Data Structures: Graphs

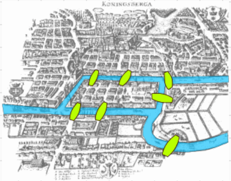
Wei-Mei Chen

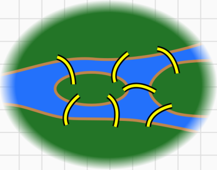
Department of Electronic and Computer Engineering
National Taiwan University of Science and Technology

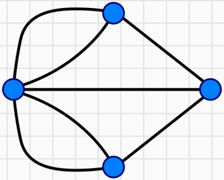
Introduction

Seven Bridges of Königsberg

- Is it possible to start at some location, walk across all the bridges exactly once, and return to the starting point?







`https://ed.ted.com/lessons/how-the-konigsberg-bridge-problem-changed-mathematics-dan-van-der-vieren`

Euler's Analysis

- areas $\mapsto$ vertices
  bridges $\mapsto$ edges
- Define the **degree** of a vertex to be the number of edges incident to it
- 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**.
- No Eulerian walk of the Königsberg bridge problem since all four vertices are of odd edges.

Applications

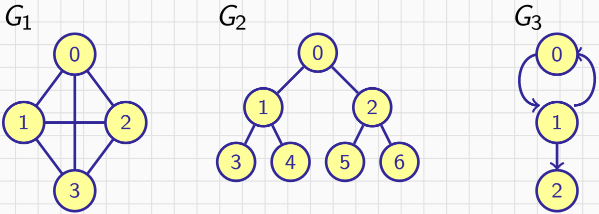
Some applications:

- Analysis of electrical circuits
- Finding shortest routes
- Project planning
- Identification of chemical compounds
- Statistical mechanics
- Genetics
- Cybernetics
- Linguistics
- Social Sciences

Definitions

- A graph, G, consists of two sets, V and E.
  - V is a finite, nonempty set of vertices.
  - E is set of pairs of vertices called edges.
- $V(G)$: the vertices of a graph G.
- $E(G)$: the edges of a graph G.
- Graphs can be either undirected graphs or directed graphs.
  - For a undirected graph, a pair of vertices (u, v) or (v, u) represent the same edge.
  - For a directed graph, a directed pair $\langle u, v \rangle$ has u as the tail and the v as the head. Therefore, $\langle u, v \rangle$ and $\langle v, u \rangle$ represent different edges.

Examples for Graphs



$V(G_1) = \{0, 1, 2, 3\}$ $\quad E(G_1) = \{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)\}$

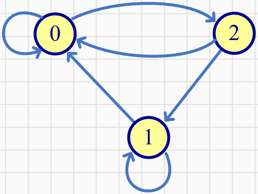
$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$ $\quad E(G_2) = \{(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)\}$

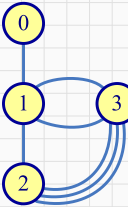
$V(G_3) = \{0, 1, 2\}$ $\quad E(G_3) = \{\langle 0,1\rangle, \langle 1,0\rangle, \langle 1,2\rangle\}$

Graph Restrictions

- A graph may not have an edge from a vertex back to itself.
  - (v, v) or $\langle v, v \rangle$ are called self edge or self loop.
  - If a graph with self edges, it is called a graph with self edges.
- A graph may not have multiple occurrences of the same edge.

☞ If without this restriction, it is called a **multigraph.**





Complete Graphs

- A **complete graph** is a graph that has the max number of edges.
- A complete undirected graph is an undirected graph with exactly $n(n-1)/2$ edges.
- A complete directed graph is a directed graph with exactly $n(n-1)$ edges.

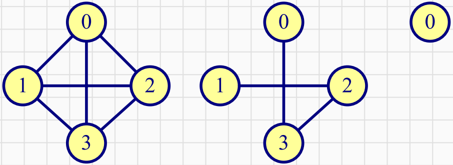
Graph Edges

- If (u, v) is an edge in $E(G)$, vertices u and v are **adjacent** and the edge (u, v) is **incident on** vertices u and v.
- For a directed graph, $\langle u, v \rangle$ indicates u is **adjacent to** v and v is **adjacent from** u.

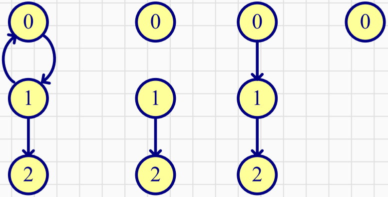
Subgraphs

- Subgraph: A subgraph of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.

