Directed Graphs

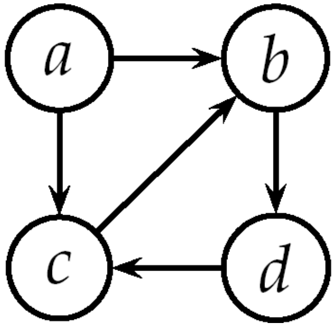
Definitions. Representations. ADT's. Single Source Shortest Path Problem (Dijkstra, Bellman-Ford, Floyd-Warshall). Traversals for DGs. Parenthesis Lemma. DAGs. Strong Components. Topological Sort

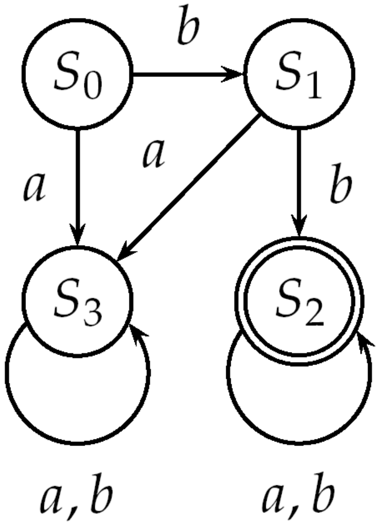
Directed Graphs. Definitions

- Directed graph (digraph): $G=(V, E)$
  - V : set of vertices (nodes, points); $n=|V|$
  - E : set of edges (arcs, directed lines); $e=|E|$
  - Arc: ordered pair (u, v)
    - arc is from u to v
    - u=tail v=head
    - v is adjacent to u
  - Path from v_1 to v_n : $\langle v_1, v_2, v_3, \ldots, v_n\rangle$ such that $v_1 \rightarrow v_2$, $v_2 \rightarrow v_3, \ldots, v_{n-1} \rightarrow v_n$ are arcs
    - Path length: number of edges on the path
    - Simple path: all vertices distinct (except possibly for last and first)
    - Simple cycle: simple path of length ≥1 that begins and ends at the same node

Directed Graphs. Representations

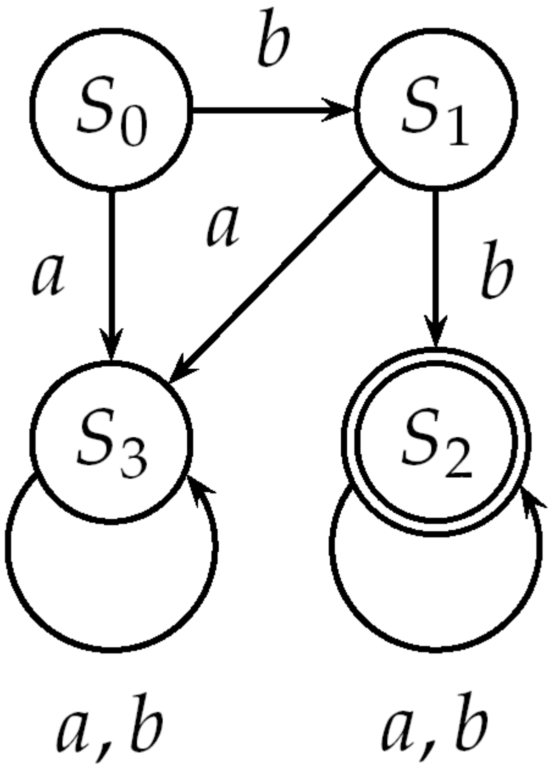
- Example
  - *b, d, c, b* – simple cycle of length 3
- Labeled digraph
  - Label: value of any given data type
  - Example: transition graph
- Representations
  - Adjacency matrix
  - Adjacency list





Directed Graphs. Representations

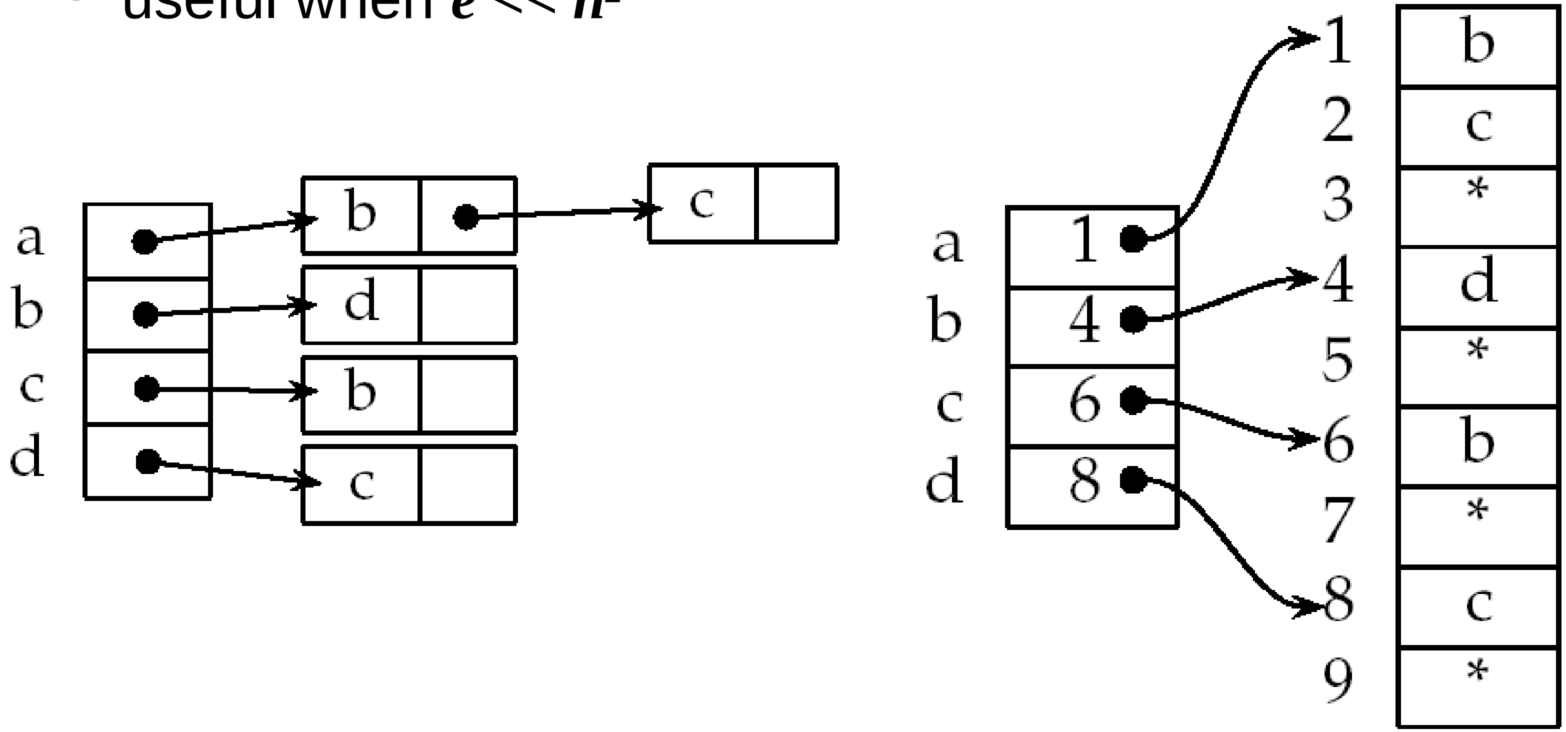
- Adjacency matrix: *n* × *n* of Booleans (0,1)
- Labeled adjacency matrix



$$\begin{array}{c|cccc|}
 & 0 & 1 & 2 & 3 \\
\hline
0 & & b & & a \\
1 & & & b & a \\
2 & & & a,b & \\
3 & & & & a,b \\
\hline
\end{array}$$

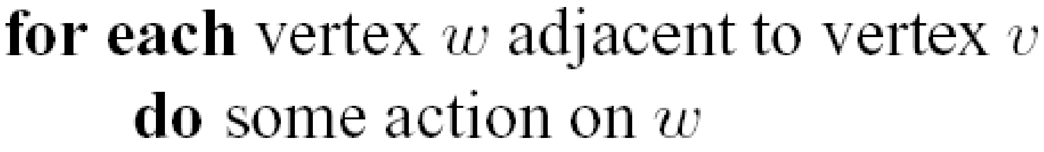
Directed Graphs. Representations

- Adjacency list
  - useful when $e << n^2$



Directed Graph ADT's

- Typical processing in programs



```
for each vertex $w$ adjacent to vertex $v$
    do some action on $w$
```

- Operations:
  - **first**(*v*) returns the index for the first vertex adjacent to v. The index for the null vertex (Λ) is returned if there is no vertex adjacent to *v*.
  - **next**(*v, i*) returns the index after index i for the vertices adjacent to *v*. Λ is returned if i is the last index for vertices adjacent to *v*.
  - **vertex**(*v, i*) returns the vertex with index *i* among the vertices adjacent to *v*.

