

Graph Representation and Traversal

Chapter 7

2015

Representation of graphs

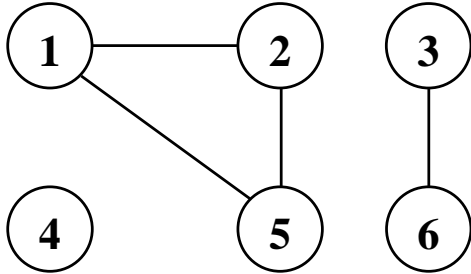
- A *directed graph* or *digraph* is a pair $G = (V, E)$ s.t.:
 - V is a finite set called the *set of vertices* of G .
 - $E \subseteq V \times V$ is a binary relation on V called the *set of arcs* of G . An *arc* of G is denoted by an ordered pair of vertices (u, v) , $u, v \in V$. Note that $(u, v) \neq (v, u)$.
- An *undirected graph* is a pair $G = (V, E)$ s.t.:
 - V is a finite set called the *set of vertices* of G .
 - E is a set of unordered pairs of vertices $\{u, v\}$, $u, v \in V$ called the *edges* of G . For uniformity we denote an edge with (u, v) , but in an undirected graph $(u, v) = (v, u)$.
- There are two basic techniques for representing graphs in the computer memory.
 - Adjacency matrix
 - Adjacency lists

2015

Adjacency matrix

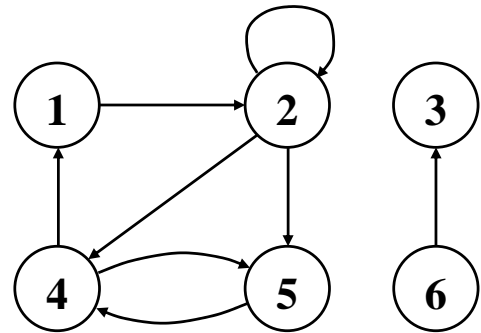
- Let $G = (V, E)$ be a graph with the vertices labeled $1, 2, \dots, |V|$. The *adjacency matrix* of G is a matrix A of size $|V| \times |V|$ defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{if } (i, j) \notin E \end{cases}$$



An undirected graph G_1

$$A_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$



a. A directed graph G_2

$$A_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

2015

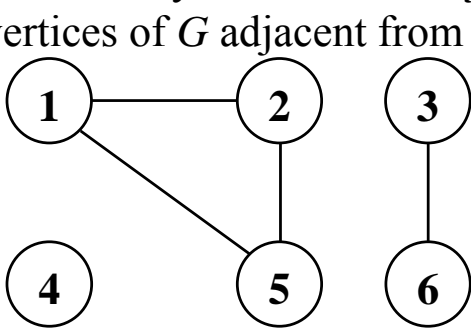
Adjacency matrix – properties

- The memory consumption for an adjacency matrix is $\Theta(|V|^2)$. Note that it depends only on the number of graph vertices.
- Property:** the adjacency matrix for an undirected graph is *symmetric* because (u,v) and (v,u) represent the same edge. This observation allows us to reduce the memory space required for representing an undirected graph with 50 %. The representation can be made even more efficient if we are using a single bit for representing a matrix element.
- Advantage:** it is recommended when the graph is *dense* and the operation for checking if two vertices are connected by an edge must be done fast. For adjacency matrix representation this operation takes $\Theta(1)$ time.
- Disadvantage:** it is inefficient when the graph is *sparse*, i.e. it contains only a few edges (the percent of vertices connected by an edge from the set of all pairs of vertices is small).
- The representation can be adapted for graphs with costs attached to edges. In this case a_{ij} will be equal with the cost of the edge (i,j) , if such an edge exists or a value that can't be a cost (NIL , 0 or ∞) if there is no such edge.

2015

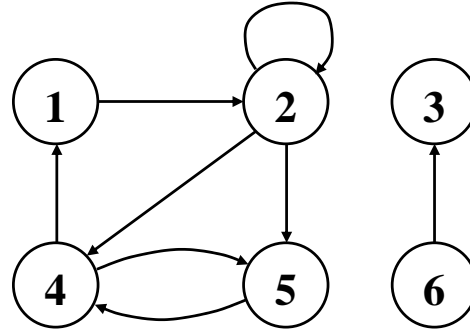
Adjacency lists

- Let $G = (V, E)$ be a graph with the vertices labeled $1, 2, \dots, |V|$. The *adjacency lists* of G are stored in an array Adj of size $|V|$. An element $Adj[u]$ of this array is called the adjacency list of vertex u and it contains all the vertices of G adjacent from u .