Some subgraphs of G_1



Some subgraphs of G_3

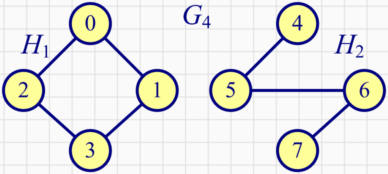


Paths

- A **path** from vertex u to vertex v in graph G is a sequence of vertices $u, i_1, i_2, \ldots, i_k, v$, such that $(u, i_1), (i_1, i_2), \ldots, (i_k, v)$ are edges in $E(G)$.
  ➠ A path $(0, 1), (1, 3), (3, 2)$ can be written as $0, 1, 3, 2$.
- The **length** of a path is the number of edges on it.
- A **simple path** is a path in which all vertices except possibly the first and last are distinct.
- A **cycle** is a simple path in which the first and last vertices are the same.
- Similar definitions of path and cycle can be applied to directed graphs.

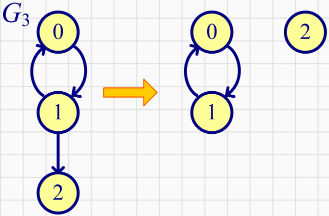
Connected Graphs

- Two vertices u and v are **connected** in an undirected graph $\Leftrightarrow$ there is a path from u to v (and v to u).
- An undirected graph is connected $\Leftrightarrow$ for every pair of distinct vertices u and v in $V(G)$ there is a path from u to v in G.
- A **connected component** of an undirected graph is a maximal connected subgraph.
- A tree is a **connected acyclic** graph.



Strongly Connected Graphs

- A directed graph G is strongly connected
  $\Leftrightarrow$ for every pair of distinct vertices u and v in $V(G)$, there is directed path from u to v and also from v to u.
- A strongly connected component is a maximal subgraph that is strongly connected.
- Example: There are two strongly connected component in G_3.



Degree of a Vertex

- The **degree** of a vertex is the number of edges incident to that vertex.
- If G is a directed graph, then we define
  - **in-degree of a vertex**: is the number of edges for which vertex is the head.
  - **out-degree of a vertex**: is the number of edges for which the vertex is the tail.
- For a graph G with n vertices and e edges, if d_i is the degree of a vertex i in G, then the number of edges of G is

$$e = \frac{\sum_{i=0}^{n-1} d_i}{2}$$

ADT Graph

ADT *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 $graph \in Graph$, $v \in v_1$, $v_2 \in Vertices$

Graph Create()	::=	**return** an empty graph
Graph InsertVertex(*graph*, v)	::=	**return** a graph with v inserted. v has no incident edges.
Graph InsertEdge(*graph*, v_1, v_2)	::=	**return** a graph with a new edge between v_1 and v_2
Graph DeleteVertex(*graph*, v)	::=	**return** a graph in which v and all edges incident to it are removed
Graph DeleteEdge(*graph*, v_1, v_2)	::=	**return** a graph in which the edge (v_1, v_2) is removed. Leave the incident nodes in the graph.
Boolean IsEmpty (*graph*)	::=	**if** (*graph* == empty graph) **return** *TRUE* **else return** *FALSE*
List Adjacent(*graph*, v)	::=	**return** a list of all vertices that are adjacent to v

Graph Representations

☞ Three most commonly used representations

- Adjacency matrices,
- Adjacency lists, and
- Adjacency multilists.

☞ Answer some questions about graphs

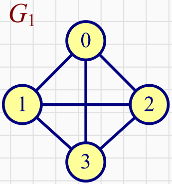
- How many edges are there in G?
- Is G connected?

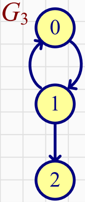
☞ Good representation brings some speed-up.

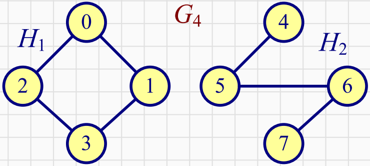
Adjacency Matrices

- Let $G(V, E)$ be a graph with n vertices, $n \geq 1$. The adjacency matrix of G is a two-dimensional $n \times n$ array, a.
  - $a[i][j] = 1 \Leftrightarrow$ the edge (i, j) is in $E(G)$.
  - The adjacency matrix for a undirected graph is symmetric. Note that it may not be the case for a directed graph.
- The space needed to represent a graph using its adjacency matrix is n^2 bits.
- For an undirected graph the degree of any vertex i is its row sum:
$$\sum_{j=0}^{n-1} a[i][j]$$
- For a directed graph, the row sum is the out-degree and the column sum is the in-degree.