Directed Graph ADT's

```
FIRST($v$)
for $i \leftarrow 1$ to $n$
    do if $A[v][i]$
        then return $i$
        else return $0$
```

```
NEXT($v, i$)
for $j \leftarrow i + 1$ to $n$
    do if $A[v][j]$
        then return $j$
        else return $0$
```

- Processing adjacency for a vertex

```
$i \leftarrow$ FIRST($v$)
while $i \neq 0$
    do $w \leftarrow$ VERTEX($v, i$)
        ▷ some action on $w$
        $i \leftarrow$ NEXT($w, i$)
```

The Single Source Shortest Path Problem

- Given:
  - directed graph G = (V,E) in which each arc has a nonnegative label, and
  - one vertex is specified as the *source*
- Determine:
  - the cost of the shortest path from the source to every other vertex in V
- *Weighted* graph: arcs are labeled with costs
- One solution: Dijkstra’s
  - Set *S* of vertices whose shortest distance from the source is already known.
  - At each step we add to *S* a remaining vertex *v* whose distance from the source is as short as possible.
  - We can always find a shortest path from the source to *v* that passes only through vertices in *S* (*special* path).
  - Use an array *D* to record the length of the shortest special path to each vertex.

Dijkstra's algorithm

- $G = (V,E)$, where $V = \{1, 2, \ldots, n\}$ and vertex 1 is the source

```
DIJKSTRA($G, C$)
1  $S \leftarrow \{1\}$
2  for $i \leftarrow 2$ to $n$
3        do $D[i] \leftarrow C[1][i]$   ▷ initialize D
4  for $i \leftarrow 1$ to $n$
5        do choose a vertex $w \in V \setminus S$ such that $D[w]$ is minimum
6            add $w$ to $S$
7            for each $v \in V \setminus S$
8                 do $D[v] \leftarrow \min(D[v], D[w] + C[w][v])$
```

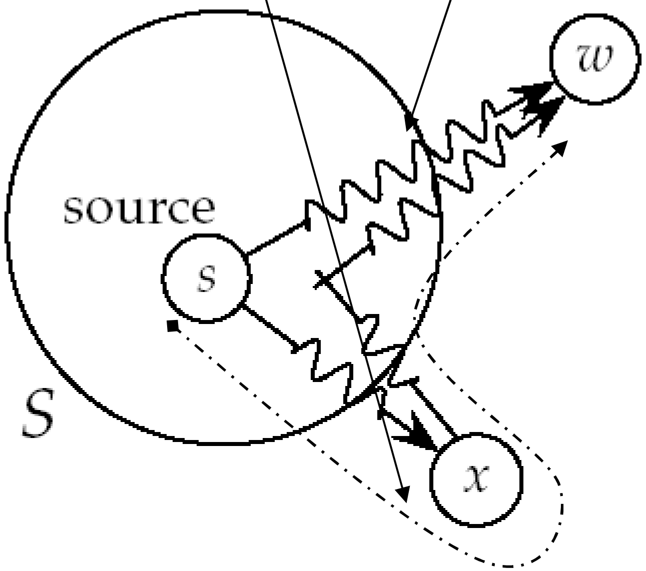
- loop at lines 7–8 takes ***O*(*n*)** time and it is executed ***n***–1 times, resulting in a total time of ***O*(*n*^{2})**

Running Time of Dijkstra's Algorithm

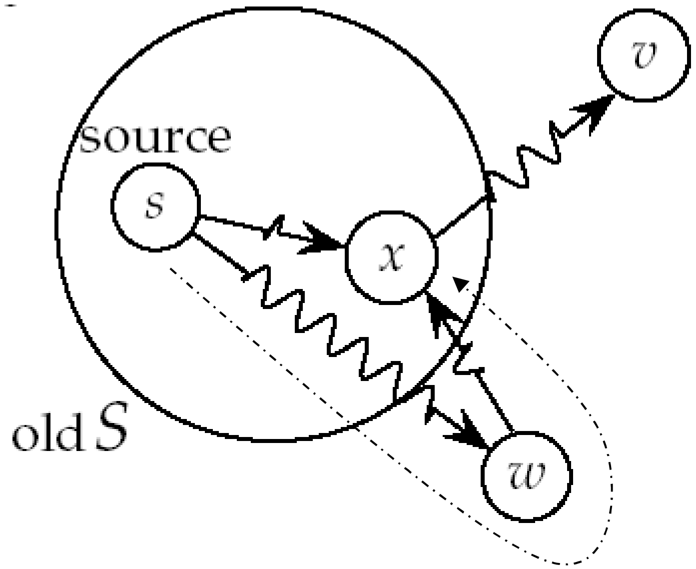
- loop at lines 7–8 takes $O(n)$ time and it is executed $\boldsymbol{n}-1$ times, resulting in a total time of $O(n^2)$
- If $\boldsymbol{e} \ll \boldsymbol{n}^2$ an adjacency list is a better choice
  - use priority queue implemented as a partially ordered tree (heap) for V - S.
  - loop at lines 7–8 can be implemented by going down the adjacency list for *w* and <u>updating</u> the distances in the priority queue
  - $\boldsymbol{e}$ updates of $\mathbf{O(log}\ \boldsymbol{n}\mathbf{):}$ total $\boldsymbol{O}\mathbf{(}\boldsymbol{e}\ \mathbf{log}\ \boldsymbol{n}\mathbf{)}$ (if $\boldsymbol{e} \ll \boldsymbol{n}^2$)

Correctness of Dijkstra’s Algorithm

(1) If length(s ⇨ x ⇨ w) < length(s ⇨ w), then s ⇨ x = special path < shortest special path to w.
In that case when we made the selection we would have chosen x instead.



Hypothetical shorter path (1)



Impossible shorter path (2)

(2) $x \in S$ before w => the shortest of all paths from the source to x runs through old S alone. Therefore, length(s ⇨ w ⇨ x)≥ length(s ⇨ x)

Another Form of Dijkstra's Algorithm

- Assume each vertex v in the graph stores two values, which describe a *tentative* shortest path from a source vertex, s, to v:
  - **dist(v)** – the length of the tentative shortest path from s to v.
  - **pred(v)** – the predecessor of v in the tentative shortest path from s to v.
- Predecessor pointers automatically define a tentative shortest path tree
- $\mathrm{dist}(s)=0$ and $\mathrm{pred}(s)=\mathrm{NIL}$.
- For every vertex $v \neq s$, we initially set $\mathrm{dist}(v)$ =1 and $\mathrm{pred}(v)=\mathrm{NIL}$ (do not know any path from s to v)
- We call an edge $u \rightarrow v$ *tense* if

