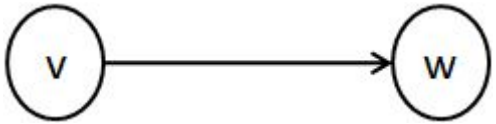
Graphs Algorithms

Unit 4: Lecture 03

Directed Graph Representation

A directed graph (**digraph**) G is a set of vertices (nodes) V and a set of arcs (directed edges) E

An arc is an ordered pair of vertices (v,w) where v is the tail and w is the head of the arc (represented as v→w)



In the directed graph shown, the arc is **from v to w** and w is **adjacent** to v

The vertices of a diagraph can be used to **represent objects** and the arcs **relationships** between them

Terminology

A **path** in a digraph is a sequence of vertices v1, v2, …, vn such that v1→v2→...vn are arcs

The **length of a path** is the number of arcs on the path: the length of path for the arc v→w is 1 and that of the above sequence is (n-1)

A **path is simple** if all vertices on the path except possibly the first and last are distinct

A digraph in which each arc and/or each vertex can have an associated label is called a **labeled digraph**

Representation of a Diagraph

Adjacency Matrix Representation: An **n×n matrix** A of **booleans** for n nodes such that A[i][j] is true iff there is an arc from vertex i to j

In adjacency matrix representation the **time required to access an element** is **independent of the size** of V and E

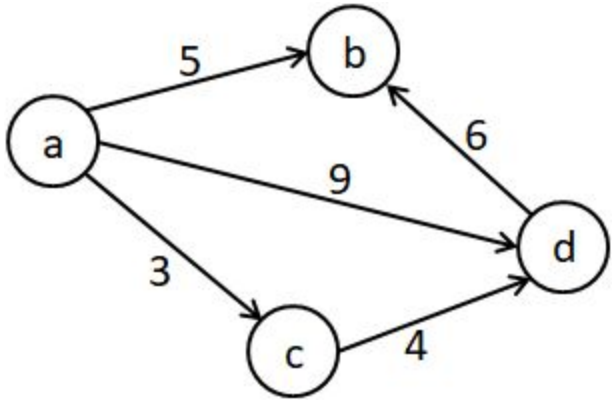
This representation is useful in those graph algorithms in which **frequent tests** (if an arc is present or not) are to be done

The adjacency matrix **for a labeled graph** contains labeled values in A[i][j]

Adjacency Matrix

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

Adjacency Matrix



A digraph

$$\begin{bmatrix} 0 & 5 & 3 & 9 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 6 & 0 & 0 \end{bmatrix}$$

Labeled Adjacency Matrix

Adjacency Matrix (contd.)

Adjacency matrix representation is easy for implementation

Reading or examining the adjacency matrix would require $O(n^2)$ time

Disadvantage: the matrix requires $\Omega(n^2)$ storage even if the graph has too less acrs (sparse matrix)

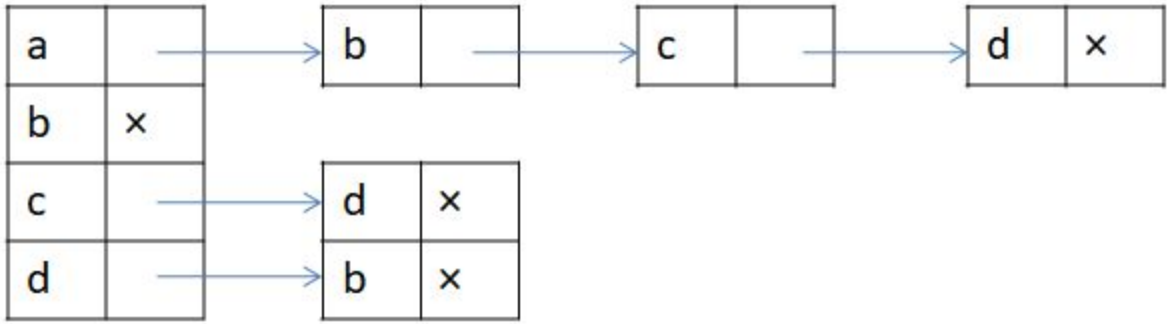
To avoid the disadvantage of adjacency matrix, an **adjacency list** is used most frequently

Adjacency List

The **adjacency list** for a vertex i is a list of all vertices adjacent to i

The graph can be represented by **an array** Head[] where Head[i] is a pointer to the adjacency list for vertex i

Example:



In the labeled graphs, the **labels of arcs** can be included in the linked list

Adjacency List (contd.)