Example - Adjacency Matrices







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

$$\begin{array}{c c} & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$\begin{array}{c c} & \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} & \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

How many edges in $G \Rightarrow O(n^2)$

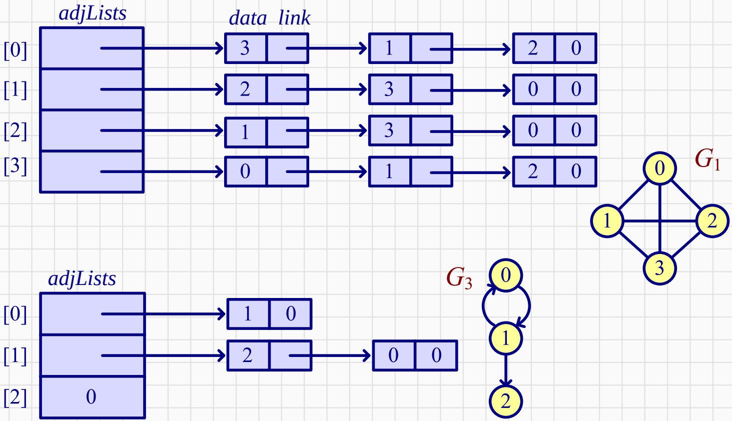
Adjacency Lists $\downarrow$

Adjacency Lists

Use n chains to represent the n vertices.

- Each list has a head node.
- Each node in the chain contains two fields: *data* and *link*.
  ☞ *data*: stores the indices of vertices adjacent to a vertex i.
- For an undirected graph with n vertices and e edges, we need n head nodes and $2e$ chain nodes.
- The degree of any vertex may be determined by counting the number of nodes in its adjacency list.
- The number of edges in G can be determined in $O(n + e)$.

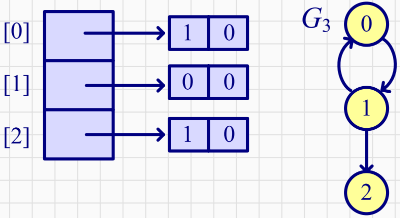
Example - Adjacency Lists



Inverse Adjacency Lists

- For a directed graph (also called digraph),
  - The out-degree of any vertex can be determined by counting the number of nodes in its adjacency list.
  - Determining the in-degree of any vertex is a little more complex

    ➠ keeping another set of lists called inverse adjacency lists.



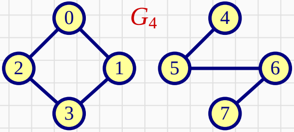
Using Sequential Lists

Use sequential list, the adjacency lists can be packed into an integer array *node* with $n + 2e + 1$ elements.

- $node[i] \leftarrow$ the starting point of the list for vertex i for $0 \leq i \leq n-1$
- $node[n] \leftarrow n + 2e + 1$
- The vertices adjacent from vertex i are stored in $node[i], \ldots, node[i+1]-1$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

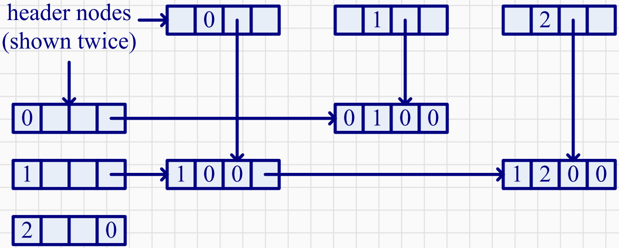
$n = 8$
$e = 7$
$n + 2e + 1 = 23$



Orthogonal List Representation

One simplified version for matrix representation

- the head of the edge represented by the node
- the tail of the edge represented by the node
- links for row chains
- links for column chains



$$\begin{array}{c c} & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

Adjacency Multilists

In the adjacency-list representation of an undirected graph, each edge is represented by two entries.

