Chapter 6 Graphs

- The Graph Abstract Data Type
- Elementary Graph Operations
- Minimum Cost Spanning Trees
- Shortest Paths and Transitive Closure
- Activity Networks

Introduction

- **Königsberg problem**: A river Pregel flows around the island Keniphof and then divides into two. Four land areas A, B, C, and D have this river on their borders. The four lands are connected by 7 bridges, a - g.

- Is there a possible way to **traverse every bridge exactly once**.
**Introduction**

Euler used graph theory to solve the seven bridges of Königsberg problem. Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each vertex is even. This walk is called Eulerian or **Euler tour**.

- No Eulerian walk of the Königsberg bridge problem since all four vertices are of odd edges.

```
B -> a -> A -> e -> D -> g -> C -> d -> A -> b -> B -> f -> D ???
```

---

**Graph Definition**

- A **graph** \( G = (V, E) \), \( V \) and \( E \) are two sets
  - \( V \): finite non-empty set of **vertices**
  - \( E \): set of pairs of vertices, **edges**

**Example:** a graph \( G_1 \)

- \( V(G_1) = \{0, 1, 2, 3\} \), **four vertices**
- \( E(G_1) = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\} \), **six edges**
Undirected and Directed Graphs

**Undirected graph**
- The pair of vertices representing any edge is unordered. Thus, the pairs (u, v) and (v, u) represent the same edge.

**Example:** a graph G₂
- \( V(G₂) = \{0, 1, 2, 3, 4, 5, 6\} \)
- \( E(G₂) = \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)\} \)

G₂ is also a tree that is a special case of graph.

**Directed graph** (or digraph)
- Each edge is represented by an ordered pair \(<u, v>\).

**Example:** a graph G₃
- \( V(G₃) = \{0, 1, 2\} \)
- \( E(G₃) = \{<0, 1>, <1, 0>, <1, 2>\} \)

Examples of Graphlike Structures

- **Graph with self loops**
- **Multigraph**
Complete Graph

- A **complete graph** is a graph that has the maximum number of edges
  - For undirected graph with n vertices, the maximum number of edges is \( n(n-1)/2 \)
  - For directed graph with n vertices, the maximum number of edges is \( n(n-1) \)

**Example: G_1**
- edges is \( 4*(4-1)/2 = 6 \) edges

Adjacent and Incident

- If \((u,v)\) is an edge in an undirected graph,
  - **Adjacent**: \(u\) and \(v\) are adjacent
  - **Incident**: The edge \((u,v)\) is incident on vertices \(u\) and \(v\)
- If \(<u,v>\) is an edge in a directed graph
  - **Adjacent**: \(u\) is adjacent to \(v\), and \(v\) is adjacent from \(u\)
  - **Incident**: The edge \(<u,v>\) is incident on \(u\) and \(v\)
Subgraph

A subgraph of $G$ is a graph $G'$ such that
- $V(G') \subseteq V(G)$,
- $E(G') \subseteq E(G)$

Some of the subgraphs of $G_1$

Some of the subgraphs of $G_3$

Path

Path from $u$ to $v$ in $G$
- a sequence of vertices $u, i_1, i_2, \ldots, i_k, v$
- If $G$ is undirected: $(u, i_1), (i_1, i_2), \ldots, (i_k, v) \in E(G)$
- If $G$ is directed: $<u, i_1>, <i_1, i_2>, \ldots, <i_k, v> \in E(G)$

Length
- The length of a path is the number of edges on it.

Example: Length of $0, 1, 3, 2$ is 3

Simple path
- is a path in which all vertices except possibly the first and last are distinct.

Example: $0, 1, 3, 2$ is simple path
- $0, 1, 3, 1$ is path but not simple
Cycle

- **Cycle** is a simple path, first and last vertices are same.
- **Example:** 0, 1, 2, 0 is a cycle

- **Acyclic graph:** no cycle is in graph

Connected Components

- Two vertices u and v are **connected** if they are in an undirected graph G, there is a path in G from u to v.
- A graph G is **connected**, if any vertex pair (u, v) is connected.
- **Connected component** is a maximal connected subgraph.
- **Tree** is a connected acyclic graph.
- **Example:**
  A graph G with two connected components H₂ and H₃

```
0 1 2 3
H₁

4 5 6 7
H₂
```
**Strongly Connected Components**