$$\mathrm{dist}(u)+w(u \rightarrow v) < \mathrm{dist}(v)$$

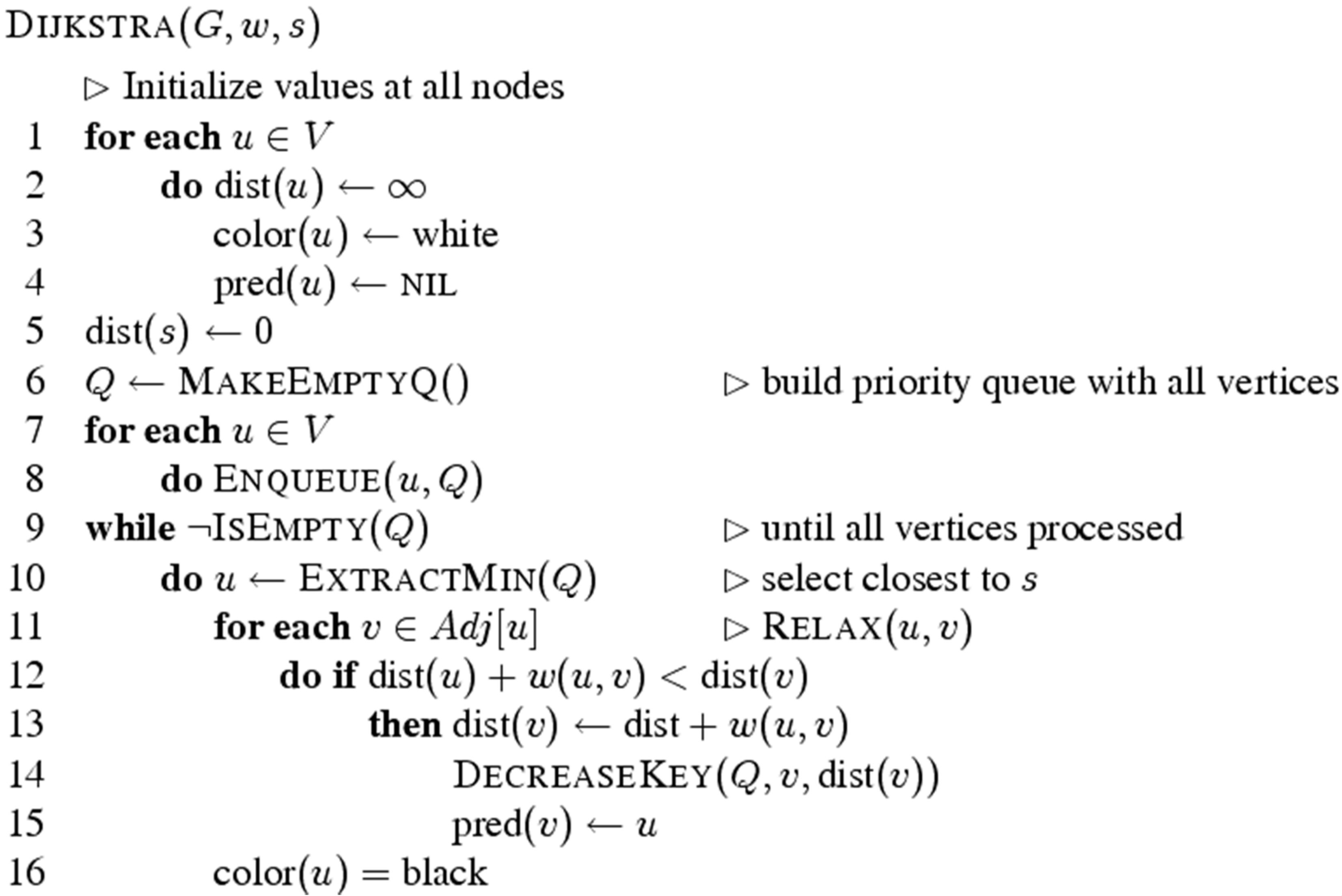
Another Form of Dijkstra's Algorithm

- Our generic algorithm repeatedly finds a tense edge and *relaxes* it:

```
RELAX($u \to v$)
  if $\mathrm{dist}(u) + w(u \to v) < \mathrm{dist}(v)$
      then $\mathrm{dist}(v) \leftarrow \mathrm{dist}(u) + w(u \to v)$
          $\mathrm{pred}(v) \leftarrow u$
```

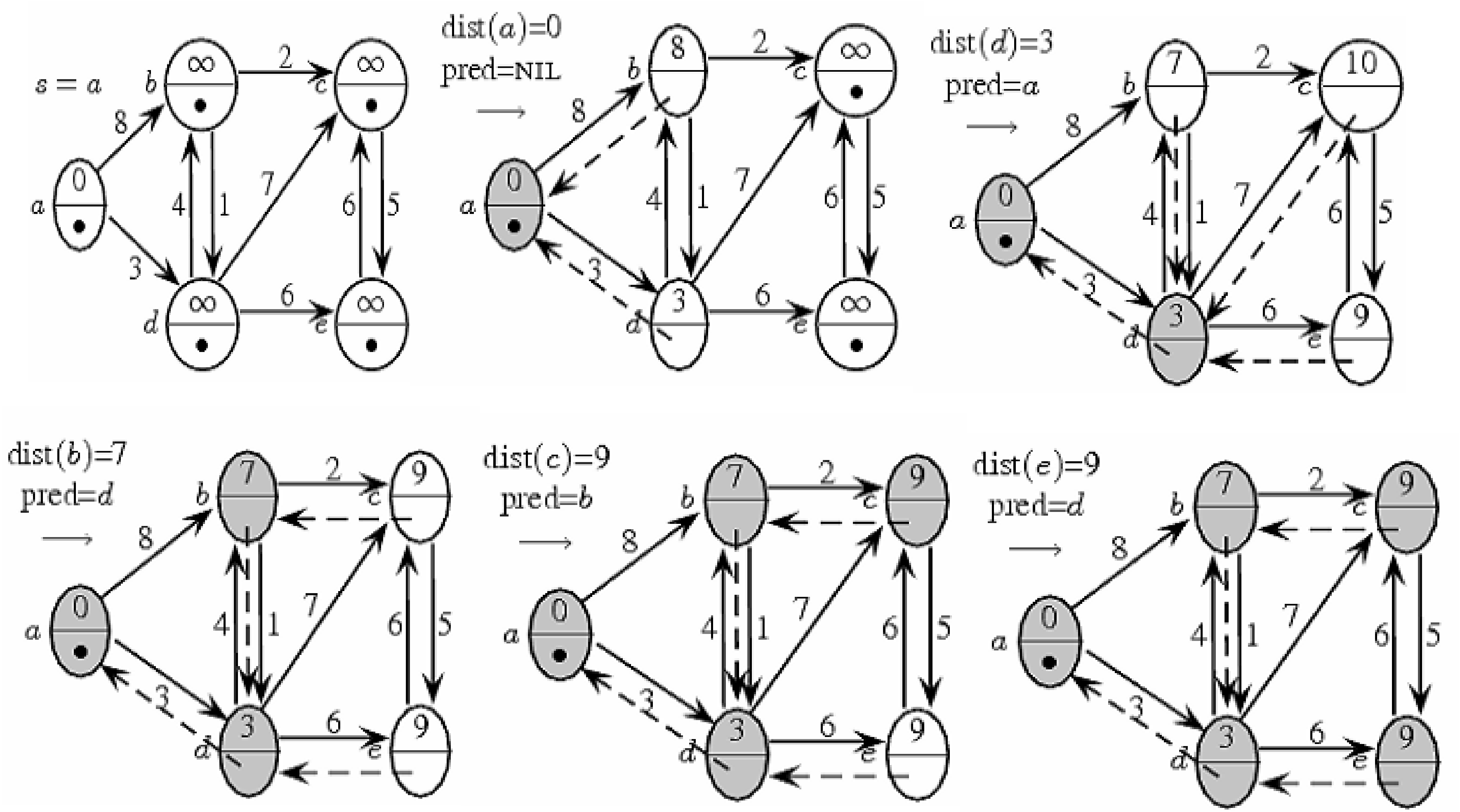
- Implementation
  - Uses a priority queue to hold vertices in $V \setminus S$
  - Needs cross references to vertices to allow for the *DecreaseKey* operation

Dijkstra’s algorithm (again)



```
DIJKSTRA($G, w, s$)
    ▷ Initialize values at all nodes
    for each $u \in V$
        do dist($u$) $\leftarrow \infty$
           color($u$) $\leftarrow$ white
           pred($u$) $\leftarrow$ NIL
    dist($s$) $\leftarrow 0$
    $Q \leftarrow$ MAKEEMPTYQ()                       ▷ build priority queue with all vertices
    for each $u \in V$
        do ENQUEUE($u, Q$)
    while ¬ISEMPTY($Q$)                               ▷ until all vertices processed
        do $u \leftarrow$ EXTRACTMIN($Q$)             ▷ select closest to $s$
           for each $v \in Adj[u]$                    ▷ RELAX($u, v$)
               do if dist($u$) $+ w(u, v) <$ dist($v$)
                     then dist($v$) $\leftarrow$ dist $+ w(u, v)$
                          DECREASEKEY($Q, v$, dist($v$))
                          pred($v$) $\leftarrow u$
           color($u$) $=$ black
```

Dijkstra's algorithm (again)