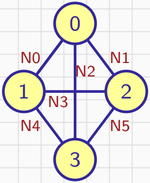
vertex 0: N0 $\to$ N1 $\to$ N2

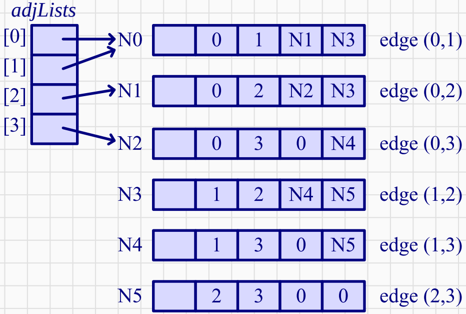
vertex 1: N0 $\to$ N3 $\to$ N4

vertex 2: N1 $\to$ N3 $\to$ N5

vertex 3: N2 $\to$ N4 $\to$ N5

marked	*vertex*1	*vertex*2	*link*1	*link*2





Weighted Edges

- Very often the edges of a graph have weights associated with
  - distance from one vertex to another
  - cost of going from one vertex to an adjacent vertex.
- To represent weight, we need additional field, weight, in each entry.
- A graph with weighted edges is called a *network*.

Operations

Graph Operations

- A general operation on a graph G is to visit all vertices in G that are reachable from a vertex v.
  - Depth-first search $\approx$ preorder tree traversal
  - Breadth-first search $\approx$ level order tree traversal
- We assume that the linked **adjacency list** representation for graph is used.

Depth-First Search

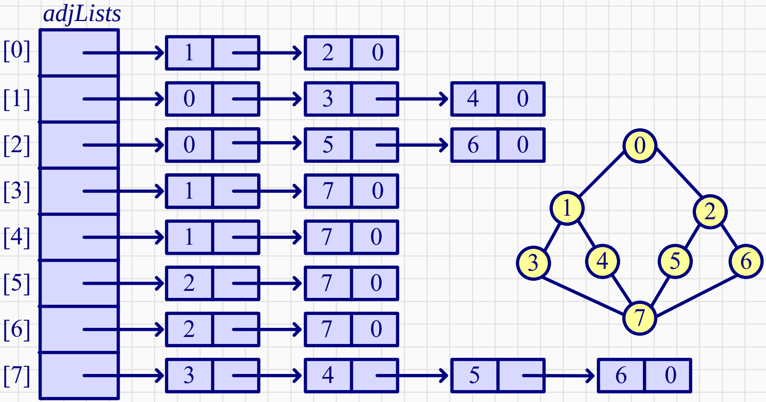
dfs(v): start from vertex v

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES]
```

```
void dfs(int v)
{
    nodePointer w;
    visited[v] = TRUE;
    printf("%5d",v);
    for(w = graph[v]; w; w = w->link)
        if (!visited[w->vertex])
            dfs(w->vertex);
}
```

Recursive Algorithm

Graph G and Its Adjacency Lists



dfs(0): $0, 1, 3, 7, 4, 5, 2, 6$

Analysis of `dfs`

- If G is represented by its adjacency lists, the time complexity of `dfs` is $O(e)$.

  $\because$ `dfs` examines each node in the adjacency lists at most once.

- If G is represented by its adjacency matrix, the time complexity of `dfs` is $O(n^2)$.

  $\because$ `dfs` visits at most n vertices.

Breadth-First Search

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

Analysis of `bfs`

- If adjacency lists are used, the time complexity is $d_1 + d_2 + \cdots + d_n = O(e)$ where $d_i = degree(v_i)$
- If an adjacency matrix is used, the time complexity is $O(n^2)$.

Connected Components