- \( u, v \) are strongly connected if they are in a directed graph (digraph) \( G \), \( \exists \) a path in \( G \) from \( u \) to \( v \).
- A directed graph \( G \) is strongly connected, if any vertex pair \( (u,v) \) is connected

**Example 1:** \( G_3 \) is a strong connected graph

**Example 2:** A graph \( G \) with two strongly connected components \( H_1 \) and \( H_2 \)

**Strongly Connected Component** is a maximal strongly connected subgraph

**Degree**

- Degree of a vertex is the number of edges incident to that vertex

- Degree in directed graph
  - Indegree
  - Outdegree
- Summation of all vertices’ degrees are \( 2|E| \)
The Graph Abstract Data Type

- **ADT for Graph** is
  - objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices
  - functions: for all \( graph \in \text{Graph} \), \( v, v_1 \), and \( v_2 \in \text{Vertices} \)
    - \( \text{Graph Create}() ::= \text{return an empty graph} \)
    - \( \text{Graph InsertVertex}(graph, v) ::= \text{return a graph with } v \text{ inserted.} \) \( v \text{ has no incident edge.} \)
    - \( \text{Graph InsertEdge}(graph, v_1, v_2) ::= \text{return a graph with new edge between } v_1 \text{ and } v_2 \)
    - \( \text{Graph DeleteVertex}(graph, v) ::= \text{return a graph in which } v \text{ and all edges incident to it are removed} \)
    - \( \text{Graph DeleteEdge}(graph, v_1, v_2) ::= \text{return a graph in which the edge } (v_1, v_2) \text{ is removed} \)
    - \( \text{Boolean IsEmpty}(graph) ::= \text{if } (graph==\text{empty graph}) \text{ return TRUE} \)
      - else return FALSE
    - \( \text{List Adjacent}(graph, v) ::= \text{return a list of all vertices that are adjacent to } v \)

Graph Representations

- Adjacency Matrix
- Adjacency Lists
- Adjacency Multilists
- Weighted Edge
Let $G = (V, E)$ with $n$ vertices, $n \geq 1$, the adjacency matrix of $G$ is a $2$-dimensional $n \times n$ matrix, $A$

- $A(i, j) = 1$ iff $(v_i, v_j) \in E(G)$
- $A(i, j) = 0$ otherwise.

The adjacency matrix for an undirected graph is symmetric

The adjacency matrix for a digraph need not be symmetric

Examples of Adjacency Matrix

$$
\begin{bmatrix}
0 & 1 & 2 & 3 \\
0 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 \\
2 & 1 & 0 & 1 \\
3 & 1 & 1 & 1 \\
\end{bmatrix}
$$

$$
\begin{bmatrix}
0 & 1 & 0 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
0 & 0 & 0 \\
\end{bmatrix}
$$

$$
\begin{bmatrix}
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\end{bmatrix}
$$
Merits of Adjacency Matrix

- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is
  \[ \sum_{j=0}^{n-1} \text{adj} \_ \text{mat}[i][j] \]
- For a digraph, the row sum is the out\_degree, while the column sum is the in\_degree
  \[ \text{ind} (vi) = \sum_{j=0}^{n-1} A[j,i] \quad \text{outd} (vi) = \sum_{j=0}^{n-1} A[i,j] \]

Adjacency Lists

- Replace n rows of the adjacency matrix with n linked list. Each row in adjacency matrix is represented as an adjacency list.
- Data Structures for Adjacency Lists
  ```c
  #define MAX_VERTICES 50
  typedef struct node *nodePointer;
  typedef struct node {
    int vertex;
    struct node *link;
  };
  nodePointer graph[MAX_VERTICES];
  int n=0; /* vertices currently in use */
  ```
Examples of Adjacency Lists

G₁

0 -> 1 -> 2 -> 3
1 -> 0 -> 2 -> 3
2 -> 0 -> 1 -> 3
3 -> 0 -> 1 -> 2

G₃

0 -> 1
1 -> 0 -> 2
2

Examples (Cont’d)

G₄

An undirected graph with n vertices and e edges

==⇒ n head nodes and 2e list nodes
Interesting Operations

- **degree of a vertex** in an undirected graph
  - number of nodes in adjacency list
- **Number of edges** in a graph
  - determined in $O(n+e)$
- **out-degree** of a vertex in a directed graph
  - number of nodes in its adjacency list
- **in-degree** of a vertex in a directed graph
  - traverse the whole data structure

Compact Representation

- **G4**: node[0] … node[n-1]: starting point for vertices
  - node[n]: n+2e+1
  - node[n+1] … node[n+2e]: head node of edge

```
```

C-C Tsai
Inverse Adjacency List of $G_3$

Alternative Adjacency List of $G_3$
Alternative Adjacency List of $G_1$

An edge in an undirected graph is represented by two nodes in adjacency list representation.

