Graph Algorithms

Dr. Natarajan Meghanathan
Associate Professor of Computer Science
Jackson State University
Jackson, MS 39217

Graph Traversal Algorithms

Pseudo Code of BFS

```
ALGORITHM  BFS(G)
    //Implements a breadth-first search traversal of a given graph
    //Input: Graph G = ⟨V, E⟩
    //Output: Graph G with its vertices marked with consecutive integers
    //            in the order they are visited by the BFS traversal
    mark each vertex in V with 0 as a mark of being "unvisited"
    count ← 0
    for each vertex v in V do
        if v is marked with 0
            bfs(v)

    bfs(v)
    //visits all the unvisited vertices connected to vertex v
    //by a path and numbers them in the order they are visited
    //via global variable count
    count ← count + 1;   mark v with count and initialize a queue with v
    while the queue is not empty do
        for each vertex w in V adjacent to the front vertex do
            if w is marked with 0
                count ← count + 1;   mark w with count
                add w to the queue
        remove the front vertex from the queue
```

BFS can be implemented with graphs represented as:
adjacency matrices: Θ(V^2); adjacency lists: Θ(|V|+|E|)

Comparison of DFS and BFS

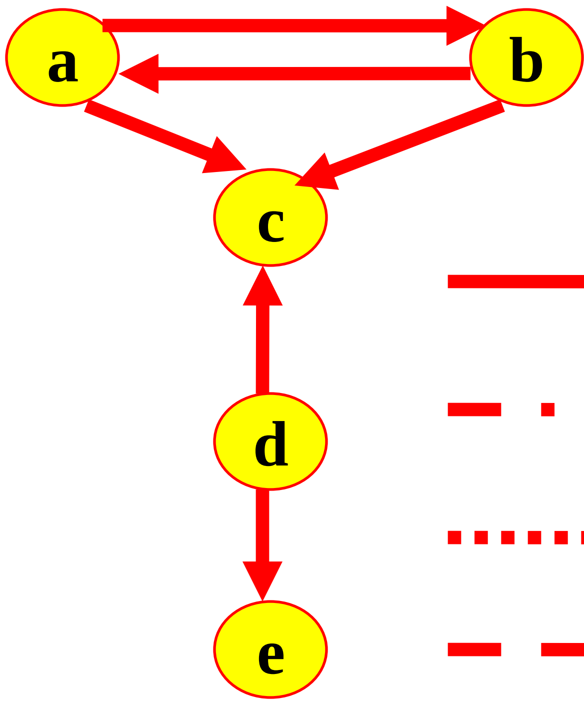
	DFS	BFS
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(\lvert V^2 \rvert)$	$\Theta(\lvert V^2 \rvert)$
Efficiency for adjacency lists	$\Theta(\lvert V \rvert + \lvert E \rvert)$	$\Theta(\lvert V \rvert + \lvert E \rvert)$

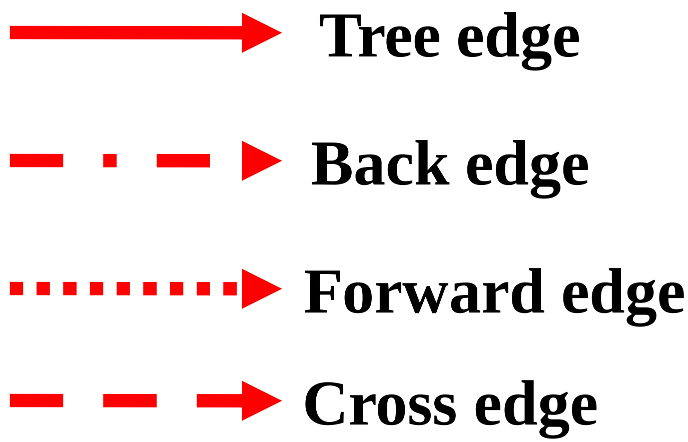
With the levels of a tree, referenced starting from the root node,
A back edge in a DFS tree could connect vertices at different levels; whereas, a cross edge in a BFS tree always connects vertices that are either at the same level or at adjacent levels.