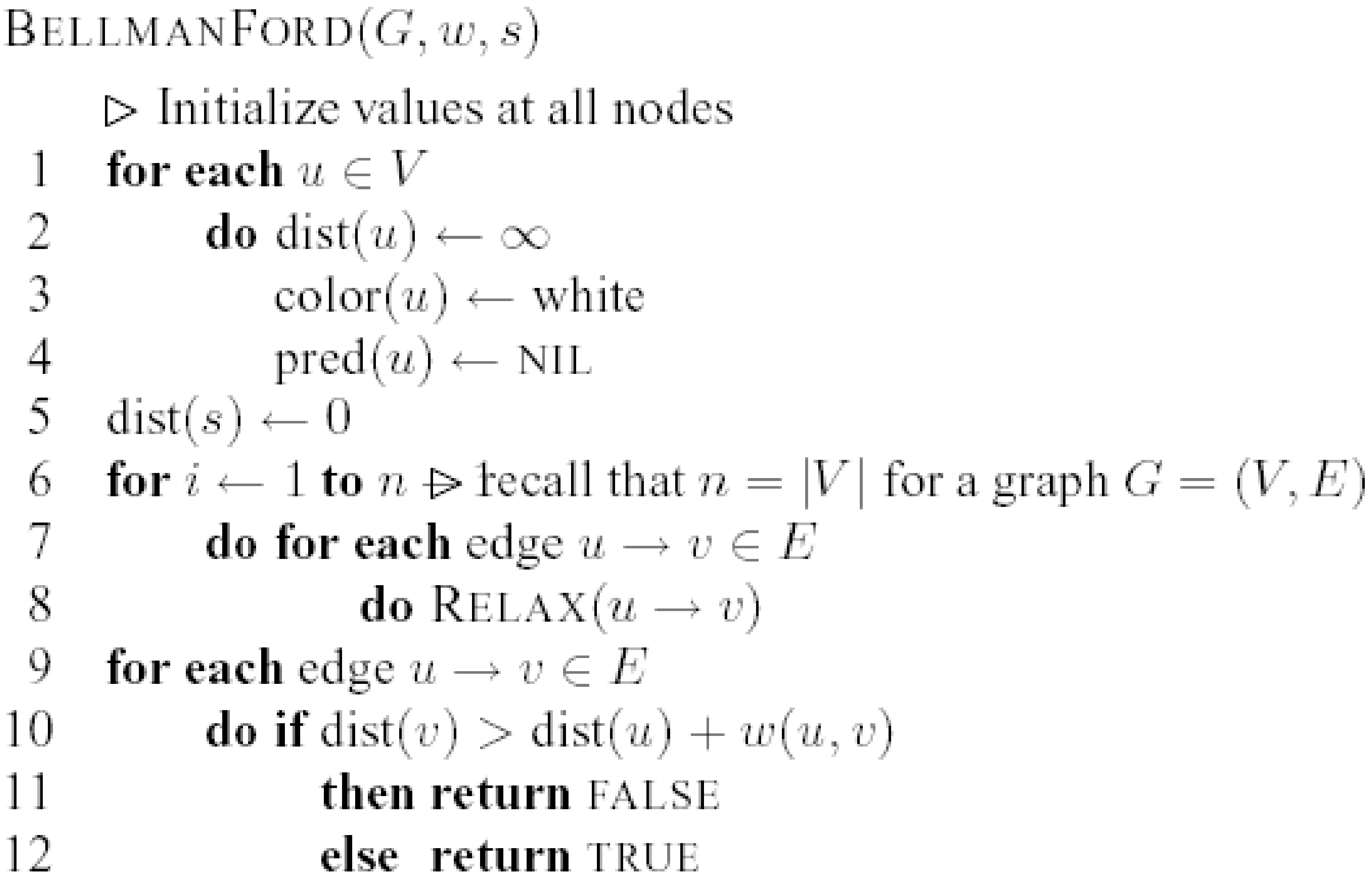
Animations for Dijkstra’s Algorithm

- http://www.cs.sunysb.edu/~skiena/combinatorica/animations/dijkstra.html
- http://www.cse.yorku.ca/~aaw/HFHuang/DijkstraStart.html
- http://www.cs.auckland.ac.nz/software/AlgAnim/dijkstra.html
- http://www.unf.edu/~wkloster/foundations/DijkstraApplet/DijkstraApplet.htm

Bellman-Ford(Moore)

- Moore in 1957, then independently by Bellman in 1958 -- Bellman used the idea of edge relaxation, first proposed by Ford in 1956
- able to deal with negative edge weights, but no negative cost cycles (otherwise we could make the path infinitely short by cycling forever through such a cycle)
- simply applies relaxation to *every* edge in the graph, and repeats this process n-1 times
- running time is $\Theta(ne)$:
    - initialization process $\Theta(n)$
    - n-1 passes through the set of edges: $\Theta(e)$

Bellman-Ford(Moore)



```
BELLMANFORD($G, w, s$)
    ▷ Initialize values at all nodes
    for each $u \in V$
        do dist($u$) $\leftarrow \infty$
           color($u$) $\leftarrow$ white
           pred($u$) $\leftarrow$ NIL
    dist($s$) $\leftarrow 0$
    for $i \leftarrow 1$ to $n$ ▷ recall that $n = |V|$ for a graph $G = (V, E)$
        do for each edge $u \rightarrow v \in E$
               do RELAX($u \rightarrow v$)
    for each edge $u \rightarrow v \in E$
        do if dist($v$) > dist($u$) + $w(u, v)$
              then return FALSE
              else return TRUE
```

Correctness of Bellman-Ford (Moore's) Algorithm

- any shortest path is a sequence

  $s \rightsquigarrow u : \langle v_0, v_1, \ldots v_k \rangle$, where $v_0 = s$ and $v_k = u$.
- a shortest path will never visit the same vertex twice (why?); path consists of at most $n - 1$ edges.
- true shortest path satisfies

$$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i)$$

- after the i^{th} pass of the loop 9–12

$$\text{dist}(v_i) = \delta(s, v_i)$$

- prior to pass i,

$$\text{dist}(v_{i-1}) = \delta(s, v_{i-1})$$

- after pass i, we have

$$\text{dist}(v_i) \leq \text{dist}(v_{i-1}) + w(v_{i-1}, v_i) = \delta(s, v_i)$$

Bellman-Ford (Moore's) Algorithm Animation

- http://www.laynetworks.com/Simulation%20of%20Bellman%20Algorithm.htm
- Local applet (Demos\BellManFord\BellmanFord.html)

Floyd-Warshall algorithm

- **The All-Pairs Shortest Paths Problem**

input for the algorithm is a $n \times n$ matrix W of edge weights

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) & \text{if } i \neq j \text{ and } i \rightarrow j \in E, \\ \infty & \text{if } i \neq j \text{ and } i \rightarrow j \notin E. \end{cases}$$

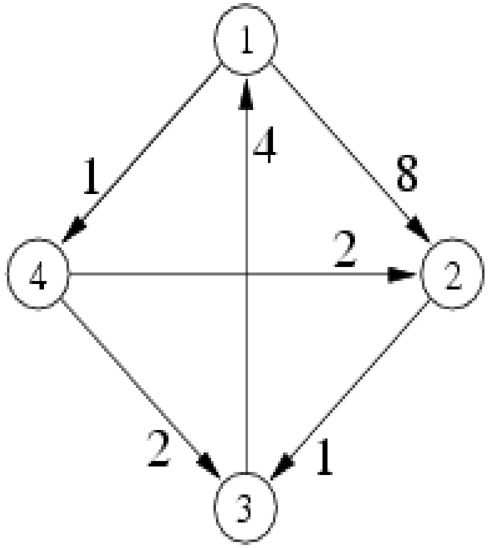
$p = \langle v_0, v_1, \ldots v_\ell \rangle$, vertices $v_1, v_3, \ldots v_{\ell-1}$ are *intermediate vertices.*

are chosen from the set $\{1, 2, \ldots, k\}$.

we consider a path $i \rightsquigarrow j$ which either

1. consists of the single edge $i \rightarrow j$, or
2. visits other vertices along the way, but only from the set $\{1, 2, \ldots, k\}$.

Floyd Warshall formulation



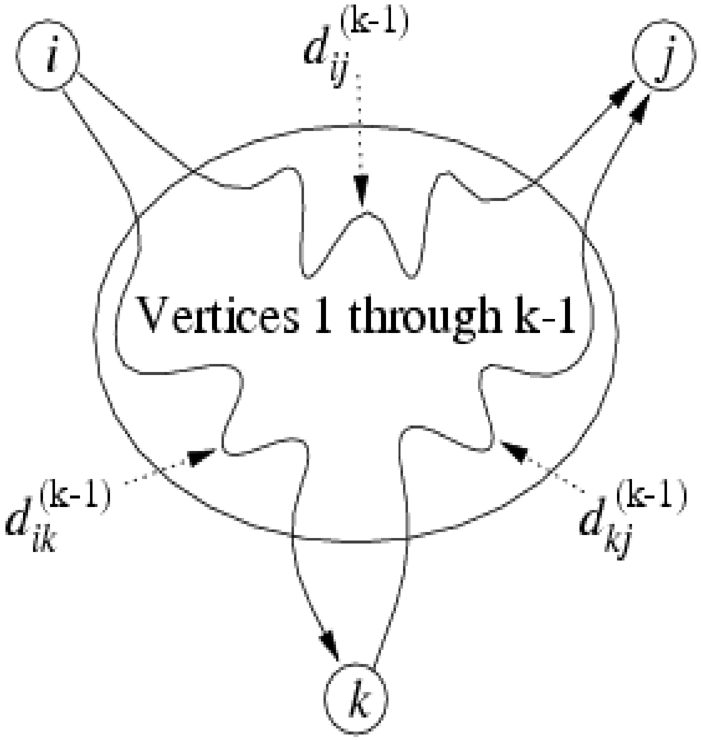
$d_{3,2}^{(0)} = \text{INF (no path)}$

$d_{3,2}^{(1)} = 12 \quad (3,1,2)$

$d_{3,2}^{(2)} = 12 \quad (3,1,2)$

$d_{3,2}^{(3)} = 12 \quad (3,1,2)$

$d_{3,2}^{(4)} = 7 \quad (3,1,4,2)$



Floyd-Warshall formulation

- Compute $d_{ij}^{(k)}$ knowing matrix at previous step $D^{(k-1)}$
- Two cases
  - Don't go through k. The shortest path passes only through intermediate vertices $\{1, 2, \ldots, k\text{-}1\}$ resulting in a path length of $d_{ij}^{(k-1)}$ (discovered at the previous step)
  - Do go through k. We can assume there is exactly one pass through vertex k (if there are no negative cost cycles):
    - go from i to k and from k to j
    - to get an overall minimum, take the shortest path $i \Rightarrow k$, and the shortest path $k \Rightarrow j$.
    - this path uses only the intermediate vertices in $\{1, 2, \ldots, k\text{-}1\}$, the length of the path is $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

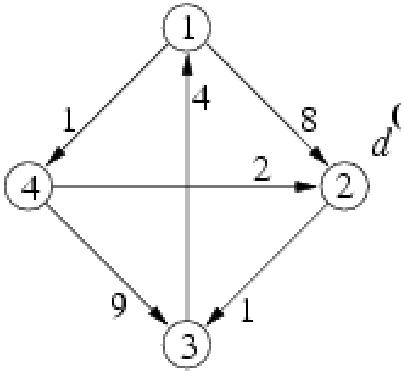
Floyd-Warshall

$$d_{ij}^{(0)} = w_{ij} \quad \text{for } k = 0$$
$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \quad \text{for } k \geq 1$$

```
PATH($i, j$)
    if $mid_{ij}$ = NIL
        then LIST($i, j$)
        else PATH($i, mid_{ij}$)
             PATH($mid_{ij}, j$)
```