Adjacency Multilists
- lists in which nodes may be shared among several lists.
  (an edge is shared by two different paths)

```c
typedef struct edge *edgePointer;
typedef struct edge {
    short int marked;
    int vertex1, vertex2;
    edgePointer path1, path2;
} edge;

typedef struct edge
```
Example for Adjacency Multlists

Lists: vertex 0: N0->N1->N2, vertex 1: N0->N3->N4
vertex 2: N1->N3->N5, vertex 3: N2->N4->N5

G1

Vertex 0: N0->N1->N2
Vertex 1: N0->N3->N4
Vertex 2: N1->N3->N5
Vertex 3: N2->N4->N5

Weighted Edges

- In many applications, the edges of a graph are assigned weights
- These weights may represent the distance from one vertex to another
- A graph with weighted edges is called a network
Elementary Graph Operations

- Traversal: given $G = (V,E)$ and vertex $v$, find or visit all $w \in V$, such that $w$ connects $v$
  - Depth First Search (DFS): preorder tree traversal
  - Breadth First Search (BFS): level order tree traversal

- Applications
  - Connected component
  - Spanning trees
  - Biconnected component

Depth First Search

- Begin the search by visiting the start vertex $v$
  - If $v$ has an unvisited neighbor, traverse it recursively
  - Otherwise, backtrack

Example:
- Start vertex: 0
- Traverse order: 0, 1, 3, 7, 4, 5, 2, 6
# Depth First Search of a Graph

```c
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
void dfs(int v)
{
    node_pointer w;
    visited[v] = TRUE;
    printf("%5d", v);
    for (w=graph[v]; w; w=w->link)
        if (!visited[w->vertex])
            dfs(w->vertex);
}
```

**Time complexity**
- Adjacency list: $O(e)$
- Adjacency matrix: $O(n^2)$

---

## Breadth First Search

- Begin the search by visiting the start vertex $v$
  - Traverse $v$'s neighbors
  - Traverse $v$'s neighbors' neighbors

**Example:**
- Start vertex: 0
- Traverse order: 0, 1, 2, 3, 4, 5, 6, 7

![Breadth First Search Diagram]
Breadth First Search of a Graph

```c
void bfs(int v)
{
    nodePointer w;
    queuePointer front, rear;
    front = rear = NULL;
    printf("%5d", v);
    visited[v] = TRUE;
    addq(&front, &rear, v);
    while (front)
    {
        v = deleteq(&front);
        for (w=graph[v]; w; w=w->link)
        {
            if (!visited[w->vertex])
            {
                printf("%5d", w->vertex);
                addq(&front, &rear, w->vertex);
                visited[w->vertex] = TRUE;
            }
        }
    }
}
```

```c
typedef struct queue *queuePointer;
typedef struct queue {
    int vertex;
    queuePointer link;
} queuePointer;

void addq(queuePointer *q,
          queuePointer *rear,
          int item);

int deleteq(queuePointer *front);
```

Time complexity:
- Adjacency list: $O(e)$
- Adjacency matrix: $O(n^2)$

Connected Component

- Find all connected component in a graph $G$
  - Select one unvisited vertex $v$, and start DFS (or BFS) on $v$
  - Select one another unvisited vertex $v$, and start DFS (or BFS) on $v$

Example:
- Start on 0
  - 0, 1, 3, 4, 7 are visited
  - These 5 vertices is in the same connected component
- Choose 2
  - 2, 5, 6 are visited

```c
void connected(void)
{
    for (i=0; i<n; i++)
    {
        if (!visited[i])
        {
            dfs(i); printf("\n");
        }
    }
}
```
Other Applications

- Find articulation point (cut vertex) in undirected connected graph
- Find bridge (cut edge) in undirected connected graph
- Find strongly connected component in digraph
- Find biconnected component in connected graph
- Find Euler path in undirected connected graph
- Determine whether a graph G is bipartite graph?
- Determine whether a graph G contain cycle?
- Calculate the radius and diameter of a tree

Spanning Trees

- Spanning tree: any tree consists of only edges in G and includes all vertices in G.