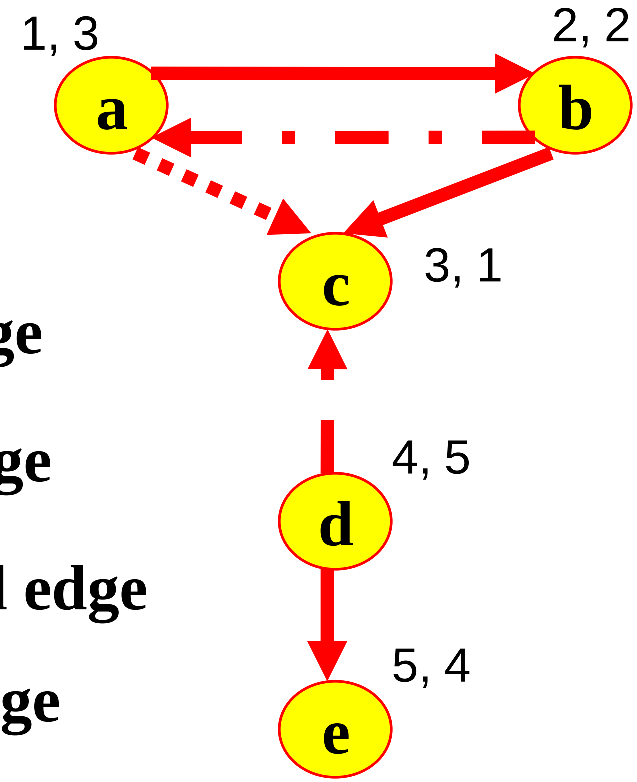
Source: Table 3.1: Levitin, 3rd Edition: Introduction to the Design and Analysis of Algorithms, 2012.

5.2 Topological Sort

DFS: Edge Terminology for directed graphs





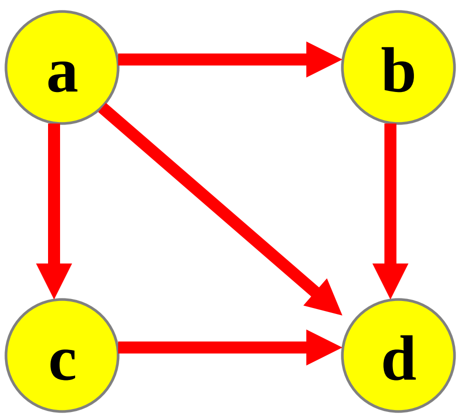


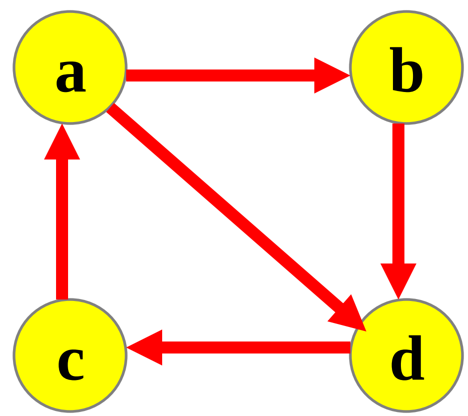
<u>Tree edge</u> – an edge from a parent node to a child node in the tree
<u>Back edge</u> – an edge from a vertex to its ancestor node in the tree
<u>Forward edge</u> – an edge from an ancestor node to its descendant node in the tree. The two nodes do not have a parent-child relationship. The back and forward edges are in a single component (the DFS tree).
<u>Cross edge</u> – an edge between two different components of the DFS Forest. So, basically an edge other than a tree edge, back edge and forward edge

Directed Acyclic Graphs (DAG)

- A directed graph is a graph with directed edges between its vertices (e.g., u → v).
- A DAG is a directed graph (digraph) without cycles.
  - A DAG is encountered for many applications that involve pre-requisite restricted tasks (e.g., course scheduling)

a DAG

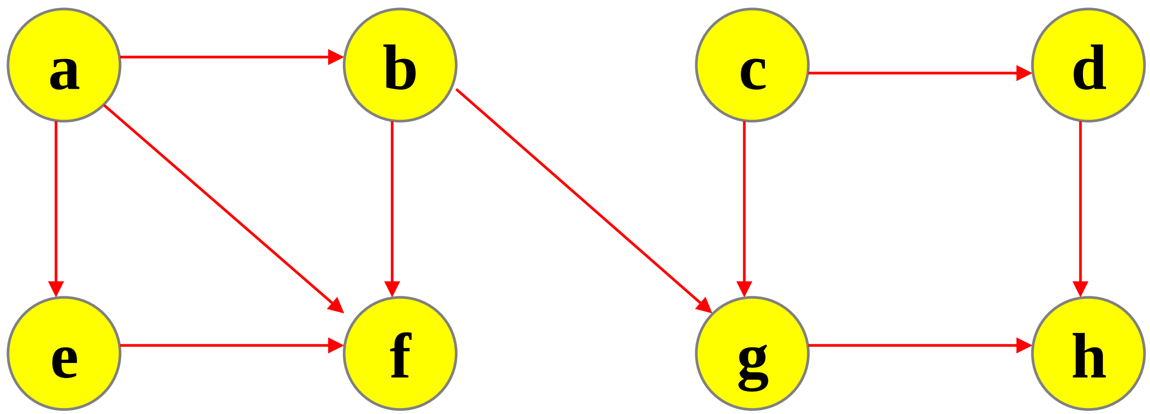




not a DAG

To test whether a directed graph is a DAG, run DFS on the directed graph. If a back edge is not encountered, then the directed graph is a DAG.