An undirected graph G_1

$$A_1 = \begin{bmatrix} (2, 5) \\ (1, 5) \\ (6) \\ () \\ (1, 2) \\ (3) \end{bmatrix}$$



a. A directed graph G_2

$$A_2 = \begin{bmatrix} (2) \\ (2, 4, 5) \\ () \\ (1, 5) \\ (4) \\ (3) \end{bmatrix}$$

2015

Adjacency lists – properties

- The array Adj can be represented as an array of pointers with each element $Adj[u]$ pointing to the first element of the adjacency list of u . The adjacency list of u can be represented as a linked list.
- The memory consumption for adjacency lists is $\Theta(|V|+|E|)$.
- Property: summing the lengths of the adjacency lists we get $2|E|$ for an undirected graph and $|E|$ for a directed graph (why?).
- Advantage: the memory consumption is less than for the adjacency matrix. The representation is efficient even for sparse graphs.
- Disadvantage: checking if two vertices u and v are connected by an edge involves searching for v in the adjacency list of u . The operation takes $O(|Adj(u)|)$ time.
- The representation can be adapted for graphs with costs on edges. The cost of the edge (u,v) is simply added as a new field of the element of the adjacency list $Adj[u]$ that corresponds to v .

2015

Graph search fundamentals

- *Graph searching* = systematic exploration of graph vertices such that each vertex accessible (or reachable) from a given vertex s is visited exactly once. Graph searching is a basis for many graph algorithms.
- There are two basic techniques for searching graphs that differ in the order they visit the graph vertices:
 - *breadth first*
 - *depth first*

2015

Breadth first search fundamentals

- Breadth first search:
 - The search starts with visiting the source vertex s .
 - Then it visits all the vertices accessible from s in increasing order of their distance (in number of edges) from s , i.e. first the vertices of distance 1 (the neighbors of s), next the vertices of distance 2, ...
- The BFS algorithm (see the next slide) builds a *breadth first tree* rooted at the source vertex s that contains all the vertices accessible from s .
- For each vertex u it computes $\pi[u]$, the parent vertex of u in this tree. Obviously, for the source vertex s we have $\pi[s] = \emptyset$.
- Also the BFS algorithm computes for each vertex u its depth $d[u]$ in this tree. $d[u]$ is the minimum distance from s to u .

2015

Breadth first search algorithm

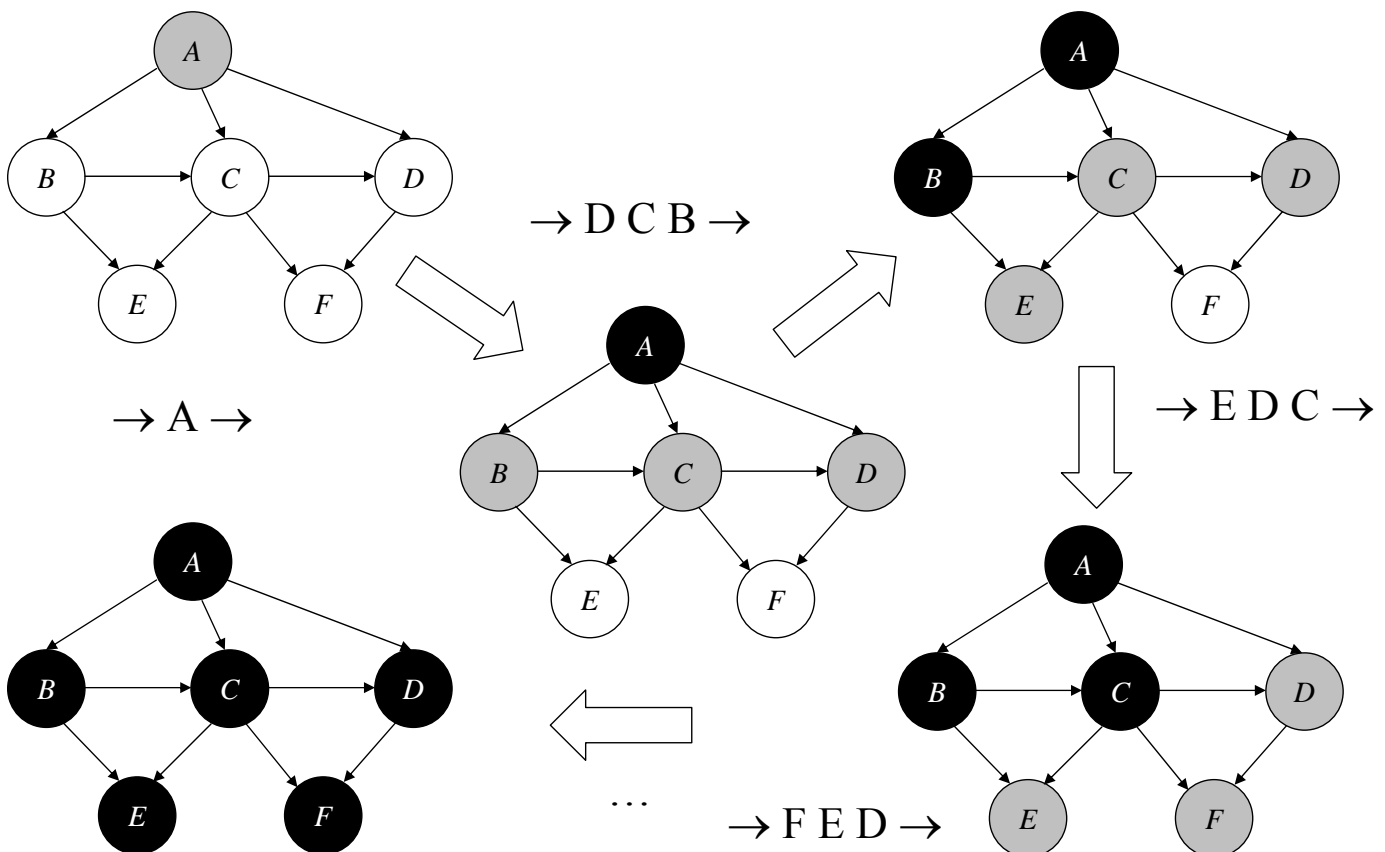
BFS(G, s)