Example:

G

Three possible spanning trees.

- How many spanning trees?
Spanning Trees

- Either DFS or BFS can be used to create a spanning tree
  - When DFS is used, the resulting spanning tree is known as a depth first spanning tree
  - When BFS is used, the resulting spanning tree is known as a breadth first spanning tree
- While adding a nontree edge into any spanning tree, this will create a cycle

DFS VS BFS Spanning Trees

DFS spanning tree

BFS spanning tree
**Biconnected Components**

- A spanning tree is a **minimal subgraph**, $G'$, of $G$ such that $V(G') = V(G)$ and $G'$ is connected.
- Any connected graph with $n$ vertices must have at least $n-1$ edges.
- A **biconnected graph** is a connected graph that has no articulation points.

![Connected Graph](connected_graph.png)

- Connected graph
- One connected graph
- Two connected components

---

**Biconnected Components of a Graph**

- Connected graph
- Biconnected components

![Connected Graph](connected_graph.png)

- Connected graph
- Biconnected components

---

C-C Tsai
Find Biconnected Components

**Depth first spanning tree**

(\(dfn\)-depth first number)

If \(u\) is an ancestor of \(v\) then \(dfn(u) < dfn(v)\).

*The root of a depth first spanning tree is an articulation point iff it has at least two children.*

*Any other vertex \(u\) is an articulation point iff it has at least one child \(w\) such that we cannot reach an ancestor of \(u\) using a path that consists of (1) only \(w\) (2) descendants of \(w\) (3) single back edge.*

**low(u)** = minimum of \(\{\text{dfn}(u), \text{min}\{\text{low}(w) \mid w \text{ is a child of } u\}, \text{min}\{\text{dfn}(w) \mid (u,w) \text{ is a back edge}\}\}

\(u\): articulation point \(\text{low}(\text{child}) \geq \text{dfn}(u)\)

---

<table>
<thead>
<tr>
<th>Vertex</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>(dfn)</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>(low)</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Find Biconnected Components

<table>
<thead>
<tr>
<th>vertex</th>
<th>dfn</th>
<th>low</th>
<th>child</th>
<th>low:child</th>
<th>low:dfn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
<td>null</td>
<td>null</td>
<td>null:4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>(3,4,0)</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>(2,0,0)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>(0,0,0)</td>
<td>4,5</td>
<td>0,5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>(1,0,0)</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>(5,5,5)</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>5</td>
<td>(6,5,5)</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>5</td>
<td>(7,8,5)</td>
<td>8,9</td>
<td>9,8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>9</td>
<td>(9,9,9)</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>8</td>
<td>(8,8,8)</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

Program: Initializaiton and Determining dfn and low

```c
void init(void)
{
    for (int i = 0; i < n; i++)
    {
        visited[i] = FALSE;
        dfn[i] = low[i] = -1;
    }
    num = 0;
}

void dfnlow(int u, int v)
{
    node_pointer ptr;
    int w;
    dfn[u] = low[u] = num++;
    for (ptr = graph[u]; ptr; ptr = ptr ->link)
    {
        w = ptr ->vertex;
        if (dfn[w] < 0) /*w is an unvisited vertex */
        {
            dfnlow(w, u);
            low[u] = MIN2(low[u], low[w]);
        } else if (w != v) low[u] =MIN2(low[u], dfn[w] );
    }
}
```
Program: Biconnected components of a graph

```c
void bicon(int u, int v)
{
    node_pointer ptr;
    int w, x, y;
    dfn[u] = low[u] = num ++;
    for (ptr = graph[u]; ptr; ptr = ptr->link)
    {
        w = ptr->vertex;
        if (v != w && dfn[w] < dfn[u])
            add(&top, u, w);
        else if (w != v)
            low[u] = MIN2(low[u], dfn[w]);
        if (dfn[w] < 0)
        {
            bicon(w, u);
            low[u] = MIN2(low[u], low[w]);
            if (low[w] >= dfn[u])
            {
                printf("New biconnected component: ");
                do { /* delete edge from stack */
                    delete(&top, &x, &y);
                    printf(" <%d, %d>", x, y);
                } while (!((x = u) && (y = w)));
                printf("n");
            }
        } else if (w != v)
            low[u] = MIN2(low[u], dfn[w]);
    }
}
```

Minimum Cost Spanning Tree

The cost of a spanning tree of a weighted undirected graph is the sum of the costs of the edges in the spanning tree. A minimum cost spanning tree is a spanning tree of least cost.