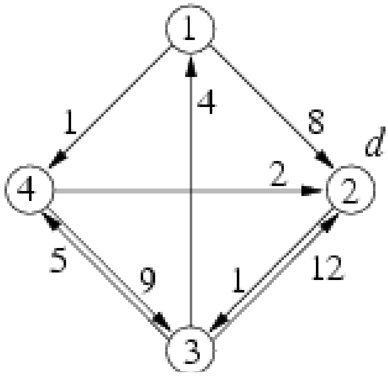
```
FLOYDWARSHALL($n, W$)
    $D^{(0)} \leftarrow W$
    for $k \leftarrow 1$ to $n$                                   ▷ use intermediates $\{1, 2, \ldots, k\}$
        do for $i \leftarrow 1$ to $n$                           ▷ ...from $i$
            do for $j \leftarrow 1$ to $n$                       ▷ ...to $j$
                do if $d_{ik}^{(k-1)} + d_{kj}^{(k-1)} < d_{ij}^{(k-1)}$
                    then $d_{ij}^{(k)} \leftarrow d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$
                         $mid_{ij} \leftarrow k$                 ▷ new shorter path length is through $k$
                    else $mid_{ij} \leftarrow$ NIL
    return $D^{(n)}$                                              ▷ Matrix of final distances
```

Floyd-Warshall trace

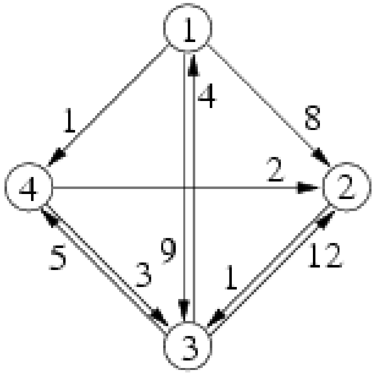


$$d^{(0)} = \begin{bmatrix} 0 & 8 & ? & 1 \\ ? & 0 & 1 & ? \\ 4 & ? & 0 & ? \\ ? & 2 & 9 & 0 \end{bmatrix}$$

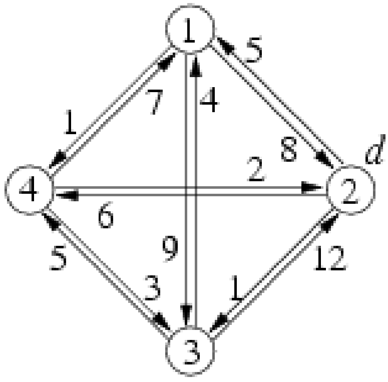
? = infinity



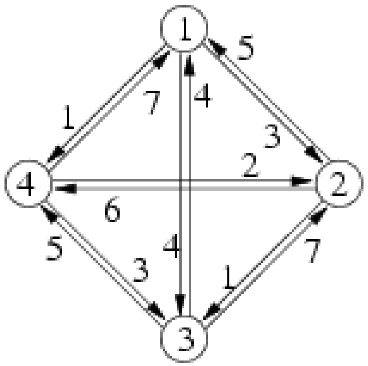
$$d^{(1)} = \begin{bmatrix} 0 & 8 & ? & 1 \\ ? & 0 & 1 & ? \\ 4 & 12 & 0 & 5 \\ ? & 2 & 9 & 0 \end{bmatrix}$$



$$d^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ ? & 0 & 1 & ? \\ 4 & 12 & 0 & 5 \\ ? & 2 & 3 & 0 \end{bmatrix}$$



$$d^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$



$$d^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

Floyd-Warshall Algorithm Animation

- http://students.ceid.upatras.gr/~papagel/project/kef5_7_2.htm
- http://www.cs.man.ac.uk/~graham/cs2022/dynamic/floydwarshall/
- http://www.pms.ifi.lmu.de/lehre/compgeometry/Gosper/shortest_path/shortest_path.html#visualization

Transitive closure

- Graph *G*=(*V*, *E*). Transitive closure: graph *G**=(*V*,*E**)

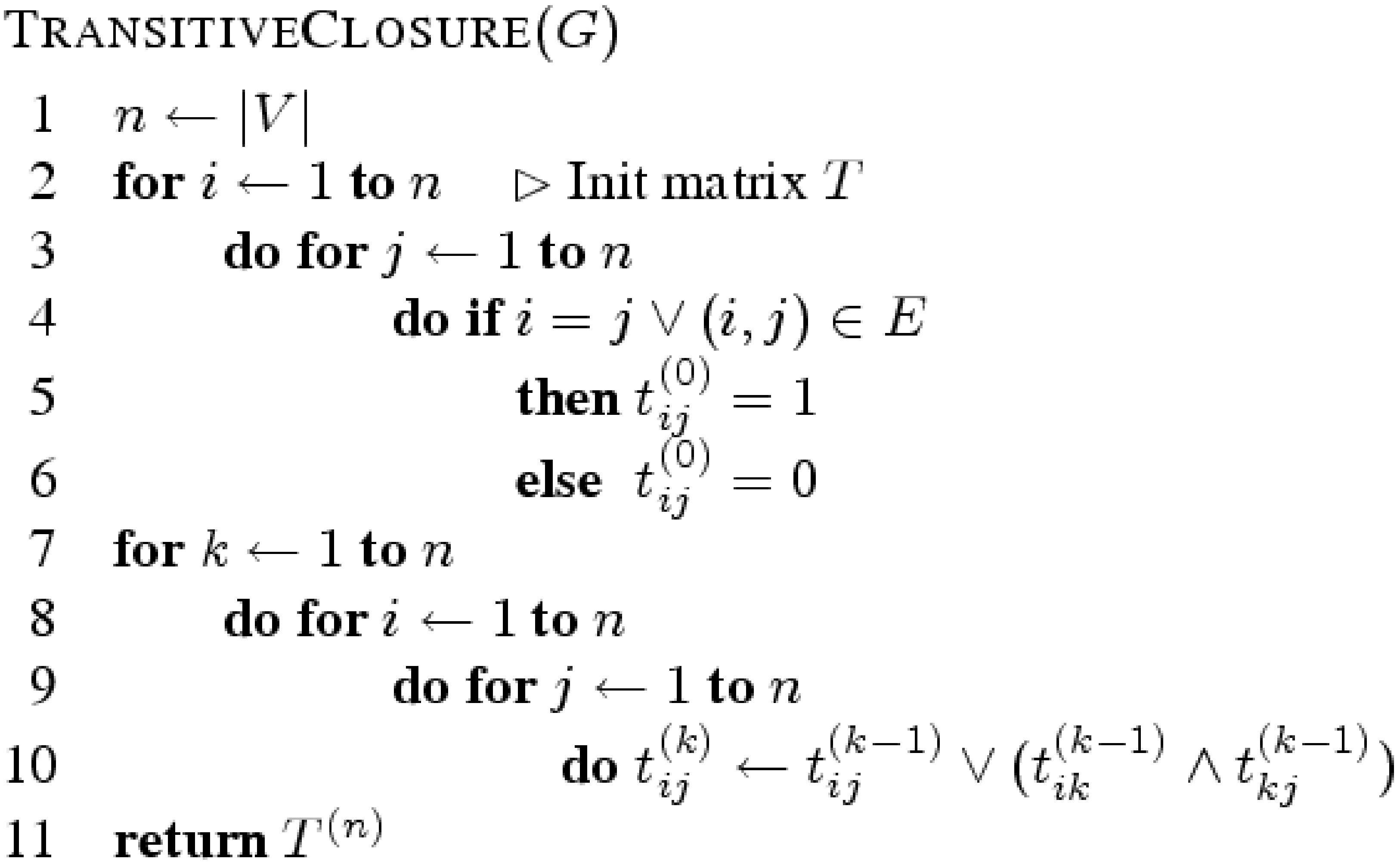
$$E^* = \{(i,j) \bullet \ \textit{there exists a path } i \leadsto j \textit{ in } G\}$$

- Solutions:
  - Floyd Warshall with weights 1
  - or use recurrence:

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \end{cases}$$

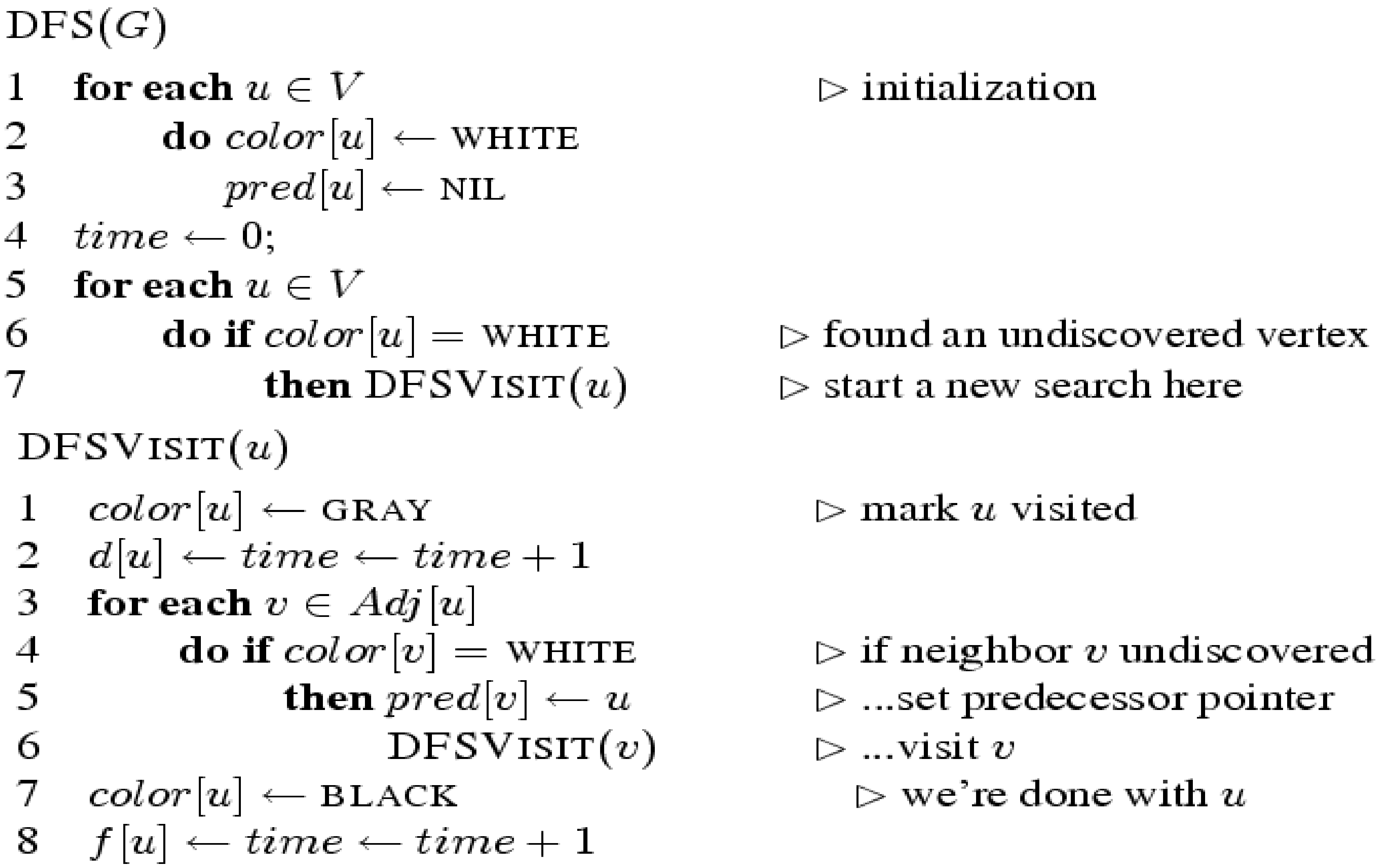
$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}) \text{ for } k \geq 1.$$

Transitive closure



```
TransitiveClosure($G$)
    $n \leftarrow |V|$
    for $i \leftarrow 1$ to $n$        ▷ Init matrix $T$
        do for $j \leftarrow 1$ to $n$
            do if $i = j \vee (i, j) \in E$
                then $t_{ij}^{(0)} = 1$
                else $t_{ij}^{(0)} = 0$
    for $k \leftarrow 1$ to $n$
        do for $i \leftarrow 1$ to $n$
            do for $j \leftarrow 1$ to $n$
                do $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$
    return $T^{(n)}$
```

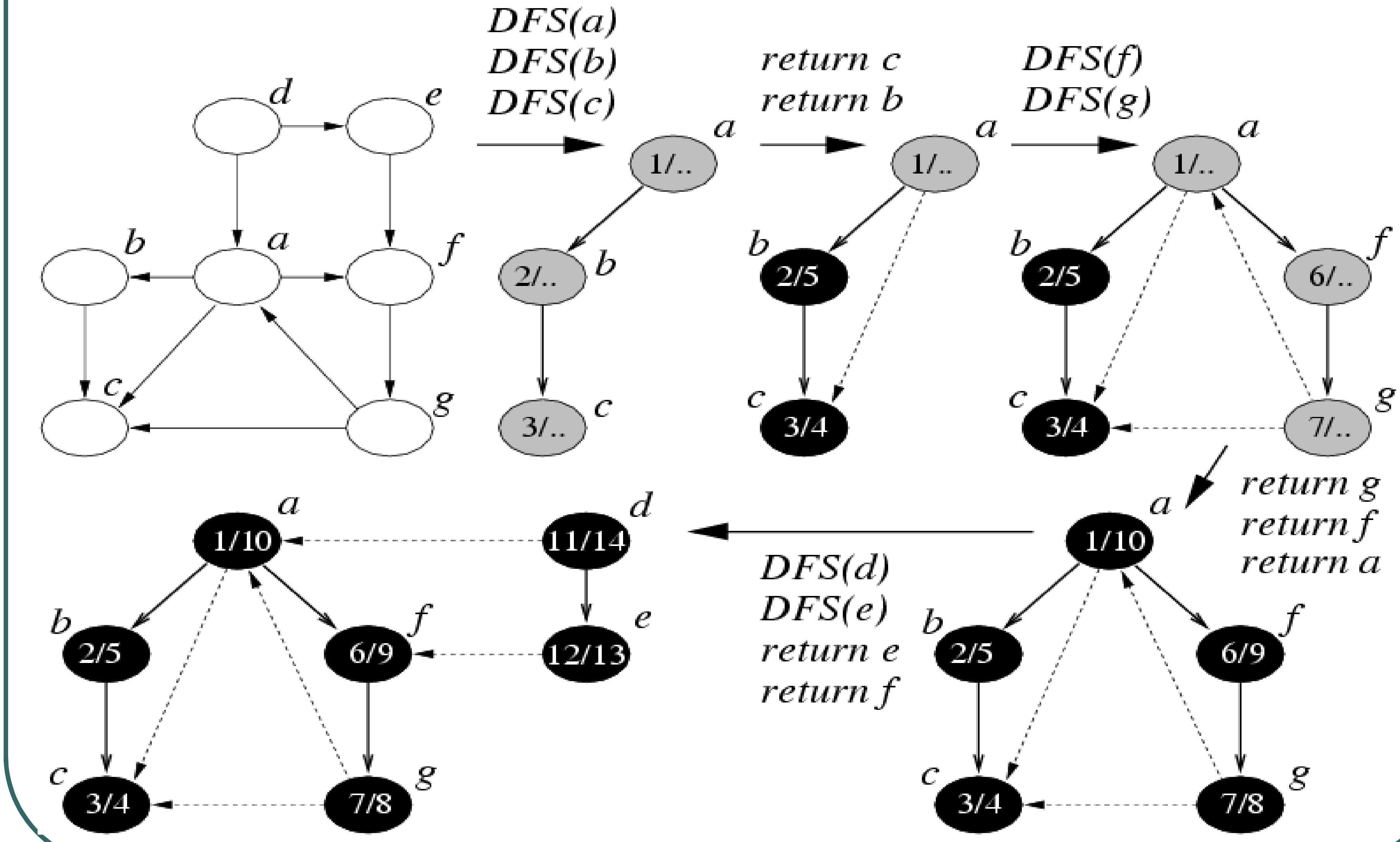
Traversals. Depth first search



```
DFS($G$)
for each $u \in V$                                  ▷ initialization
    do $color[u] \leftarrow$ WHITE
       $pred[u] \leftarrow$ NIL
$time \leftarrow 0$;
for each $u \in V$
    do if $color[u] =$ WHITE                        ▷ found an undiscovered vertex
          then DFSVisit($u$)                        ▷ start a new search here
```

```
DFSVisit($u$)
$color[u] \leftarrow$ GRAY                          ▷ mark $u$ visited
$d[u] \leftarrow time \leftarrow time + 1$
for each $v \in Adj[u]$
    do if $color[v] =$ WHITE                        ▷ if neighbor $v$ undiscovered
          then $pred[v] \leftarrow u$               ▷ ...set predecessor pointer
               DFSVisit($v$)                        ▷ ...visit $v$
$color[u] \leftarrow$ BLACK                         ▷ we're done with $u$
$f[u] \leftarrow time \leftarrow time + 1$
```

Depth first search example



Analysis of DFS

- If we ignore the time spent in the recursive calls DFS takes $O(n)$ time
- Each vertex is visited exactly once in the search=>call to DFSVisit is made exactly once for each vertex
- Let $outdeg(u)$ be the number of outgoing arcs from u
- Each vertex *u* can be processed in $O(1 + outdeg(u))$ time
- Total time is