DFS on a DAG



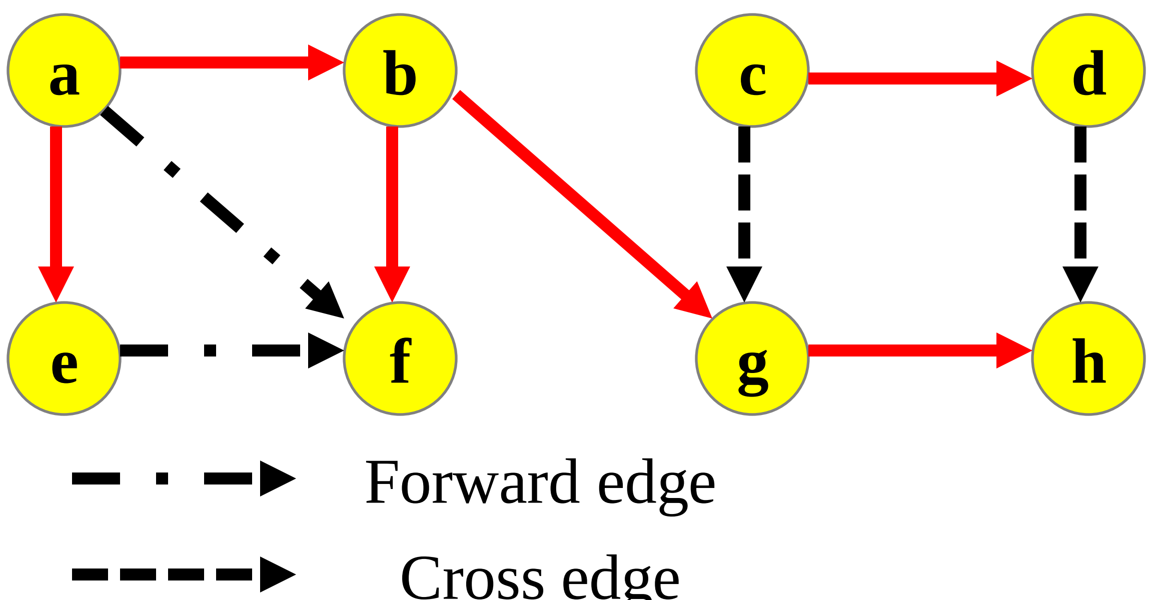
$h_{5,2}$

$g_{4,3}$

$f_{3,1}$

$b_{2,4}$ $e_{6,5}$ $d_{8,7}$

$a_{1,6}$ $c_{7,8}$



Order in which the Vertices are popped of from the stack

f	h	g	b	e	a	d	c

Reverse the order

Topological Sort

c	d	a	e	b	g	h	f

Dijkstra’s Shortest Path Algorithm

Shortest Path (Min. Wt. Path) Problem

- Path *p* of length *k* from a vertex *s* to a vertex *d* is a sequence (v_0, v_1, v_2, …, v_k) of vertices such that $v_0 = s$ and $v_k = d$ and (v_{i-1}, v_i) Є *E*, for *i* =1, 2,…, *k*

- Weight of a path *p* = (v_0, v_1, v_2, …, v_k) is $$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- The weight of a shortest path from *s* to *d* is given by

  $\delta(s, d) = \min \{w(p): s \xrightarrow{p} d$ if there is a path from *s* to *d*}
  $= \infty$ otherwise

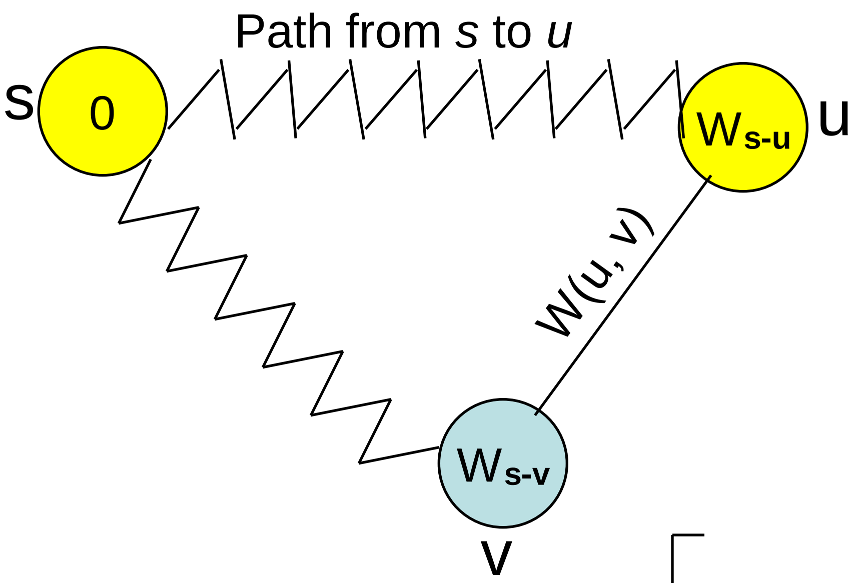
Examples of shortest path-finding algorithms:

- Dijkstra algorithm – $\Theta(|E|*log|V|)$
- Bellman-Ford algorithm – $\Theta(|E|*|V|)$

Dijkstra Algorithm

- **Assumption:** $w(u, v) >= 0$ for each edge $(u, v) \in E$
- **Objective:** Given G = (V, E, w), find the shortest weight path between a given source s and destination d
- **Principle:** Greedy strategy
- Maintain a minimum weight path estimate $d[v]$ from s to each other vertex v.
- At each step, pick the vertex that has the smallest minimum weight path estimate
- **Output:** After running this algorithm for $|V|$ iterations, we get the shortest weight path from s to all other vertices in G
- Note: Dijkstra algorithm does not work for graphs with edges (other than those leading from the source) with negative weights.

Principle of Dijkstra Algorithm



<u>Principle in a nutshell</u>

During the beginning of each iteration we will pick a vertex *u* that has the minimum weight path to *s*. We will then explore the neighbors of *u* for which we have not yet found a minimum weight path. We will try to see if by going through *u*, we can reduce the weight of path from *s* to *v*, where *v* is a neighbor of *u*.

Relaxation Condition

If $W_{s-v} > W_{s-u} + W(u, v)$ then

  $W_{s-v} = W_{s-u} + W(u, v)$

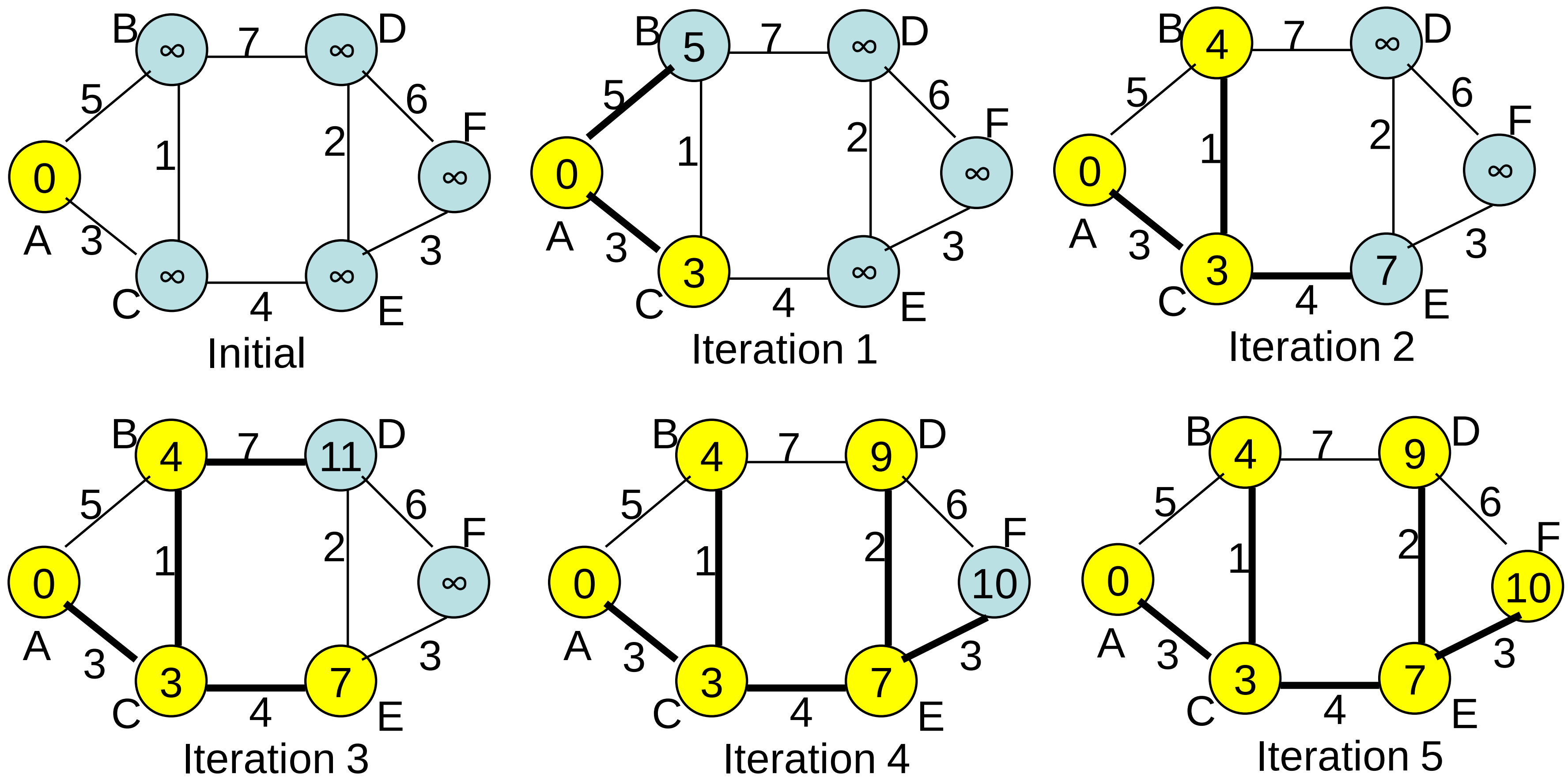
  Predecessor (v) = u

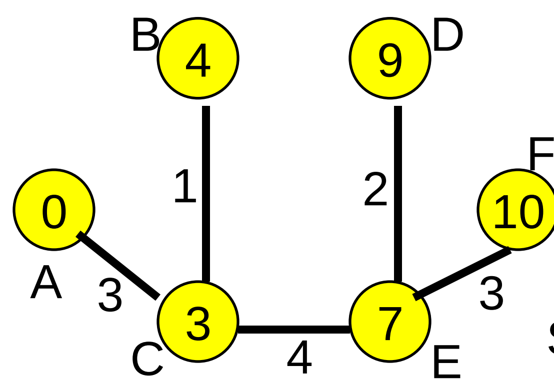
else

  Retain the current path from *s* to *v*

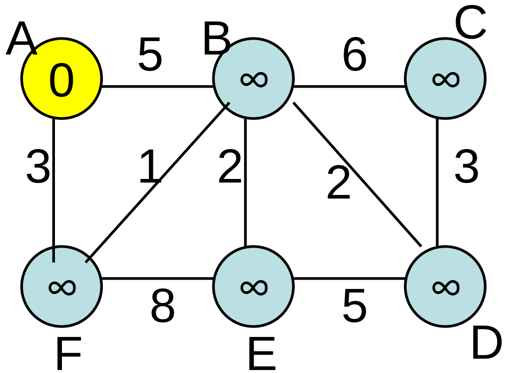
<u>Note:</u> Sub-path of a shortest path is also a shortest path. For example, if *s – a – c – f – g – d* is the minimum weight path from *s* to *d*, then *c – f – g – d* and *a – c – f – g* are the minimum weight paths from *c* to *d* and *a* to *g* respectively.