The adjacency list representation requires **storage** proportional to (no. of vertices + no. of arcs)

The **list representation is useful** when (no. of arcs) << n^2

Disadvantage: it may take O(n) time to determine whether there is an arc from vertex i to vertex j

Shortest Path Algorithms

Single Source Shortest Path:

Determine the **cost of the shortest path** from the source to given vertex: Dijkstra's algorithm is most frequently used

Dijkstra's Algorithm (Greedy approach): the algorithm works by **maintaining a set S** of vertices whose shortest distance from the source is already known

Initially S contains only the **source** vertex

At each step, a remaining vertex is added to S whose distance from the source is **as short as possible**

Dijkstra’s Algorithm (contd.)

Dijkstra’s algorithm assumes the weights to be non-negative

The algorithm works in **$O(n^2)$** time for a digraph with n vertices and e arcs using an **adjacency matrix**

If $e << n^2$, **using adjacency list** the cost may be reduced to **O(e log n)**

All Pair Shortest Path (APSP)

The problem states to find a shortest path among all possible pairs in a graph with non-negative weights

Floyd's Algorithm: The algorithm uses an n×n matrix A[][] for a graph with n vertices with the following iteration:

$$A_k[i][j]=\text{Min}(A_{k-1}[i][j], A_{k-1}[i][k]+A_{k-1}[k][j])$$

The subscript k denotes the value of A[][] after kth iteration such that initially $A_0[i][j]$ contains only adjacent weights, other entries are assumed to be ∞

The algorithm tries to **find an intermediate vertex** k such that the direct weight of i and j (i→j) is outperformed by the weight i→k+k→j

APSP (contd.)

```
procedure Floyd(A[n][n])
        initialize A[][] with costs
        initialize P[][] with zeros      //Path matrix
        for k=1:n
                for i=1:n
                        for j=1:n
                                if(A[i][k]+A[k][j]<A[i][j])
                                        A[i][j]=A[i][k]+A[k][j]
                                        P[i][j]=k
end procedure
```

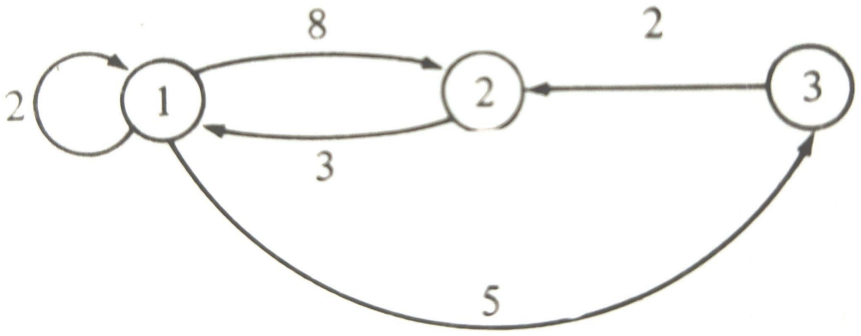
The **time complexity** of the algorithm is $O(n^3)$

APSP (contd.)

Path Extraction: To print the intermediate vertices on the shortest path from i to j, the following recursive procedure is used

```
procedure path(i,j)
        k=P[i][j]
        if(k==0) return
        path(i,k)
        print(k)
        path(k,j)
end procedure
```

APSP: An Example



$$A_0 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix} \qquad A_1 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \infty & 2 & 0 \end{bmatrix}$$

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

$$A_2 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix} \qquad A_3 = \begin{bmatrix} 0 & 7 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$$

APSP Example (contd.)

path(1,2)=?