Example:

Select n-1 edges from a weighted graph of n vertices with minimum cost.
**Greedy Strategy**

- An **optimal solution** is constructed in stages.
- At each stage, the **best decision** is made at this time.
- Since this decision cannot be changed later, we make sure that the decision will result in a feasible solution.
- Typically, the selection of an item at each stage is based on a least cost or a highest profit criterion.
- Three different algorithms can be used:
  - Kruskal & Prim
  - Prim
  - Sollin

---

**Kruskal’s and Prim’s algorithm for MST**

- An **optimal solution** is constructed in stages.
  
  **Step 1**: Sort all edges into nondecreasing order.
  
  **Step 2**: Add the next smallest weight edge to the forest if it will not cause a cycle.
  
  **Step 3**: Stop if n-1 edges. Otherwise, go to Step 2.

```plaintext
T = {};  
while (T contains less than n-1 edges & E is not empty)  
{   choose a least cost edge (v,w) from E;  
    delete (v,w) from E;  
    if ((v,w) does not create a cycle in T)   add (v,w) to T  
      else discard (v,w);  
}  
if (T contains fewer than n-1 edges) printf("No spanning tree\n");
```

- **Time complexity**: $O(|E| \log |E|)$
Example of Kruskal’s and Prim’s algorithm

<table>
<thead>
<tr>
<th>Edge</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>10</td>
</tr>
<tr>
<td>(3,6)</td>
<td>15</td>
</tr>
<tr>
<td>(4,6)</td>
<td>20</td>
</tr>
<tr>
<td>(2,6)</td>
<td>25</td>
</tr>
<tr>
<td>(1,4)</td>
<td>30</td>
</tr>
<tr>
<td>(3,5)</td>
<td>35</td>
</tr>
</tbody>
</table>

### Prim’s Algorithm

**Step 1**: \( x \in V \), Let \( A = \{x\} \), \( B = V - \{x\} \).

**Step 2**: Select \((u, v) \in E\), \(u \in A\), \(v \in B\) such that \((u, v)\) has the smallest weight between \(A\) and \(B\).

**Step 3**: Put \((u, v)\) in the tree. \(A = A \cup \{v\}\), \(B = B - \{v\}\)

**Step 4**: If \(B = \emptyset\), stop; otherwise, go to Step 2.

\[ T = \{\}; TV = \{\} \]

while (T contains fewer than \(n-1\) edges)

\{ let \((u,v)\) be a least cost edge such that \(u \in TV\) and \(v \notin TV\)

if (there is no such edge ) break;

add \(v\) to TV; add \((u,v)\) to T;

\}

if (T contains fewer than \(n-1\) edges) printf("No spanning tree\n")

- Time complexity: \(O(|V|^2)\)
Example of Prim’s Algorithm

<table>
<thead>
<tr>
<th>Edge</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>10</td>
</tr>
<tr>
<td>(2,6)</td>
<td>25</td>
</tr>
<tr>
<td>(3,6)</td>
<td>15</td>
</tr>
<tr>
<td>(6,4)</td>
<td>20</td>
</tr>
<tr>
<td>(3,5)</td>
<td>35</td>
</tr>
</tbody>
</table>

Sollin’s Algorithm

Unlike Kruskal’s and Prim’s algorithms, Sollin’ algorithm selects multiple edges at each stage.

- At the start of a stage, the selected edges and all the n vertices form a spanning forest.
- During each stage, an minimum-cost edge is selected for each tree in the forest.
- It’s possible that two trees in the forest to select the same edge. Only one should be used.
- Also, it’s possible that the graph has multiple edges with the same cost. So, two trees may select two different edges that connect them together. Again, only one should be retained.
Example: Sollin’s Algorithm

<table>
<thead>
<tr>
<th>vertex</th>
<th>edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 -- 10 --&gt; 5, 0 -- 28 --&gt; 1</td>
</tr>
<tr>
<td>1</td>
<td>1 -- 14 --&gt; 6, 1 -- 16 --&gt; 2, 1 -- 28 --&gt; 0</td>
</tr>
<tr>
<td>2</td>
<td>2 -- 12 --&gt; 3, 2 -- 16 --&gt; 1</td>
</tr>
<tr>
<td>3</td>
<td>3 -- 12 --&gt; 2, 3 -- 18 --&gt; 6, 3 -- 22 --&gt; 4</td>
</tr>
<tr>
<td>4</td>
<td>4 -- 22 --&gt; 3, 4 -- 24 --&gt; 6, 5 -- 25 --&gt; 5</td>
</tr>
<tr>
<td>5</td>
<td>5 -- 10 --&gt; 0, 5 -- 25 --&gt; 4</td>
</tr>
<tr>
<td>6</td>
<td>6 -- 14 --&gt; 1, 6 -- 18 --&gt; 3, 6 -- 24 --&gt; 4</td>
</tr>
</tbody>
</table>

