P' \) into two paths: a \((u_0, v')\)-path \( P_1 \) and a \((v', v)\)-path \( P_2 \). We have

\[
\begin{align*}
    w(P') &= w(P_1) + w(P_2) \\
    &\geq D[v'] + w(P_2) \\
    &\geq D[v] \quad (w(P_2) \geq 0 \text{ because all edge weights are } \geq 0).
\end{align*}
\]

This is a contradiction because we assumed \( w(P') < w(P) \). \( \square \)
Updating $D$ and $\pi$

Lemma 6.12 says we can add $v$ to $S$.

To update a value $D[v']$ for $v' \not\in S$, we consider the new “candidate” path consisting of the shortest $(u_0, v)$-path together with the edge $vv'$.

If this is shorter than the current best $(u_0, v')$-path, then update.

Updating is only required for vertices $v' \in Adj[v]$
Dijkstra’s Algorithm

Algorithm: Dijkstra\((G, w, u_0)\)

\[ S \leftarrow \{u_0\} \]
\[ D[u_0] \leftarrow 0 \]

for all \( v \in V \setminus \{u_0\} \)
\[ \begin{cases} D[v] \leftarrow w(u_0, v) \
\pi[v] \leftarrow u_0 \end{cases} \]

while \(|S| < n\)
\[ \begin{cases} \text{choose } v \in V \setminus S \text{ such that } D[v] \text{ is minimized} \\
S \leftarrow S \cup \{v\} \end{cases} \]

for all \( v' \in V \setminus S \)
\[ \begin{cases} \text{if } D[v] + w(v, v') < D[v'] \\
\begin{cases} D[v'] \leftarrow D[v] + w(v, v') \\
\pi[v'] \leftarrow v \end{cases} \end{cases} \]

return \((D, \pi)\)
Finding the Shortest Paths

Algorithm: \textit{FindPath}(u_0, \pi, v)

\begin{verbatim}
path ← v
u ← v

\textbf{while} u \neq u_0
\textbf{do } \\
\quad u ← \pi[u]
\quad path ← u \parallel path

\textbf{return} (path)
\end{verbatim}
Dijkstra and Prim

*Dijkstra’s algorithm* (for shortest paths) is very similar to *Prim’s algorithm* (for MST).

The only difference is the updating step (also known as relaxation)

<table>
<thead>
<tr>
<th>Prim</th>
<th>Dijkstra</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( w(v, v') &lt; W[v'] ) then ( W[v'] \leftarrow w(v, v') )</td>
<td>if ( D[v] + w(v, v') &lt; D[v'] ) then ( D[v'] \leftarrow D[v] + w(v, v') )</td>
</tr>
</tbody>
</table>
Shortest Paths and Negative Weight Cycles

Subsequent algorithms we will be studying will solve shortest path problems as long as there are no cycles having negative weight.

If there is a negative weight cycle, then there is no shortest path (why?).

There is still a shortest simple path, but there are apparently no known efficient algorithms to find the shortest simple paths in graphs containing negative weight cycles.

If there are no negative weight cycles, we can assume WLOG that shortest paths are simple paths (any path can be replaced by a simple path having the same weight).

Negative weight edges in an undirected graph are not allowed, as they would give rise to a negative weight cycle (consisting of two edges) in the associated directed graph.
**Shortest Paths in a DAG**

If $G$ is a DAG, we perform a topological ordering of the vertices. Suppose the resulting ordering is $v_1, \ldots, v_n$. Then we find all the shortest paths in $G$ with source $v_1$.

**Algorithm:** *DAG Shortest paths*($G, w, v_1$)

```plaintext
for $j \leftarrow 1$ to $n$
    do $
        \{ D[v_1] \leftarrow \infty \\
        \quad \pi[v_j] \leftarrow \text{undefined}
    \}$

$D[v_1] \leftarrow 0$

for $j \leftarrow 1$ to $n - 1$
    do
        for all $v' \in \text{Adj}[v_j]$
            do
                if $D[v_j] + w(v_j, v') < D[v']$
                    then
                        $D[v'] \leftarrow D[v_j] + w(v_j, v')$
                        $
                        \pi[v'] \leftarrow v_j$
```

return $(D, \pi)$
Example

A directed graph, where all edges are directed from left to right:

\[
\begin{array}{c|ccccccc}
\hline
0 & 0 & \infty & \infty & \infty & \infty & \infty \\
1 & 0 & 5 & 3 & \infty & \infty & \infty \\
2 & 0 & 5 & 1 & 11 & \infty & \infty \\
3 & 0 & 5 & 1 & 8 & 5 & 3 \\
4 & 0 & 5 & 1 & 8 & 5 & 3 \\
5 & 0 & 5 & 1 & 8 & 5 & 2 \\
\end{array}
\]
Bellman-Ford

The Bellman-Ford algorithm solves the single source shortest path problem in any directed graph without negative weight cycles.

The algorithm is very simple to describe:

Repeat $n - 1$ times: relax every edge in the graph (where relax is the updating step in Dijkstra’s algorithm).

Dijkstra’s algorithm has complexity $O(m \log n)$ (using priority queues) whereas Bellman-Ford has complexity $O(mn)$.

However, Dijkstra requires that the graph contain no negative weight edges.
All-Pairs Shortest Paths

Problem 6.13

Instance: A directed graph $G = (V, E)$, and a weight matrix $W$, where $W[i, j]$ denotes the weight of edge $ij$, for all $i, j \in V, i \neq j$.