Dijkstra Algorithm Example 2

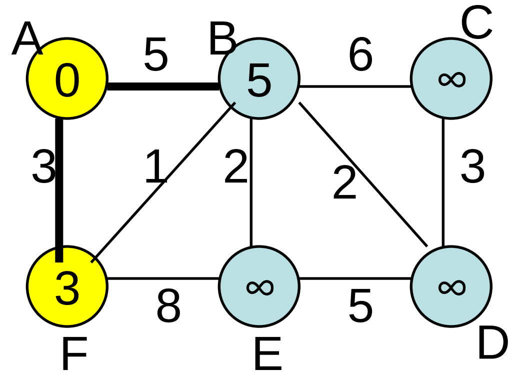




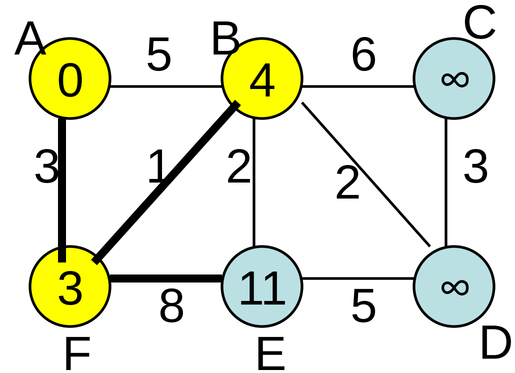
Shortest Path Tree



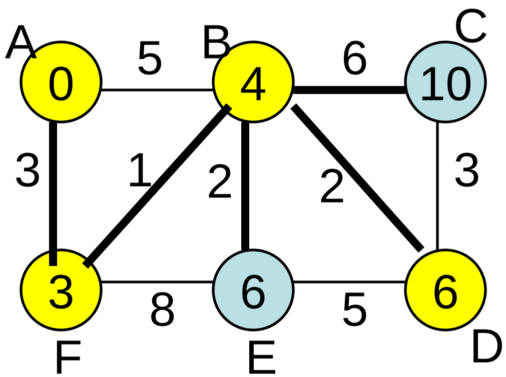
Initial



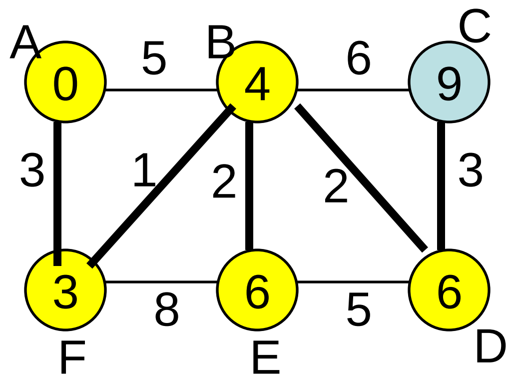
Iteration 1



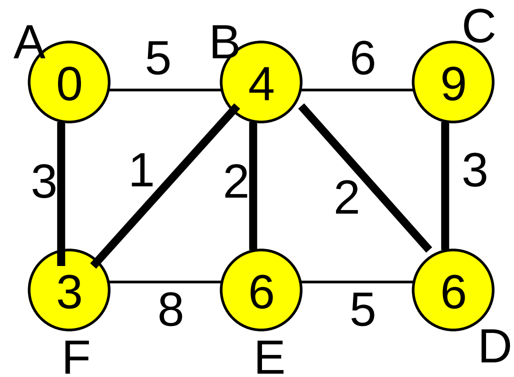
Iteration 2



Iteration 3

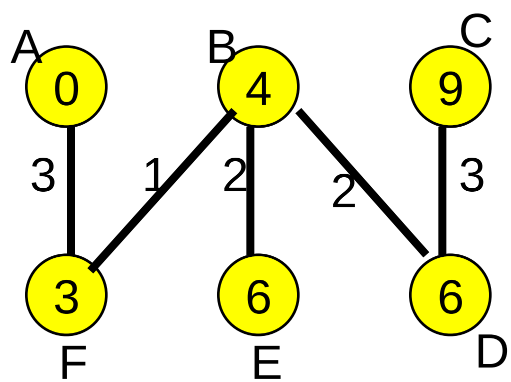


Iteration 4



Iteration 5

Dijkstra Algorithm Example 3



Shortest Path Tree

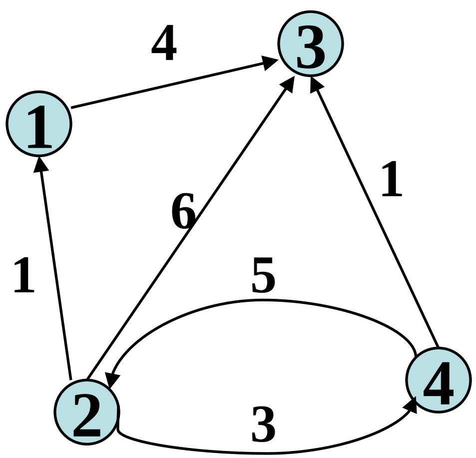
All Pairs Shortest Paths Problem

Floyd's Algorithm: All pairs shortest paths

Problem: **In a weighted (di)graph, find shortest paths between every pair of vertices**

idea: construct solution through series of matrices $D^{(0)}$, …, $D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