1. **for** each vertex $u \in V[G] \setminus \{s\}$ **do**
2. $color[u] \leftarrow WHITE$
3. $d[u] \leftarrow \infty$
4. $\pi[u] \leftarrow NIL$
5. $color[s] \leftarrow GRAY$
6. $d[s] \leftarrow 0$
7. $\pi[s] \leftarrow NIL$
8. $Q \leftarrow \{s\}$
9. **while** \neg QUEUE-EMPTY(Q) **do**
10. $u \leftarrow$ QUEUE-FRONT(Q)
11. **for** each vertex $v \in Adj[u]$ **do**
12. **if** $color[v] = WHITE$ **then**
13. $color[v] \leftarrow GRAY$
14. $d[v] \leftarrow d[u] + 1$
15. $\pi[v] \leftarrow u$
16. QUEUE-PUSH(Q, v)
17. QUEUE-POP(Q)
18. $color[u] \leftarrow BLACK$

- A vertex that has not been reached yet is colored *white*.
- Lines 1 – 4 perform some initialization, including coloring all the vertices with white.
- A vertex that has been reached becomes *gray* and stays gray until all his neighbors are reached, when it becomes *black*. The gray vertices are stored in queue Q .
- Lines 5 – 8 start visiting s by initializing the structures $color$, π , d and Q .
- Lines 9 – 18 contain the main program loop. Line 10 picks up the vertex u in front of queue Q .
- The for loop (lines 11 – 16) finds all the white vertices v in the neighborhood of u , prepares them for being visited (by graying them), updates π and d accordingly and pushes them on Q .
- Finally u is popped from Q and colored black (lines 17 – 18).

2015

Breadth first search example



2015

Classification of vertices and arcs

- Vertices:
 - Unexplored (white)
 - Visited (gray and black)
- Arcs:
 - Unexplored
 - Visited
- Observation:
 - A visited vertex for which all the arcs incident out of it are visited is black; otherwise it is gray.

2015

Breadth first search analysis

- Each vertex is made white only once and therefore popped from the queue only once.
- The operations of $\text{QUEUE-PUSH}(Q, v)$ and $\text{QUEUE-POP}(Q)$ take $O(1)$ time.
- The total time for queue operations is $O(|V|)$.
- The adjacency list for each vertex is scanned at most once.
- The sum of the lengths of all adjacency lists is $\Theta(|E|)$.
- At most $O(|E|)$ time is spent scanning adjacency lists.
- Initialization overhead is $O(|V|)$.
- We conclude that the total runtime for BFS is $O(|V| + |E|)$, i.e. linear time in the number of vertices plus the number of edges of the graph.