From the matrix P[1][2]=3 (k≠0), path(1,2) enforces to estimate path(1,3) and path(3,2) with intermediate node 3

As path(1,3) and path(3,2) do not add any intermediate node further, the shortest path between 1 and 2 contains the intermediate node 3, i.e., 1→2 can be reached by 1→3→2

Note: Dijkstra's algorithms for single source shortest path can be applied to find the shortest path for all pairs which **costs $O(n^3)$**

Applications

Transitive Closure: When the cost matrix is just the adjacency matrix (with boolean values only), it represents only **the paths of length 1** (otherwise 0)

In this case, the matrix can be computed as A[i][j]=1 if there is a **path of length 1 or more** from i to j otherwise A[i][j]=0

Such a matrix is called the **transitive closure of the adjacency matrix**

Transitive closure of a relation matrix represents the minimum matrix such that the relation becomes transitive (aRb^bRc⇒aRc) after adding certain pairs

Finding transitive closure manually is a tedious task which may be computed easily with **Warshall's algorithm**

Warshall's Algorithm

```
procedure Warshall(A[n][n])
        initialize A[][] with costs
        for k=1:n
                for i=1:n
                        for j=1:n
                                if(A[i][j]==0)
                                        A[i][j]=A[i][k] AND A[k][j]      //logical AND
end procedure
```

The **cost** of the algorithm is $O(n^3)$

Applications (contd.)

Center of a Graph: The center of a digraph is defined as the **vertex with minimum eccentricity**

The **eccentricity** of a vertex v is defined as

Max (minimum length of a path from w to v), w is in V

The center is a **vertex that is closest to the vertex most distant** from it. As the definition suggests, finding the eccentricity leads estimating the shortest paths:

- Compute all pair shortest paths
- Find the **maximum cost in each column**: eccentricity of each vertex is obtained
- Find a vertex with **minimum eccentricity**: the center of the graph

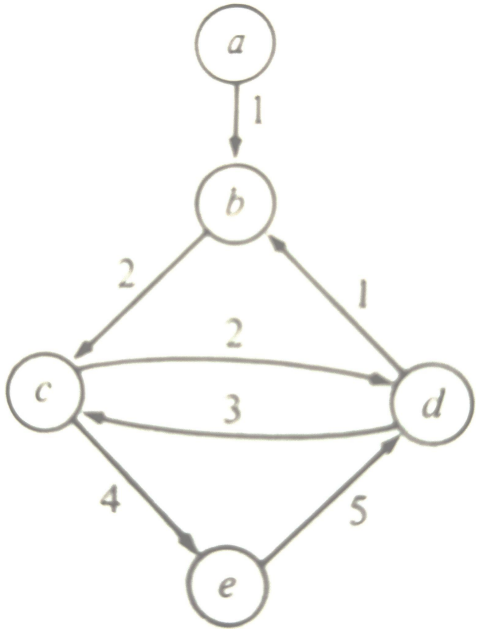
Center of a Graph

Example: For the given digraph, minimum eccentricity is 5

and thus the **center is the node d**

$$A = \begin{bmatrix} 0 & 1 & 3 & 5 & 7 \\ \infty & 0 & 2 & 4 & 6 \\ \infty & 3 & 0 & 2 & 4 \\ \infty & 1 & 3 & 0 & 7 \\ \infty & 6 & 8 & 5 & 0 \end{bmatrix}$$

$$Max \quad \infty \quad 6 \quad 8 \quad 5 \quad 7$$



Exercise

1. Compare Floyd’s algorithm with the Dijkstra’s algorithm applied multiple times to find all pair shortest paths.
2. Write the procedure to find the center of a digraph.

Reference Book

Data Structures and Algorithms: Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, 10th Impression, Pearson Education, New Delhi