Example:

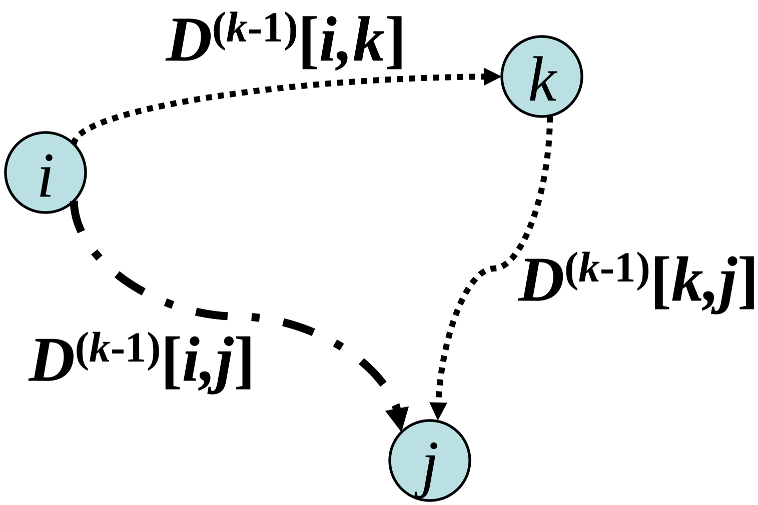


Floyd's Algorithm (matrix generation)

On the *k*-th iteration, the algorithm determines shortest paths between every pair of vertices *i, j* that use only vertices among 1,…,*k* as intermediate

$$D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$

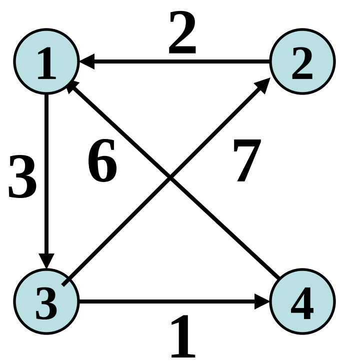
Predecessor Matrix



$$\pi_{ij}^{(0)} = \text{N/A if } i = j \text{ or } w_{ij} = \infty$$
$$= i \text{ if } i \neq j \text{ and } w_{ij} < \infty$$

$$\pi_{ij}^{(k)} = \pi_{ij}^{(k-1)} \quad \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$$
$$= \pi_{kj}^{(k-1)} \quad \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$$