2015

Shortest paths

- We denote the shortest path distance $\delta(s,v)$ as the minimum number of edges in any path from vertex s to vertex v . BFS computes the shortest-path distance from s to v , i.e. $d[v] = \delta(s,v)$ for each vertex $v \in V$.
- Theorem. Let $G = (V,E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s , and upon termination, $d[v] = \delta(s,v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s , one of the shortest paths s to v is the shortest path from s to $\pi[v]$ followed by the edge $(\pi[v],v)$ (see the textbook for the proof).
- This theorem states the following three important things:
 - BFS discovers every vertex that is reachable from the source vertex.
 - Upon termination of the BFS, it computes the shortest path distance from the source node to every reachable vertex.
 - Upon termination of the BFS we can compute a shortest path from the source vertex to every reachable vertex using the parent links of the breadth first tree that were saved in π .

2015

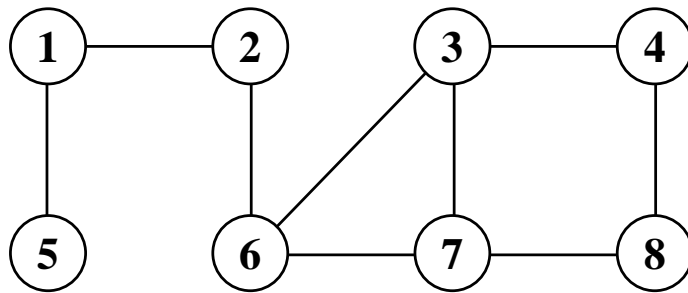
Breadth first trees

- The structure π built during the breadth first search of a graph $G=(V,E)$ constructs a *predecessor sub-graph* $G_\pi=(V_\pi,E_\pi)$ defined as:
$$V_\pi = \{v \in V \mid \pi[v] \neq \text{NIL}\} \cup \{s\}$$
$$E_\pi = \{(\pi[v],v) \in E \mid v \in V_\pi \setminus \{s\}\}$$
- G_π is called a *breadth first tree* of G if the set V_π contains all the vertices accessible from s and for every vertex $v \in V$ there is a unique path from s to v that is also a shortest path from s to v . It is easy to note that a breadth first tree is a tree because it is connected and $|E_\pi| = |V_\pi| - 1$.
- Property: the predecessor sub-graph constructed by the BFS algorithm is a breadth first tree of G .
- It is easy to devise an algorithm that, given π and a vertex v , it prints a shortest path from the source vertex s to v . The algorithm will use the parent links stored in π .

2015

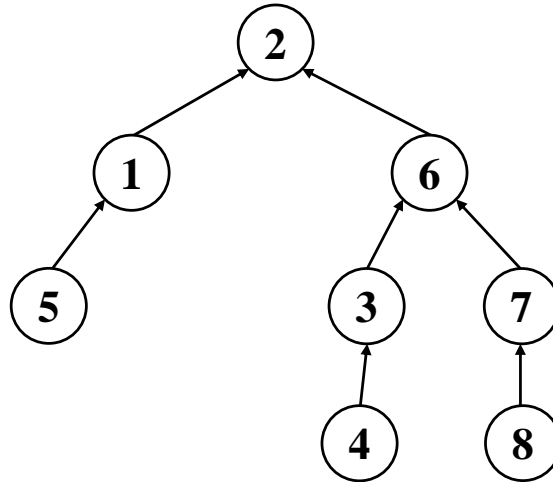
Breadth first search implementation (I)

Number of vertices: 8
Source vertex: 2



Input:

```
8 2
2 5 0
1 6 0
4 6 7 0
3 8 0
1 0
2 3 7 0
3 6 8 0
4 7 0
0 0
```



2015

Breadth first search implementation (II)

```
#include <stdio.h>
#include "queue.h"
#define VMAX 100
#define NIL 0
#define INFTY 10000
enum {WHITE = 0,
      GRAY = 1, BLACK = 2};
/* The graph structure */
int adj[VMAX][VMAX];
int v;
/* The source vertex */
int s;
/* BFS stuff */
int color[VMAX];
int pi[VMAX];
int d[VMAX];
Queue q;
```

```
void read_graph(void) {
    int i, j;
    scanf("%d", &v);
    scanf("%d", &s);
    for (i=1; i<=v; i++) {
        int w;
        j = 0;
        scanf("%d", &w);
        while (w>0) {
            adj[i][++j] = w;
            scanf("%d", &w);
        }
        adj[i][0] = j;
    }
}
```