Find: For all pairs of vertices $u, v \in V, u \neq v$, a directed path $P$ from $u$ to $v$ such that

$$w(P) = \sum_{ij \in P} W[i, j]$$

is minimized.

We allow edges to have negative weights, but we assume there are no negative-weight directed cycles in $G$. 
A Dynamic Programming Approach

Suppose we successively consider paths of length 1, 2, \ldots, n - 1. Let $L_m[i, j]$ denote the minimum-weight $(i, j)$-path having at most $m$ edges. We want to compute $L_{n-1}$. We can use a dynamic programming approach to do this.

**Initialization:** $L_1 = W$.

**Optimal structure:** Let $k$ be the predecessor of $j$ on the minimum-weight $(i, j)$-path $P$ having at most $m$ edges. Then the portion of $P$ from $i$ to $k$, say $P'$, is a minimum-weight $(i, k)$-path having at most $m - 1$ edges. This is the optimal structure required in order to find a dynamic programming algorithm.

**Updating:** For $m \geq 2$, 

$$ L_m[i, j] = \min\{L_{m-1}[i, k] + L_1[k, j] : 1 \leq k \leq n\}. $$

(Note that $k = i, j$ does not cause any problems.)

**Complexity:** $O(n^4)$. 
First Solution

Algorithm: \textit{FairlySlowAllPairsShortestPath}(W)

\begin{align*}
L_1 & \leftarrow W \\
\text{for } m & \leftarrow 2 \text{ to } n - 1 \\
\quad \text{for } i & \leftarrow 1 \text{ to } n \\
\quad \quad \text{for } j & \leftarrow 1 \text{ to } n \\
\quad \quad \quad \text{do } & \left\{ \\
\quad \quad \quad \quad \text{for } k & \leftarrow 1 \text{ to } n \\
\quad \quad \quad \quad \quad \text{do } \ell & \leftarrow \infty \\
\quad \quad \quad \quad \quad \quad \text{do } \ell \leftarrow \min\{\ell, L_{m-1}[i, k] + W[k, j]\} \\
\quad \quad \quad \quad \quad L_m[i, j] & \leftarrow \ell
\end{align*}

return \( (L_{n-1}) \)
Second Solution: Successive Doubling

**Algorithm:** \( \text{FasterAllPairsShortestPath}(W) \)

\[
\begin{align*}
L_1 & \leftarrow W \\
 m & \leftarrow 1 \\
\text{while } m < n - 1 & \quad \text{do} \\
\quad \text{for } i \leftarrow 1 \text{ to } n & \quad \text{do} \\
\quad \quad \text{for } j \leftarrow 1 \text{ to } n & \quad \text{do} \\
\quad \quad \quad \ell & \leftarrow \infty \\
\quad \quad \quad \text{for } k \leftarrow 1 \text{ to } n & \quad \text{do} \\
\quad \quad \quad \quad \ell & \leftarrow \min\{\ell, L_m[i, k] + L_m[k, j]\} \\
\quad \quad \quad L_{2m}[i, j] & \leftarrow \ell \\
\quad m & \leftarrow 2m \\
\text{return } (L_m)
\end{align*}
\]
Third Solution: Floyd-Warshall

Algorithm: $\text{FloydWarshall}(W)$

$$D_0 \leftarrow W$$

for $m \leftarrow 1$ to $n$

\[
\begin{array}{l}
\text{for } i \leftarrow 1 \text{ to } n \\
\quad \text{do } \\
\quad \quad \text{for } j \leftarrow 1 \text{ to } n \text{ do} \\
\quad \quad \quad D_m[i, j] \leftarrow \min\{D_{m-1}[i, j], D_{m-1}[i, m] + D_{m-1}[m, j]\}
\end{array}
\]

return $(D_n)$
Correctness of Floyd-Warshall

Let $P$ be the min-weight $(i, j)$ path having all of its interior vertices in the set $\{1, \ldots, m - 1\}$. Clearly $w(P) = D_{m-1}[i, j]$.

Let $P'$ be the min-weight $(i, j)$ path having $m$ as an interior vertex and having of its all interior vertices in the set $\{1, \ldots, m\}$. If $P'$ is not a simple path, it can be replaced by a simple path $P'''$ of the same weight, but this path may or may not contain $m$ as an interior vertex.

If $P'''$ has all of its interior vertices in the set $\{1, \ldots, m - 1\}$, then $w(P') = w(P''') \geq D_{m-1}[i, j]$. On the other hand, if $P'''$ contains $m$ as an interior vertex, then we can decompose $P'''$ into two disjoint simple paths, an $(i, m)$ path $P_1$ and an $(m, j)$ path $P_2$.

If $D_{m-1}[i, m] + D_{m-1}[m, j] < D_{m-1}[i, j]$ then $P_1$ and $P_2$ are disjoint since there are no negative weight cycles. Hence,

$$w(P') = w(P''') = w(P_1) + w(P_2) = D_{m-1}[i, m] + D_{m-1}[m, j].$$
Example of Floyd-Warshall

Graph:

- Vertices: a, b, c, d
- Edges and Weights:
  - a to b: 3
  - b to c: 4
  - c to d: 12
  - d to a: -1
  - a to c: 5
  - b to d: 2

The Floyd-Warshall algorithm can be used to find the shortest paths between all pairs of vertices in a weighted graph. The example above illustrates how the algorithm works by updating the shortest path matrix iteratively.