Sollin’s Algorithm (con.)
Shortest Paths and Transitive Closure

- Given a directed graph G=(V,E), a weighted function, w(e).
- How to find the shortest path from u to v?
- Single source all destinations:
  - Nonnegative edge costs: Dijkstra’s algorithm
  - General weights: Bellman-Ford’s algorithm
- All pairs shortest path
  - Floyd’s algorithm

Single Source All Destinations: Nonnegative Edge Costs

- Given a directed graph G=(V,E), a weighted function, w(e), and a source vertex v₀
- We wish to determine a shortest path from v₀ to each of the remaining vertices of G

<table>
<thead>
<tr>
<th>Path</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) v₀v₂</td>
<td>10</td>
</tr>
<tr>
<td>2) v₀v₂v₃</td>
<td>25</td>
</tr>
<tr>
<td>3) v₀v₂v₃v₁</td>
<td>45</td>
</tr>
<tr>
<td>4) v₀v₄</td>
<td>45</td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm

- Let S denotes the set of vertices, including v₀, whose shortest paths have been found.
- For w not in S, let distance[w] be the length of the shortest path starting from v₀, going through vertices only in S, and ending in w.
- We can find a vertex u not in S and distance[u] is minimum, and add u into S.
- Maintain distance properly
- Complexity: (|V|^2)

Example: Dijkstra’s algorithm

Cost adjacency matrix. All entries not shown are +∞.
All Pairs Shortest Paths

- Given a directed graph $G=(V,E)$, a weighted function, $w(e)$.
- How to find every shortest path from $u$ to $v$ for all $u,v$ in $V$?
Floyd’s Algorithm

- Represent the graph G by its length adjacency matrix with length[i][j]
- If the edge <i, j> is not in G, the length adjacency matrix with length[i][j] is set to some sufficiently large number
- \( A^k[i][j] \) is the adjacency matrix with length of the shortest path from i to j, using only those intermediate vertices with an index \( \leq k \)
- Complexity: \( O(|V|^3) \)

\[ A^k[i][j] = \min \{ A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j] \} \quad k \geq 0 \]
Example: Floyd’s Algorithm

Example cont.
Activity Networks

- We can divide all but simplest projects into several subprojects called activity.
- Model it as a graph
  - Activity-on-Vertex (AOV) Networks
  - Activity-on-Edge (AOE) Networks

Activity on Vertex (AOV) Networks

- An activity on vertex, or AOV network, is a directed graph G in which the vertices represent tasks or activities and the edges represent the precedence relation between tasks

Example: C3 is C1’s successor, C1 is C3’s predecessor
**Topological Ordering**

A **topological order** is a **linear ordering** of the vertices of a graph such that, for any two vertices i and j, if i is a predecessor of j in the network then i precedes j in the ordering.

**Example:** C1 C2 C4 C5 C6 C8 C7 C10 C13 C12 C14 C15 C11 C9

May not unique

![Graph diagram](image)

**Topological Sort**

**Step1:** Find a vertex v such that v has no predecessor, output it. Then delete it from network.

**Step2:** Repeat this step until all vertices are deleted.

- Time complexity: $O(|V| + |E|)$

```c
for (i = 0; i <n; i++)
{
  if every vertex has a predecessor
  {    fprintf(stderr, “Network has a cycle.
“ ); exit(1);}
  pick a vertex v that has no predecessors;
  output v;
  delete v and all edges leading out of v from the network;
}
```
Example

- $v_0$ has no predecessor, output it. Then delete it and three edges.