2015

Breadth first search implementation (III)

```

void bfs(int s) {
    int i;

    for (i=1;i<=v;i++) {
        if (i != s) {
            color[i] = WHITE;
            d[i] = INFTY;
            pi[i] = NIL;
        }
    }
    color[s] = GRAY;
    d[s] = 0;
    pi[s] = NIL;
    queueInit(&q);
    queuePush(&q,s);

    while (!queueEmpty(&q)) {
        int u = queueFront(&q);
        int j;
        for (j=1;j<=adj[u][0];j++) {
            int x = adj[u][j];
            if (color[x] == WHITE) {
                color[x] = GRAY;
                d[x] = d[u]+1;
                pi[x] = u;
                queuePush(&q,x);
            }
        }
        queuePop(&q);
        color[u] = BLACK;
    }
}

```

2015

Breadth first search implementation (IV)

```

void print_result(void) {
    int i;
    printf("d:\n");
    for (i=1;i<=v;i++) {
        printf("%4d",d[i]);
    }
    printf("\n"); printf("pi:\n");
    for (i=1;i<=v;i++) {
        printf("%4d",pi[i]);
    }
    printf("\n");
}

void main(void) {
    read_graph();
    while (v>0) {
        bfs(s);
        print_result();
        read_graph();
    }
}

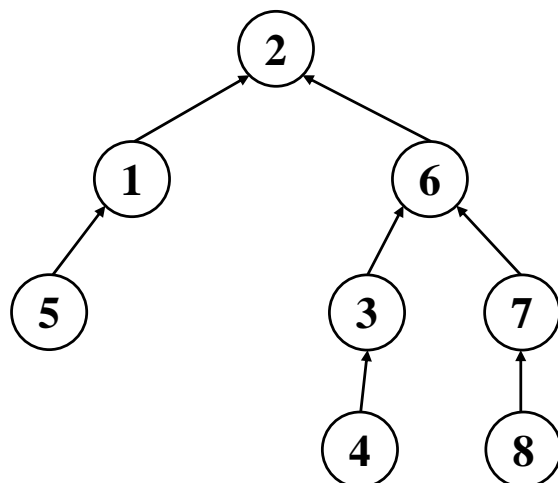
```

Output:

```

d:
    1  0  2  3  2  1  2  3
pi:
    2  0  6  3  1  2  6  7

```



2015

Depth First Search – Fundamentals

- The strategy is to firstly search as deep as possible.
 - Edges / arcs are explored out of the most recently discovered vertex v that has unexplored edges leaving it.
 - When all edges / arcs out of vertex v are explored, the search *backtracks* to explore edges leaving the predecessors of v .
 - This process continues until we have discovered all the vertices that are reachable from the original source vertex.
 - If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source.
 - The entire process is repeated until all vertices are discovered.
 - The DFS algorithm computes the structure π , like BFS. However π defines a forest of depth first trees, because in our approach to DFS we are looking for visiting all the graph vertices, not only the vertices reachable from a given source vertex.
 - Differently from BFS, DFS is using time stamps for the encountered vertices. The time stamps are stored in d and f . d saves the time when a vertex is *discovered* and f saves the time when all the neighbors of a vertex have been visited, i.e. the vertex is *finished*.

2015

Depth first search algorithm

DFS(G)

1. **for** each vertex $u \in V[G]$ **do**
2. $color[u] \leftarrow WHITE$
3. $\pi[u] \leftarrow NIL$
4. $time \leftarrow 0$
5. **for** each vertex $u \in V[G]$ **do**
6. **if** $color[u] = WHITE$ **then**
7. DFS-VISIT(u)