```
void connect(void)
{
    int i;
    for (i = 0 ; i < n ; i++)
        if (!visited [i]) {
            dfs(i);
            printf("\n");
        }
}
```

Analysis:

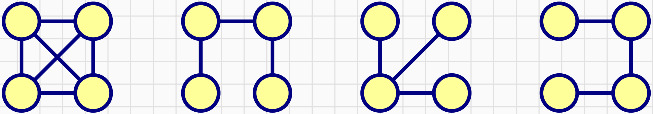
- adjacency list:
  dfs: $O(e)$
  for loop: $O(n)$
  $\Rightarrow$ total: $O(e+n)$
- adjacency matrix: $O(n^2)$

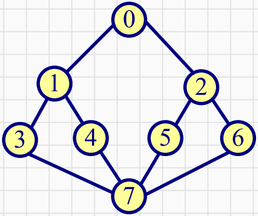
If G is connected, a dfs or bfs starting at any vertex visits all vertices in G.

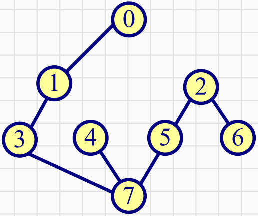
➠ The edges is divided into two sets:

- T: the set of edges traversed (tree edges)
- N: the set of remaining edges (nontree edges)

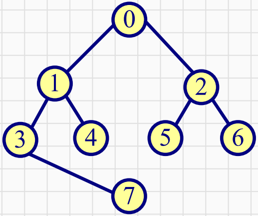
A **spanning tree** is any tree that consists solely of the edges in G and that includes all the vertices in G.







dfs(0) spanning tree



bfs(0) spanning tree

dfs(0): 0, 1, 3, 7, 4, 5, 2, 6

bfs(0): 0, 1, 2, 3, 4, 5, 6, 7

If we add $(v, w) \in N \Rightarrow$ form a cycle

The Number of Edges of Spanning Trees

- A spanning tree is a minimal subgraph, G', of G such that $V(G') = V(G)$, and G' is connected.
  ("Minimal" ➜ one with the fewest number of edges)
- Any connected graph with n vertices must have at least $n-1$ edges, and all connected graphs with $n-1$ edges are trees.
  $\Rightarrow$ A spanning tree has $n-1$ edges.
- Application: the design of communication networks
  ☞ A weighted graph.

MCST

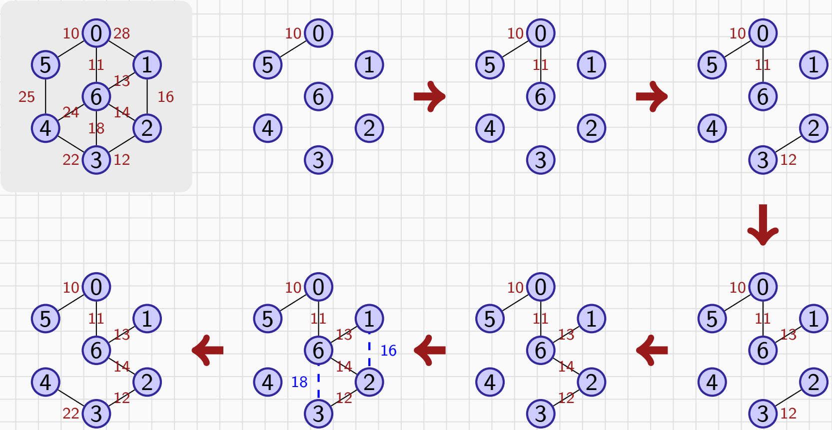
Minimum Cost Spanning Trees

- The **cost** of a spanning tree of a weighted, undirected graph is the sum of the costs (weights) of the edges in the spanning tree.
- A **minimum cost spanning tree** is a spanning tree of least cost.
- Three greedy-method algorithms available to obtain a minimum-cost spanning tree of a connected, undirected graph.
  - Kruskal's algorithm
  - Prim's algorithm
  - Sollin's algorithm
- **Note:** We must use only edges within the graph and exactly $n-1$ edges. We may not use edges that would produce a cycle.

Kruskal's Algorithm

- Kruskal's algorithm builds a minimum-cost spanning tree T by adding edges to T one at a time.
- The algorithm selects the edges for inclusion in T in nondecreasing order of their cost. An edge is added to T if it does not form a cycle with the edges that are already in T.
- Complexity: $O(e \log e)$
- Kruskal's algorithm in P. 295 of our textbook.

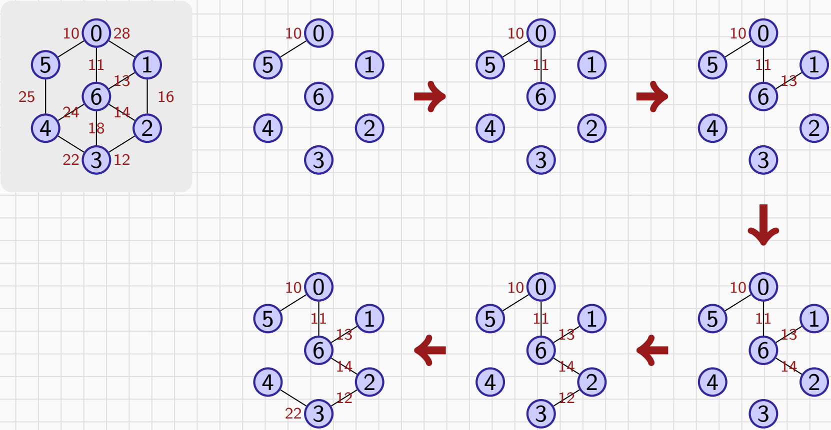
Stages in Kruskal's Algorithm



Prim's Algorithm

- Prim's algorithm constructs the minimum-cost spanning tree edge by edge.