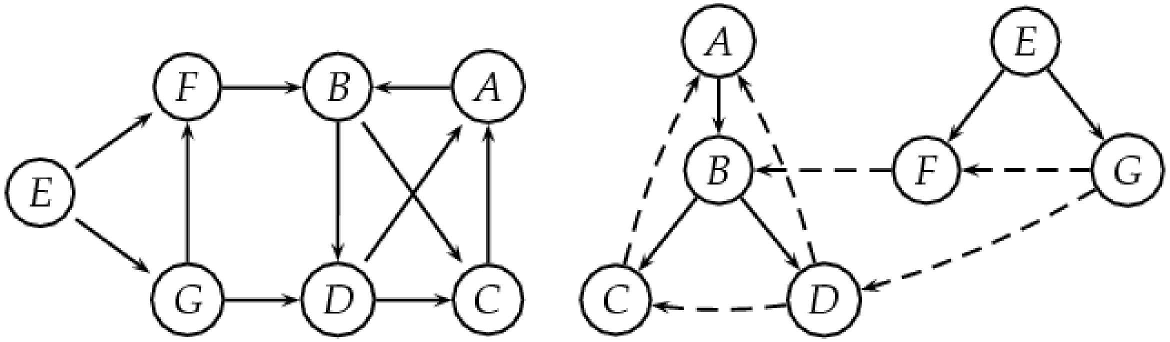
$$T(n) = n + \sum_{u \in V} (outdeg(u) + 1) = n + \sum_{u \in V} outdeg(u) + n = 2n + e \in \Theta(n + e)$$

DFS Animations

- http://maven.smith.edu/~thiebaut/java/graph/Welcome.html
- http://sziami.cs.bme.hu/~gsala/alg_anims/3/graph-e.html

DFS edges classification

- Edge classification by DFS
  - Tree edges: lead to unvisited vertices
  - Back edges: (u, v) – v is a (not necessarily proper) ancestor of u in the tree. (A self-loop is considered to be a back edge).
  - Forward edges: (u, v) – v is a proper descendent of u in the tree.
  - Cross edges: (u, v) – u and v are neither ancestors nor descendants of one another (the edge may go between different trees of the forest)

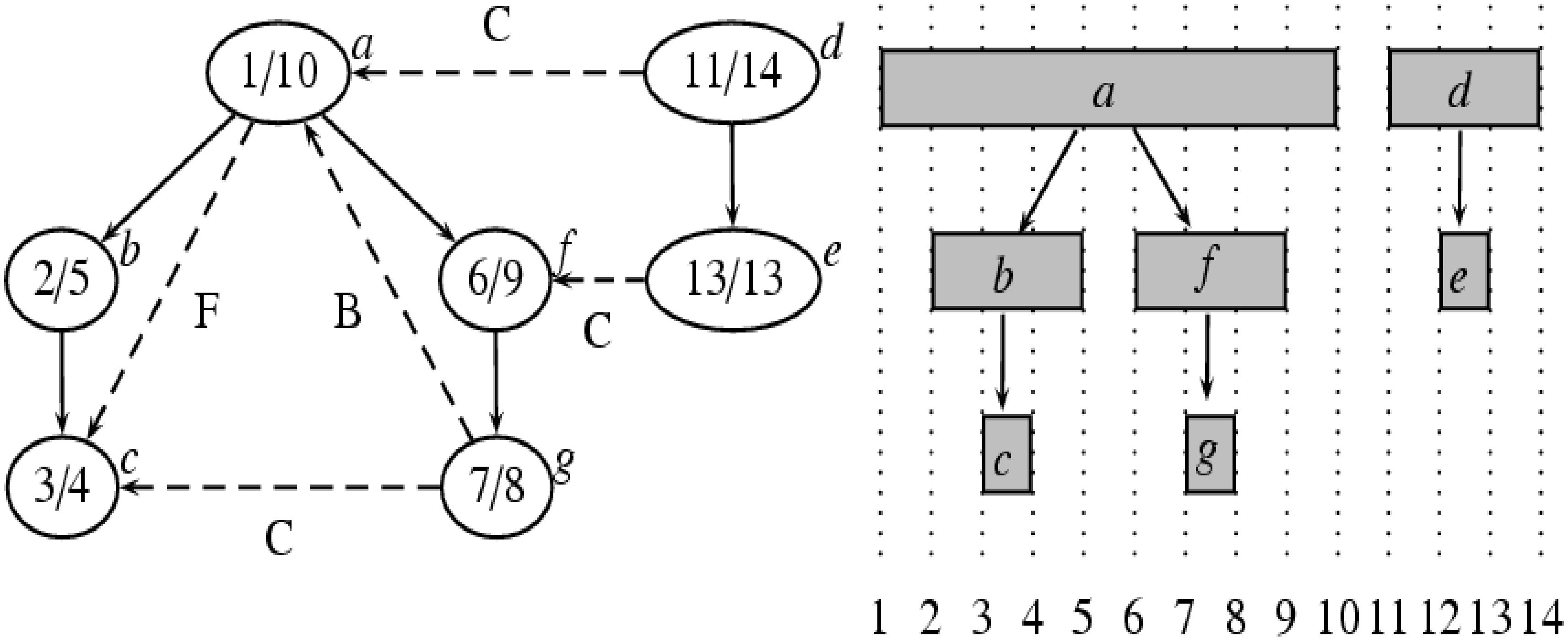


(a) A directed graph. (b) Depth-first spanning forest

Parenthesis Lemma

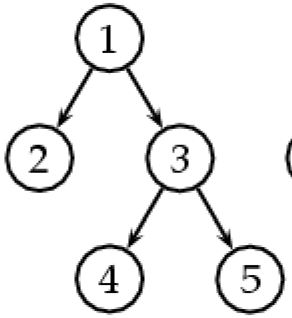
- Parenthesis Lemma. Given a digraph $G = (V, E)$, and any DFS tree for G and any two vertices $u, v \in V$, exactly one of the following three conditions hold:
  - u is a descendent of v if and only if $[d[u], f\,[u]] \subseteq [d[v], f\,[v]]$.
  - u is an ancestor of v if and only if $[d[u], f\,[u]] \supseteq [d[v], f\,[v]]$.
  - u is unrelated to v if and only $[d[u], f\,[u]] \cap [d[v], f\,[v]]. =\emptyset$.

Illustration of parenthesis lemma

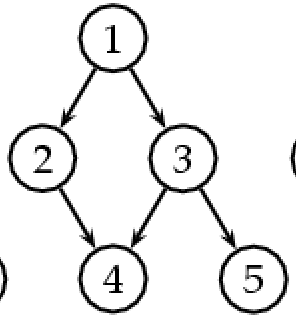


Directed acyclic graphs (DAGs)

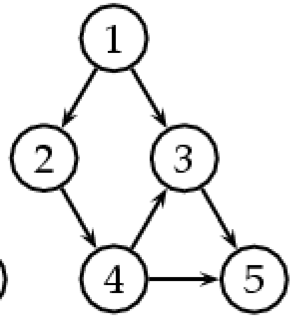
- Directed acyclic graph (dag) = directed graph with no cycles
  - more general than trees but less general than arbitrary digraphs
  - also useful in representing arithmetic expressions, partial orders,…



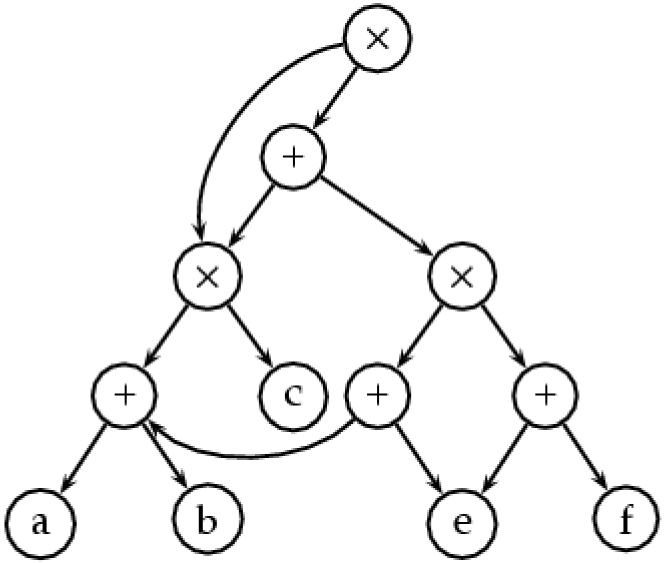
(tree)



(dag)



(directed graph with a cycle)

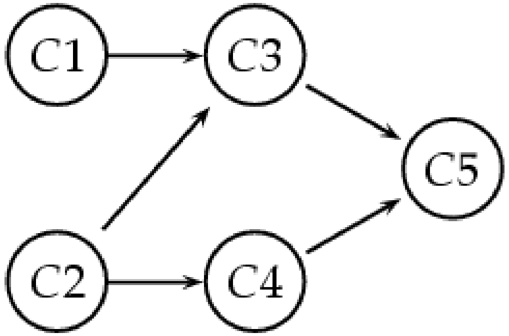


Test for acyclicity

- Can use DFS for testing
- If a directed graph has a cycle, then a back arc will always be encountered in any depth-first search of the graph
- Proof
  - suppose *G* is cyclic; do a DFS
  - there will be one vertex *v* having the lowest depth-first search number (*d*[..]) of any vertex on a cycle
  - consider an arc *u*→*v* on some cycle containing *v*
  - *u* is on the cycle, *u* must be a descendant of *v* in DFS; *u*→*v* cannot be a cross arc
  - d[*u*] > d[*v*], *u*→*v* cannot be a tree arc or a forward arc. It's a back arc