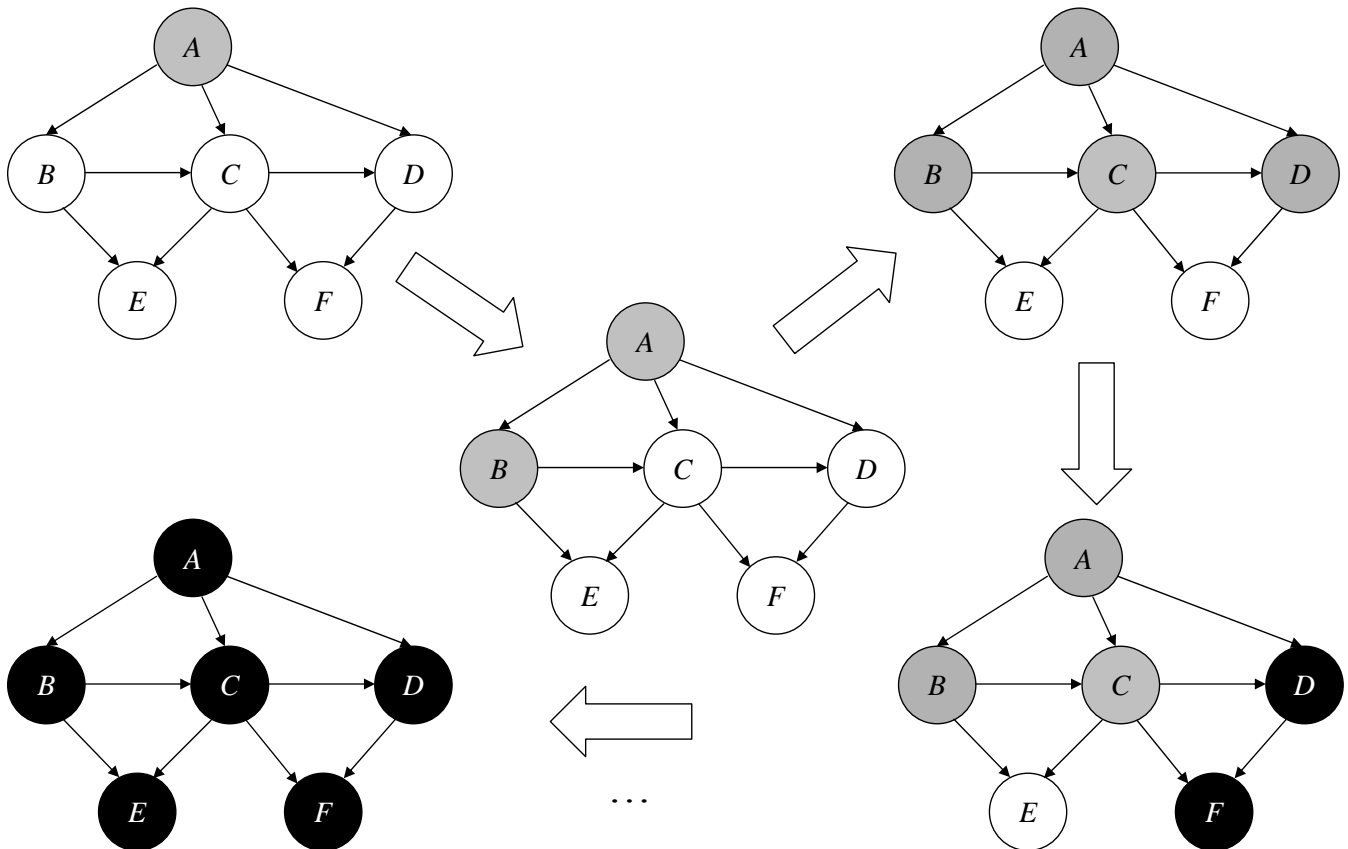
DFS-VISIT(u)

1. $color[u] \leftarrow GRAY$
2. $d[u] \leftarrow time \leftarrow time + 1$
3. **for** each $v \in Adj[u]$ **do**
4. **if** $color[v] = WHITE$ **then**
5. $\pi[v] \leftarrow u$
6. DFS-VISIT(v)
7. $color[u] \leftarrow BLACK$
8. $f[u] \leftarrow time \leftarrow time + 1$

- A vertex that has not been reached yet is colored *white*.
- Lines 1 – 4 of DFS perform some initialization, including coloring all the vertices with white and resetting the time stamp.
- Lines 5 – 7 of DFS search for a white vertex and then call DFS-VISIT to start building a new depth first tree.
- Lines 1 – 2 of DFS-VISIT perform the initialization by coloring u to gray and setting its d time stamp.
- Lines 3 – 6 examine all the neighbors of u that were not visited yet. Whenever a new white neighbor is discovered we go deeper by calling DFS-VISIT recursively.
- Finally, after all the neighbors of u were visited we color it with black and set its f time stamp in lines 7 – 8.

2015

Depth first search example



2015

Analysis of depth first search

- Analysis of the execution time
 - The loop of lines 3 – 6 of DFS-VISIT is executed $|Adj[v]|$ times.
 - Since the sum of the lengths of all the adjacency lists is $\Theta(|E|)$, the total cost of DFS-VISIT(u) is $\Theta(|E|)$.
 - Lines 1 – 3 and 5 – 7 of DFS add an overhead of $\Theta(|V|)$, so the execution time of DFS(G) is $\Theta(|V|+|E|)$.