Floyd's Algorithm (example)



$D^{(0)}$		v1	v2	v3	v4
	v1	0	∞	3	∞
	v2	2	0	∞	∞
	v3	∞	7	0	1
	v4	6	∞	∞	0

$\Pi^{(0)}$		v1	v2	v3	v4
	v1	N/A	N/A	v1	N/A
	v2	v2	N/A	N/A	N/A
	v3	N/A	v3	N/A	v3
	v4	v4	N/A	N/A	N/A

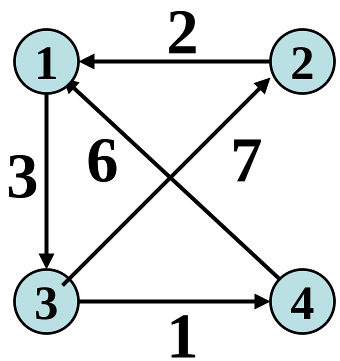
$D^{(1)}$		v1	v2	v3	v4
	v1	0	∞	3	∞
	v2	2	0	5	∞
	v3	∞	7	0	1
	v4	6	∞	9	0

$\Pi^{(1)}$		v1	v2	v3	v4
	v1	N/A	N/A	v1	N/A
	v2	v2	N/A	v1	N/A
	v3	N/A	v3	N/A	v3
	v4	v4	N/A	v1	N/A

$D^{(2)}$		v1	v2	v3	v4
	v1	0	∞	3	∞
	v2	2	0	5	∞
	v3	9	7	0	1
	v4	6	∞	9	0

$\Pi^{(2)}$		v1	v2	v3	v4
	v1	N/A	N/A	v1	N/A
	v2	v2	N/A	v1	N/A
	v3	v2	v3	N/A	v3
	v4	v4	N/A	v1	N/A

Floyd's Algorithm (example)

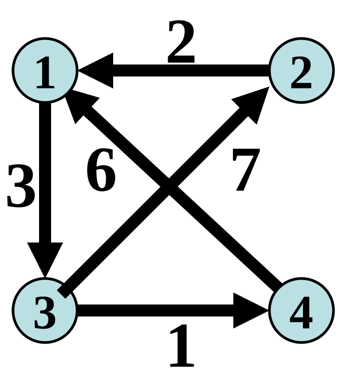


$D^{(3)}$	v1	v2	v3	v4
v1	0	10	3	4
v2	2	0	5	6
v3	9	7	0	1
v4	6	16	9	0

$\Pi^{(3)}$	v1	v2	v3	v4
v1	N/A	v3	v1	v3
v2	v2	N/A	v1	v3
v3	v2	v3	N/A	v3
v4	v4	v3	v1	N/A

$D^{(4)}$	v1	v2	v3	v4
v1	0	10	3	4
v2	2	0	5	6
v3	7	7	0	1
v4	6	16	9	0

$\Pi^{(4)}$	v1	v2	v3	v4
v1	N/A	v3	v1	v3
v2	v2	N/A	v1	v3
v3	v4	v3	N/A	v3
v4	v4	v3	v1	N/A



Deducing path for v2 to v4

π(v2-v4) = π(v2-v3) --> v3 --> v4
= π(v2-v1) --> v1 --> v3 --> v4
= v2 --> v1 --> v3 --> v4

Deducing path for v4 to v2

π(v4-v2) = π(v4-v3) --> v3 --> v2
= π(v4-v1) --> v1 --> v3 --> v2
= v4 --> v1 --> v3 --> v2

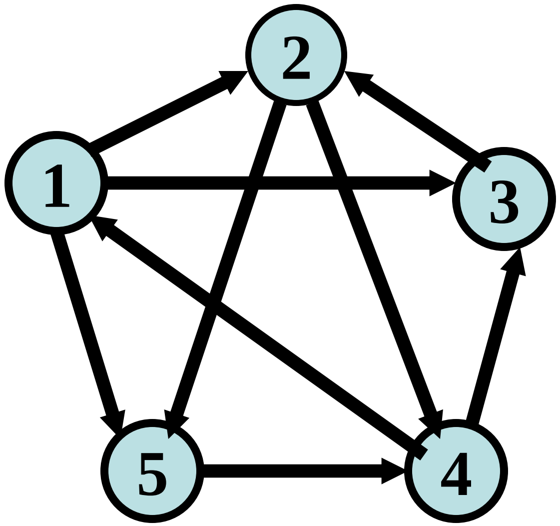
Floyd's Algorithm (pseudocode and analysis)

```
ALGORITHM  Floyd(W[1..n, 1..n])
    //Implements Floyd's algorithm for the all-pairs shortest-paths problem
    //Input: The weight matrix W of a graph with no negative-length cycle
    //Output: The distance matrix of the shortest paths' lengths
    D ← W //is not necessary if W can be overwritten
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
    return D
```