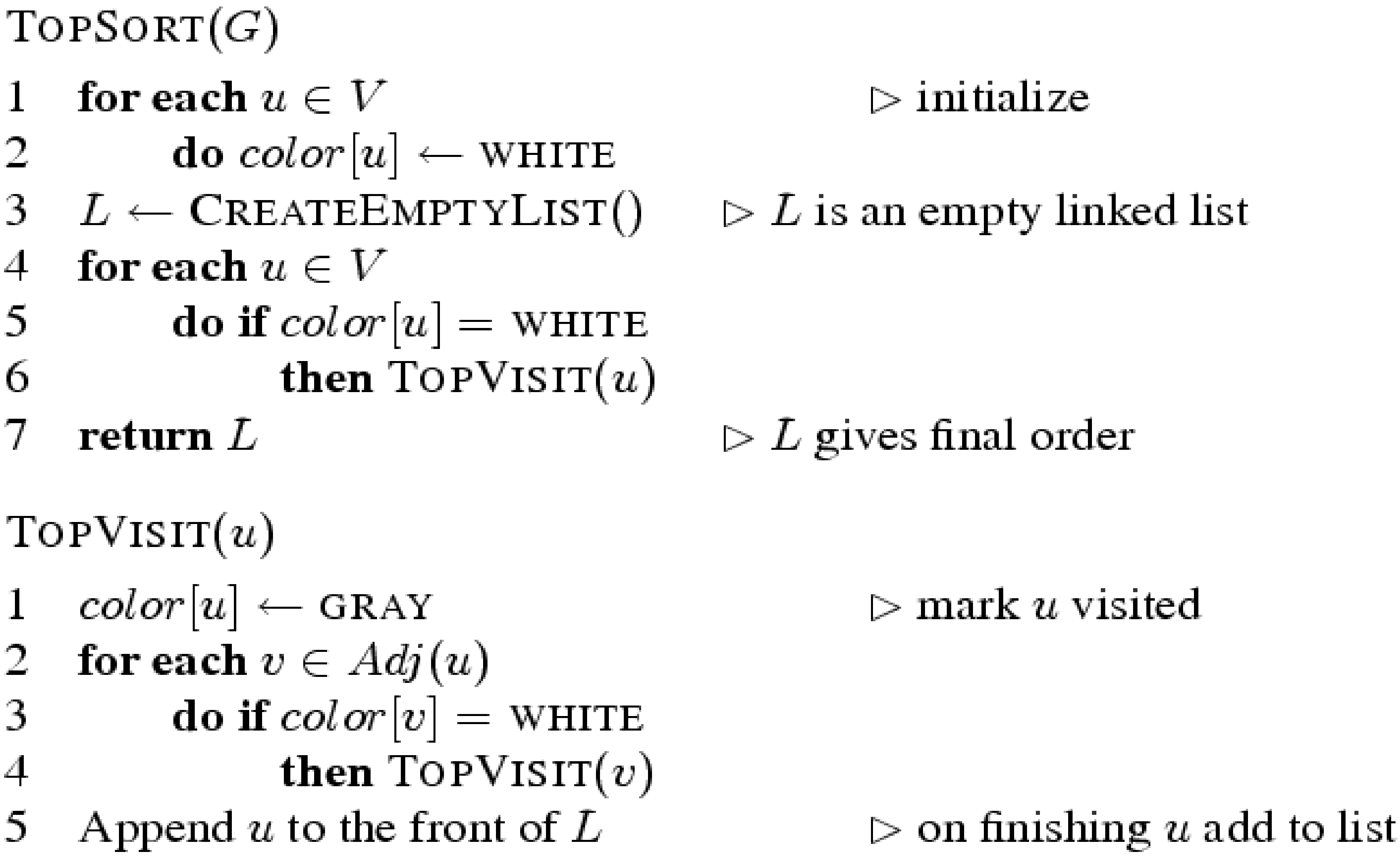
Topological sort

- Situations where activities must be completed in a certain order
  - Use dags to represent this



- Topological sort: a process of assigning a linear ordering to the vertices of a dag so that if there is an arc *i*→*j*, then *i* appears before *j* in the linear ordering
  - in general, there may be many legal topological orders for a given DAG

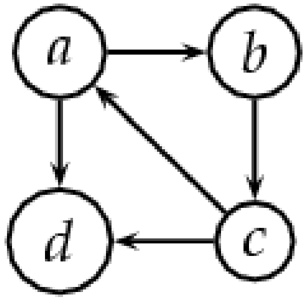
Topological sort



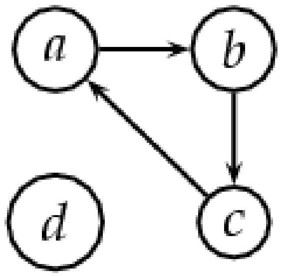
```
TopSort($G$)
for each $u \in V$                                 ▷ initialize
    do $color[u] \leftarrow$ WHITE
$L \leftarrow$ CreateEmptyList()                   ▷ $L$ is an empty linked list
for each $u \in V$
    do if $color[u] =$ WHITE
        then TopVisit($u$)
return $L$                                         ▷ $L$ gives final order

TopVisit($u$)
$color[u] \leftarrow$ GRAY                         ▷ mark $u$ visited
for each $v \in Adj(u)$
    do if $color[v] =$ WHITE
        then TopVisit($v$)
Append $u$ to the front of $L$                     ▷ on finishing $u$ add to list
```

Strong Components

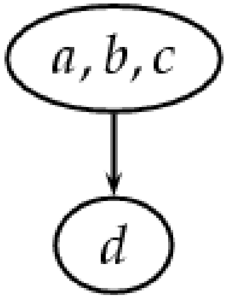
- Strongly connected component of a digraph: a maximal set of vertices in which there is a path from any one vertex in the set to any other vertex in the set.
  - Can use dfs to determine strong components



A graph



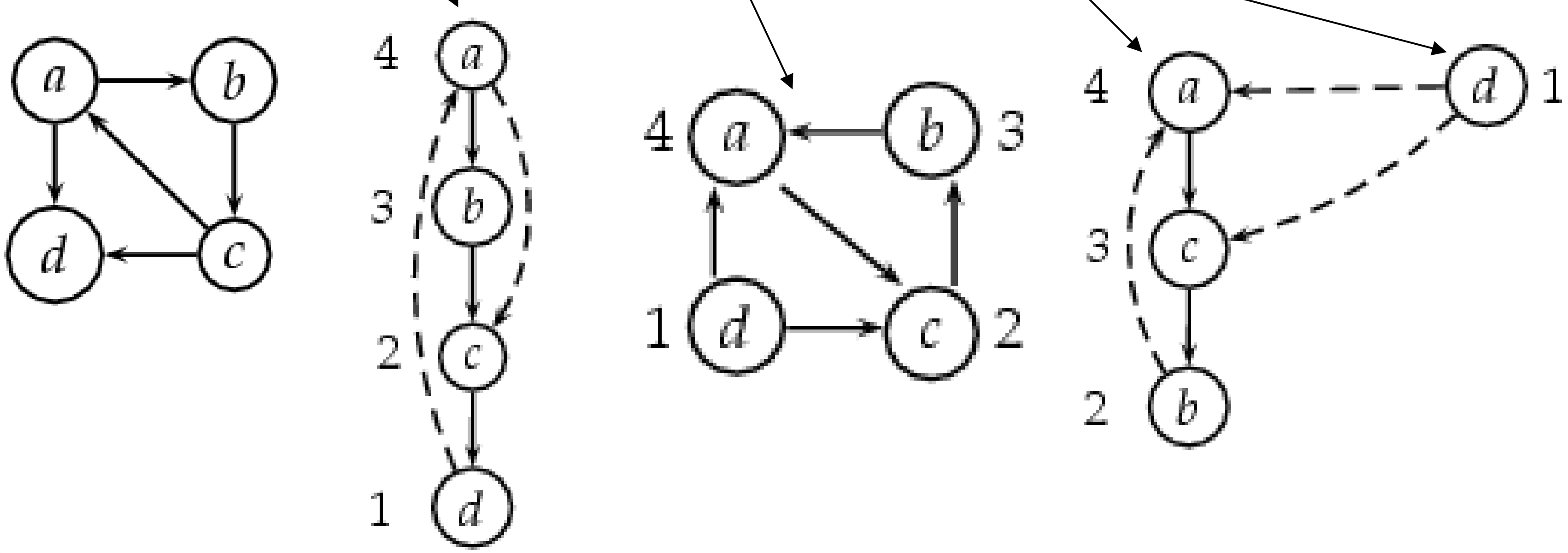
Strong components



Reduced graph. Represents connections between components

Determining strong components

```
StrongComponents($G(V, E)$)
 Run $dfs(G)$ computing finish times $f[u]$
 for each vertex $u$
 do Compute $R \leftarrow reverse(G)$, reversing all edges of $G$
       Sort vertices of $R$ decreasing by $f[u]$
       Run $dfs(R)$ using this order Each $dfs$ tree is a strong component.
```



Reading

- AHU, chapter 6
- Preiss, chapter: Graphs and Graph Algorithms
- CLR, chapter 23, sections 3-5, chapter 25, sections 1-3, chapter 26, sections 1-3
- CLRS chapter 22, section 1, 3, chapter 24 sections 1-3, chapter 25 section 2
- Notes