2015

Properties of depth first search I

- Some properties:
 - the predecessor sub-graph G_π defines a forest of trees
 - the structure of the depth-first trees and the structure of recursive calls to DFS-VISIT(v) are identical
 - discovery and finish times have a parenthesis structure, i.e. they define non partially overlapping intervals.
- Theorem: In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v , exactly one of the following three conditions holds:
 - the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint,
 - the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$ and u is a descendant of v in the depth-first tree or
 - the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$ and v is a descendant of u in the depth-first tree.

2015

Properties of depth first search II

- Intervals $[d, f]$ defined by descendants in the depth-first forest of a graph G are nested. This result is formulated below.
- Corrolary: Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if $d[u] < d[v] < f[v] < f[u]$.
- Theorem (white path theorem): In a depth-first forest of a (directed or undirected) graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $d[u]$ that the search discovers u , vertex v can be reached from u along a path consisting entirely of white vertices.

2015

Depth first search implementation (I)

```
#include <stdio.h>

#define VMAX 100
#define NIL 0
enum {WHITE = 0, GRAY = 1,
      BLACK = 2};

/* The graph structure */
int adj[VMAX][VMAX];
int v;

/* DFS stuff */
int color[VMAX];
int pi[VMAX];
int d[VMAX];
int f[VMAX];
int time;

void read_graph(void) {
    int i,j;

    scanf("%d",&v);
    for (i=1;i<=v;i++) {
        int w;
        j = 0;
        scanf("%d",&w);
        while (w>0) {
            adj[i][++j] = w;
            scanf("%d",&w);
        }
        adj[i][0] = j;
    }
}
```

2015

Depth first search implementation (II)

```
void dfs_visit(int u) {
    int i;
    color[u] = GRAY;
    d[u] = ++time;
    for (i=1;i<=adj[u][0];i++) {
        if (color[adj[u][i]]==WHITE)
        {
            pi[adj[u][i]] = u;
            dfs_visit(adj[u][i]);
        }
    }
    color[u] = BLACK;
    f[u] = ++time;
}

void dfs(void) {
    int i;

    for (i=1;i<=v;i++) {
        color[i] = WHITE;
        pi[i] = NIL;
    }
    time = 0;

    for (i=1;i<=v;i++) {
        if (color[i] == WHITE) {
            dfs_visit(i);
        }
    }
}
```

2015

Depth first search implementation (III)

```
void print_result(void) {
    int i;
    printf("d:\n");
    for (i=1;i<=v;i++) {
        printf("%4d",d[i]);
    }
    printf("\n");
    printf("f:\n");
    for (i=1;i<=v;i++) {
        printf("%4d",f[i]);
    }
    printf("\n");
    printf("pi:\n");
    for (i=1;i<=v;i++) {
        printf("%4d",pi[i]);
    }
    printf("\n");
}

void main(void) {
    read_graph();
    while (v>0) {
        dfs();
        print_result();
        read_graph();
    }
}
```

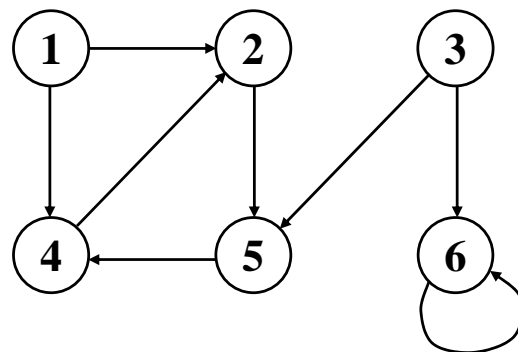
2015

Depth first search implementation (IV)

Number of vertices: 6

Input:

```
6
2 4 0
5 0
5 6 0
2 0
4 0
6 0
0
```



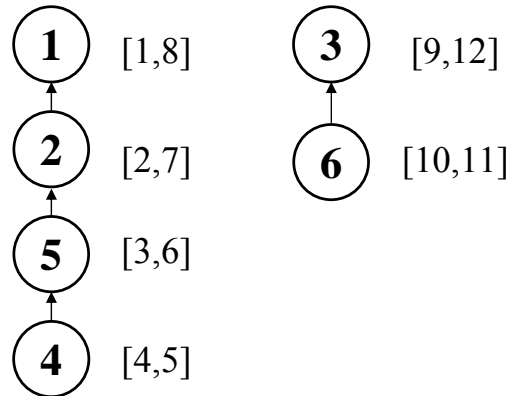
Output:

```
d:
  1  2  9  4  3 10
f:
  8  7 12  5  6 11
pi:
  0  1  0  5  2  3
```

2015

Example depth-first forest

- Note that directions of arcs are indicated according to the parent, so they are opposite from their direction in the graph.