Time efficiency: $\Theta(n^3)$

Space efficiency: $\Theta(n^2)$

Floyd’s Algorithm (example)



	Weight Matrix				
	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$D^{(0)}$	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$\Pi^{(0)}$	v1	v2	v3	v4	v5
v1	N/A	v1	v1	N/A	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	N/A	N/A
v4	v4	∞	v4	N/A	N/A
v5	N/A	N/A	N/A	v5	N/A

$D^{(1)}$	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	5	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(1)}$	v1	v2	v3	v4	v5
v1	N/A	v1	v1	N/A	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	N/A	N/A
v4	v4	v1	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

$D^{(2)}$	v1	v2	v3	v4	v5
v1	0	3	8	4	-4
v2	∞	0	∞	1	7
v3	∞	4	0	5	11
v4	2	5	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(2)}$	v1	v2	v3	v4	v5
v1	N/A	v1	v1	v2	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	v2	v2
v4	v4	v1	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

Floyd's Algorithm (example)

$D^{(3)}$	v1	v2	v3	v4	v5
v1	0	3	8	4	-4
v2	∞	0	∞	1	7
v3	∞	4	0	5	11
v4	2	-1	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(3)}$	v1	v2	v3	v4	v5
v1	N/A	v1	v1	v2	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	v2	v2
v4	v4	v3	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

Deducing path from v3 to v1

π(v3-v1) = π(v3-v4) --> v4 --> v1

= π(v3-v2) --> v2 --> v4 --> v1

= v3 --> v2 --> v4 --> v1

$D^{(4)}$	v1	v2	v3	v4	v5
v1	0	3	-1	4	-4
v2	3	0	-4	1	-1
v3	7	4	0	5	3
v4	2	-1	-5	0	-2
v5	8	5	1	6	0

$\Pi^{(4)}$	v1	v2	v3	v4	v5
v1	N/A	v1	v4	v2	v1
v2	v4	N/A	v4	v2	v1
v3	v4	v3	N/A	v2	v1
v4	v4	v3	v4	N/A	v1
v5	v4	v3	v4	v5	N/A

Deducing path from v1 to v3

π(v1-v3) = π(v1-v4) --> v4 --> v3

π(v1-v5) --> v5 --> v4 --> v3

v1 --> v5 --> v4 --> v3

$D^{(5)}$	v1	v2	v3	v4	v5
v1	0	1	-3	2	-4
v2	3	0	-4	1	-1
v3	7	4	0	5	3
v4	2	-1	-5	0	-2
v5	8	5	1	6	0

$\Pi^{(5)}$	v1	v2	v3	v4	v5
v1	N/A	v3	v4	v5	v1
v2	v4	N/A	v4	v2	v1
v3	v4	v3	N/A	v2	v1
v4	v4	v3	v4	N/A	v1
v5	v4	v3	v4	v5	N/A

Minimum Spanning Tree Problem

- Given a weighted graph, we want to determine a tree that spans all the vertices in the tree and the sum of the weights of all the edges in such a spanning tree should be minimum.
- Two algorithms:
  - Prim algorithm
  - Kruskal Algorithm

- Prim algorithm is just a variation of Dijkstra algorithm with the relaxation condition being

If $v \in Q$ and d [v] > w (u, v) then
    d [v] $\leftarrow$ w (u, v)
    Predecessor (v) = u
End If

On each iteration, the algorithm expands the current tree in a greedy manner by attaching to it the nearest vertex not in the tree. By the 'nearest vertex', we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight.

- Kruskal algorithm: Consider edges in the increasing order of their weights and include an edge in the tree, if and only if, by including the edge in the tree, we do not create a cycle!!
- Note: Shortest Path trees need not always have minimum weight and minimum spanning trees need not always be shortest path trees.

Kruskal Algorithm

```
Begin Algorithm Kruskal (G = (V, E))
    A ← Φ // Initialize the set of edges to null set

    for each vertex v_i ϵ V do
        Component (v_i) ← i

    Sort the edges of E in the non-decreasing (increasing) order of weights

    for each edge (v_i, v_j) ϵ E, in order by non-decreasing weight do
            if (Component (v_i) ≠ Component (v_j) ) then
                A ← A U (v_i, v_j)

                if Component(v_i) < Component(v_j)  then
                   for each vertex v_k in the same component as of v_j do
                      Component(v_k) ← Component(v_i)
                else
                   for each vertex v_k in the same component as of v_i do
                      Component(v_k) ← Component(v_j )
                end if
            end if
    end for
    return A
End Algorithm Kruskal
```