- The set of selected edges forms a tree at all times when using Prim's algorithm while a forest is formed when using Kruskal's algorithm.
- In Prim's algorithm, a least-cost edge (u, v) is added to T such that $T \bigcup \{(u, v)\}$ is also a tree. This repeats until T contains $n - 1$ edges.
- Prim's algorithm in Program 6.8 (P. 297) has a time complexity $O(n^2)$.

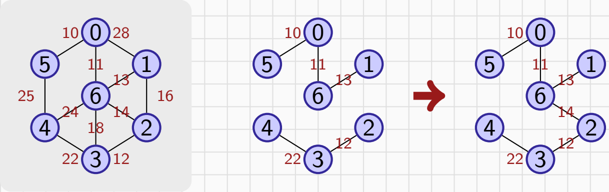
Stages in Prim's Algorithm



Sollin's Algorithm

- Contrast to Kruskal's and Prim's algorithms, Sollin's algorithm selects multiple edges at each stage.
- At the beginning, the selected edges and all the n vertices form a spanning forest.
  - During each stage, a 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.
  - Also, it's possible that the graph has multiple edges with the same cost.
- The algorithm terminates when there is only one tree at the end or no edges remain for selection.

Stages in Sollin's Algorithm



Shortest Paths

Shortest Path Problem

From city A to city B

- Is there a path from A to B?
- If there is more than one path from A to B, which is the shortest?

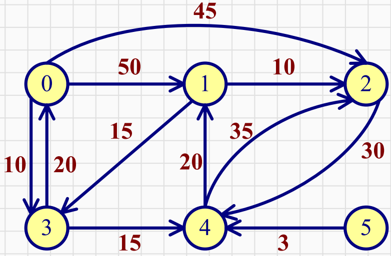
Note:

- length of a path: the sum of the weights of the edges
- starting vertex: source (v_0)
- last vertex: destination

Applications: Internet packet routing, Flight reservations, ...

Single Source All Destination: Cost ≥ 0

$G = (V, E)$: directed graph and a weighting function $w(e)$
Determine a shortest path from v_0 to each of vertices of G.



Shortest paths from 0

	path	length
1)	0, 3	10
2)	0, 3, 4	25
3)	0, 3, 4, 1	45
4)	0, 2	45

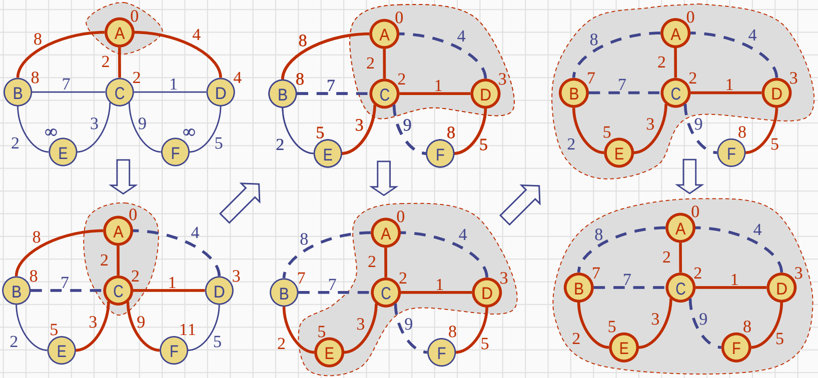
Dijkstra's Algorithm

- S: the set of vertices to which the shortest paths have already been found, including the source v_0.
- **For each vertex w not in S,**
  $\mathit{distance}[w]$: the length of the shortest path starting from v_0, going through vertices only in S and ending in w.
- Generating the paths in nondecreasing order of length leads to the following observations.
  1) If the next shortest path is to u, then the path from v_0 to u goes through only vertices in S.
  2) The destination of the next path generated must be u that has the **minimum distance** among all vertices not in S.
  3) The vertex u selected in 2) becomes a member of S and

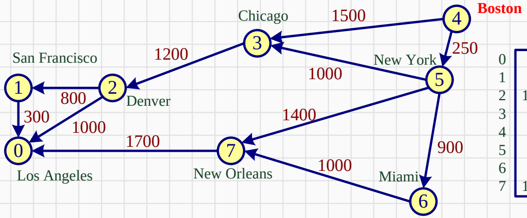
$$\mathit{distance}[w] = \mathit{distance}[u] + \mathit{length}(\langle u, w \rangle)$$

Edge Relaxation

We grow a "cloud" of vertices, beginning with vertex A and eventually covering all the vertices