2015

Edges / arcs classification

- We can define four edge types in terms of the depth-first forest G_π produced by a depth-first search on G .
 - Tree edges* are edges in the depth-first forest G_π . Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v) .
 - Back edges* are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops, which may occur in directed graphs, are considered to be back edges.
 - Forward edges* are those non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
 - Cross edges* are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.
- Homework:** Classify the edges of the example graph according to these criteria.

2015

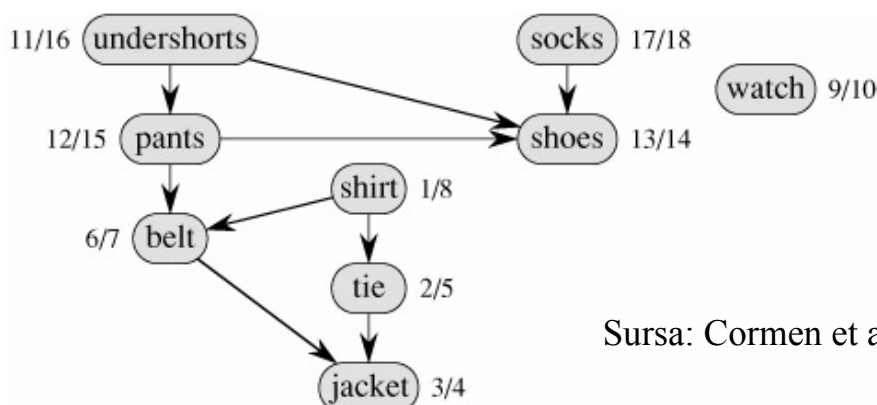
Edges classification in an undirected graph

- Theorem: In a depth-first search of an undirected graph G , every edge of G is either a tree edge or a back edge.
- So, in an undirected graph there are no cross edges or forward edges !

2015

Topological Sort

- Let us consider a DAG $G = (V, E)$. So graph G is:
 - Directed
 - Acyclic
- Definition: A *topological sort* of G is a linear ordering of the vertices V such that if G contains an arc (u, v) then u precedes v in the ordering.

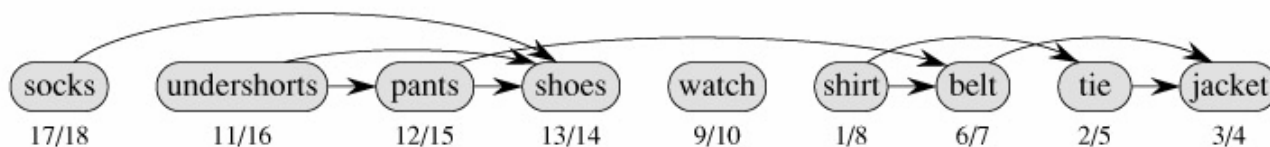


Sursa: Cormen et al, 2001

2015

Sketch of the Topological Sort Algorithm

- We can use DFS to produce a topological sort. The algorithm proceeds as follows:
 - Create an empty list L of the topological sort
 - Call $\text{DFS}(G)$ and track the computation of finishing times $f[v]$ for each $v \in V$
 - Whenever a vertex v is finished it is inserted to the front of list L
 - Return list L



Sursa: Cormen et al, 2001

2015

Correctness of Topological Sort Algorithm

- Lemma: A directed graph G is a DAG if and only if a depth-first search of G yields no back edges.
- Intuitively, a back edge determines a cycle in the graph.
- Theorem: the algorithm for computing a topological sort using DFS is correct.
- It can be shown that for any two vertices u and v if there is an edge from u to v then we have $f[u] > f[v]$. Now, let us assume that there is an edge (u,v) such that v precedes u in the topological sort ordering. Then, note that the result list of the topological sort will be sorted in decreasing order of finishing times, so we have $f[v] > f[u]$. But this is a contradiction !

2015

Homework

1. A directed graph $G = (V, E)$ is singly connected if there is at most one simple path from u to v for all vertices $u, v \in V$. Give an efficient algorithm to determine whether or not a directed graph is singly connected.
2. Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices s and t , and returns the number of paths from s to t in G .