- Choose $v_3$, output it.
- Final result: $v_0$, $v_3$, $v_2$, $v_1$, $v_4$, $v_5$

Issues in Data Structure Consideration

- Decide whether a vertex has any predecessors.
  - Each vertex has a count.
- Decide a vertex together with all its incident edges.
  - Adjacency list
Issues in Data Structure Consideration

- Decide whether a vertex has any predecessors.
  - Each vertex has a count.
- Decide a vertex together with all its incident edges.
  - Adjacency list

```plaintext
V0: count 0, link 1
V1: count 1, link 4
V2: count 1, link 4
V3: count 1, link 5
V4: count 3, link NULL
V5: count 2, link NULL
```

Decision

typedef struct node *node_pointer;
typedef struct node {
  int vertex;
  node_pointer link;
} node_pointer;
typedef struct {
  int count;
  node_pointer link;
} hdnodes;
hdnodes graph[MAX_VERTICES];
### Topological Sort

```c
void topsort (hdnodes graph [], int n)
{
    int i, j, k, top;    node_pointer ptr;
    /* create a stack of vertices with no predecessors */
    top = -1;
    for (i = 0; i < n; i++)
        if (!graph[i].count)
            {     graph[i].count = top;   top = i;     }
    for (i = 0; i < n; i++)
        if (top == -1)
            {   fprintf (stderr, "\n Network has a cycle. Sort terminated. \n"); exit(1);
            } else
            {  j = top; /* unstack a vertex */
                top = graph[top].count;
                printf("v%d, ", j);
                for (ptr = graph[j]. link; ptr ;ptr = ptr ->link )
                    {   /*decrease the count of the successor vertices of j*/
                        k = ptr ->vertex;
                        graph[k].count --;
                        if (!graph[k].count) /* add vertex k to the stack*/
                            {    graph[k].count = top;   top = k;  }
                    }
            }
}
```
Applications of AOE Network

- Evaluate performance
  - minimum amount of time
  - activity whose duration time should be shortened
  - ...

- Critical path
  - a path that has the longest length
  - minimum time required to complete the project
  - ...

Critical Path

A critical path is a path that has the longest length. \((v_0, v_1, v_4, v_7, v_8)\)

\[
\begin{align*}
  a_0 &= 6 \\
  a_1 &= 4 \\
  a_2 &= 5 \\
  a_3 &= 1 \\
  a_4 &= 1 \\
  a_5 &= 2 \\
  a_6 &= 9 \\
  a_7 &= 7 \\
  a_8 &= 4 \\
  a_9 &= 2 \\
  a_{10} &= 4
\end{align*}
\]

\[
6 + 1 + 7 + 4 = 18 \text{ (Max)}
\]
**Earliest Time**

- The earliest time of an activity, $a_i$, can occur is the length of the longest path from the start vertex $v_0$ to $a_i$'s start vertex. (Ex: the earliest time of activity $a_7$ can occur is 7.)
- We denote this time as $\text{early}(i)$ for activity $a_i$.
  \[ \therefore \text{early}(6) = \text{early}(7) = 7. \]

**Latest Time**

- The latest time, $\text{late}(i)$, of activity, $a_i$, is defined to be the latest time the activity may start without increasing the project duration.
- Ex: $\text{early}(5) = 5$ & $\text{late}(5) = 8$; $\text{early}(7) = 7$ & $\text{late}(7) = 7$
Critical Activity

- A critical activity is an activity for which early(i) = late(i).
- The difference between late(i) and early(i) is a measure of how critical an activity is.

Calculation of Earliest Times

Let activity \( a_i \) is represented by edge \((u, v)\).
- early \( (i) = \text{earliest}[u] \)
- late \( (i) = \text{latest}[v] - \text{duration of activity} \ a_i \)

We compute the times in two stages: a forward stage and a backward stage.

The forward stage:
- Step 1: earliest \([0] = 0\)
- Step 2: earliest \([j]\) = max \{earliest \([i]\) + duration of \((i, j)\)\} for \( i \) in \( P(j) \)
  - \( P(j) \) is the set of immediate predecessors of \( j \).
Calculation of Latest Times

- The backward stage:
  - Step 1: latest[n-1] = earliest[n-1]
  - Step 2: latest [j] = min {latest [i] - duration of (j, i)}
    
    S(j) is the set of vertices adjacent from vertex j.

Computing Latest from Topological Sort

Graph with non-critical activities deleted

<table>
<thead>
<tr>
<th>Activity</th>
<th>Early</th>
<th>Late</th>
<th>L - E</th>
<th>Critical</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>a1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>a2</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>a3</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>a4</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>a5</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>a6</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>a7</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>a8</td>
<td>7</td>
<td>10</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>a9</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>a10</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>Yes</td>
</tr>
</tbody>
</table>