Example - Dijkstra's Algorithm

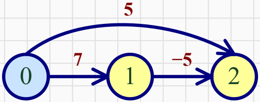


$$
\begin{array}{c|cccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
0 & 0 & & & & & & & \\
1 & 300 & 0 & & & & & & \\
2 & 1000 & 800 & 0 & & & & & \\
3 & & & 1200 & 0 & & & & \\
4 & & & & 1500 & 0 & 250 & & \\
5 & & & & 1000 & & 0 & 900 & 1400 \\
6 & & & & & & & 0 & 1000 \\
7 & 1700 & & & & & & & 0
\end{array}
$$

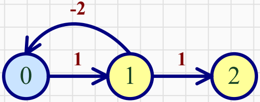
Iteration	Vertes Selected	Distance							
		LA	SF	DEN	CHI	BOST	NY	MIA	NO
		[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial	**----**				**1500**	**0**	**250**		
1	**5**				**1250**	**0**	**250**	**1150**	**1650**
2	**6**				**1250**	**0**	**250**	**1150**	**1650**
3	**3**			**2450**	**1250**	**0**	**250**	**1150**	**1650**
4	**7**	**3350**		**2450**	**1250**	**0**	**250**	**1150**	**1650**
5	**2**	**3350**	**3250**	**2450**	**1250**	**0**	**250**	**1150**	**1650**
6	**1**	**3350**	**3250**	**2450**	**1250**	**0**	**250**	**1150**	**1650**

Dijkstra's Algorithm Works for Nonnegative Cost

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
- What happens when a graph with a negative-length edge?
- What happens when a graph with a cycle of negative length?



A directed graph with a negative-length edge
→ *dist*[2]=?



A directed graph with a cycle of negative length
→ *dist*[2]=-∞

General Weights

- When negative edge lengths are permitted, the graph must not have cycles of negative length.
- When there are no cycles of negative length, there is a shortest path between any two vertices of an n-vertex graph that has at most $n - 1$ edges on it.
- $dist^{\ell}$: the length of a shortest path from the source $v \to u$ if the shortest path contains at most ℓ edges. Then
  $dist^{1}[u] = length[v][u]$ and
  $dist^{n-1}[u]$: the length of an unrestricted shortest path $v \to u$.
  ➠ Goal: compute $dist^{n-1}[u]$ for all u.

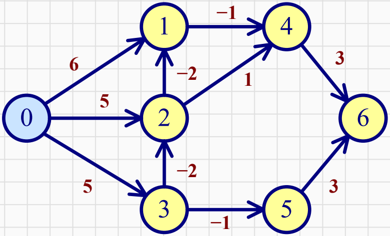
Bellman-Ford Algorithm

1. If the shortest path from v to u with at most $k, k > 1$, edges has no more than $k-1$ edges, then $dist^k[u] = dist^{k-1}[u]$.
2. If the shortest path from v to u with at most $k, k > 1$, edges has **exactly** k edges, then it is comprised of a shortest path from v to some vertex j followed by the edge $\langle j, u \rangle$. The path from v to j has $k-1$ edges, and its length is $dist^{k-1}[j]$. Then

$$dist^k[u] = \min\left\{ dist^{k-1}[u], \min_i \{ dist^{k-1}[i] + length[i][u] \} \right\}$$

```
void BellmanFord(int n, int v)
{
    for (int i = 0; i < n; i++)
        dist[i] = length[v][i];
    for (int k = 2; k <= n-1; k++)
        for (each u!=v and u has at least one incoming edge)
            for (each <i,u> in the graph)
                if (dist[u] > dist[i] + length[i][u])
                    dist[u] = dist[i] + length[i][u];
 }
```

Dynamic Programming



k	$dist^k[7]$						
	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

Analysis of `BellmanFord`

- The complexity of the "main" for loop:
  - adjacency matrices: $O(n^2)$
  - adjacency lists: $O(e)$
- The total complexity
  - adjacency matrices: $O(n^3)$
  - adjacency lists: $O(ne)$
- Improvement:
  - The main loop may be rewritten to terminate either after $n - 1$ iterations or after the first iteration in which no *dist* values are changed.
  - Maintain a queue of vertices i whose *dist* value changed on the previous iteration. These are the only values for i that need to be considered in the next iteration.

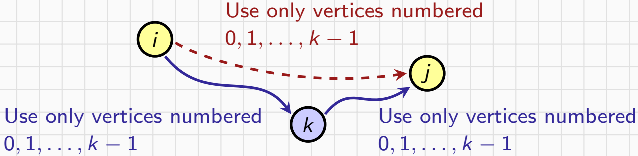
All Pairs Shortest Paths

- Find the distance between every pair of vertices in a weighted directed graph G.
  - We can make n calls to Dijkstra's algorithm (if no negative edges), which takes $O(en \log n)$ time.
  - Likewise, n calls to Bellman-Ford would take $O(n^2 e)$ time.
  - We can achieve $O(n^3)$ time using dynamic programming.
    - It works faster when G has edges with negative length, as long as the graphs have at least cn edges for suitable constant c.
- Dynamic programming algorithm:
  1. Establish a recursive property that gives the solution to an instance of the problem.
  2. Solve an instance of the problem in a bottom-up fashion by solving smaller instances first.

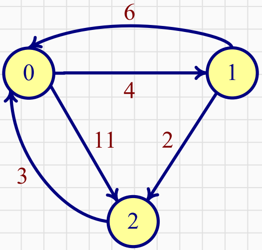
All-Pairs Shortest Paths (Cont.)

- Establish a recursive property
  - $A^{n-1}[i][j]$: the length of the shortest i-to-j path in G
  - $A^{k}[i][j]$: the length of the shortest path from i to j going through no intermediate vertex of index greater than k.
  - $A^{-1}[i][j]$: is just the $length[i][j]$
- The shortest path goes through vertex k.

$$A^{k}[i][j] = \min\left\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\right\}, k \geq 0$$



Example - All-Pairs Shortest-Paths Problem



(a)

A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

(b) A^{-1}

A^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

(c) A^{0}

A^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

(d) A^{1}

A^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

(e) A^{2}

Activity Networks

Activity Networks

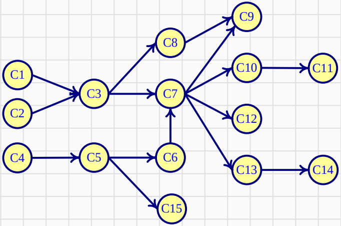
- A directed graph G in which the vertices represent tasks or activities and the edges represent precedence relations between tasks is an activity-on-vertex network or AOV network.
- Vertex i in an AOV network G is a predecessor of vertex j iff there is a directed path from vertex i to vertex j. i is an immediate predecessor of j iff $\langle i, j \rangle$ is an edge in G. If i is a predecessor of j, then j is an successor of i. If i is an immediate predecessor of j, then j is an immediate successor of i.

Topological Order

- A relation $\cdot$ is transitive iff it is the case that for all triples, i, j, k, $i \cdot j$ and $j \cdot k \Rightarrow i \cdot k$. A relation $\cdot$ is irreflexive on a set S if for no element x in S it is the case that $x \cdot x$. A precedence relation that is both transitive and irreflexive is a partial order.
- 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 linear ordering.

Applications

Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

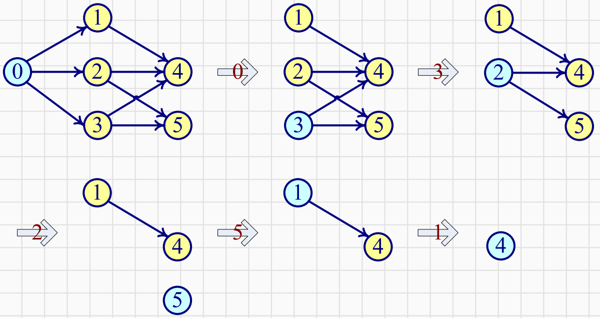


Sol1: C1, C2, C3, C4, C5, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9

Sol2: C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C12, C13, C14 .

Identify the Topological Order

- Include the vertex without precedence
- Delete the included vertex and its edges
- From the remaining, find the one without precedence and repeat
- Example $0 \to 3 \to 2 \to 5 \to 1 \to 4$



Internal Representation for Topological Sorting