[CSE3081(2반)] 알고리즘 설계와 분석

2020학년도 2학기

강의자료

(2020.11.24 화요일)

서강대학교 공과대학 컴퓨터공학과
임 인 성 교수

[주제 6]

Graph Algorithms

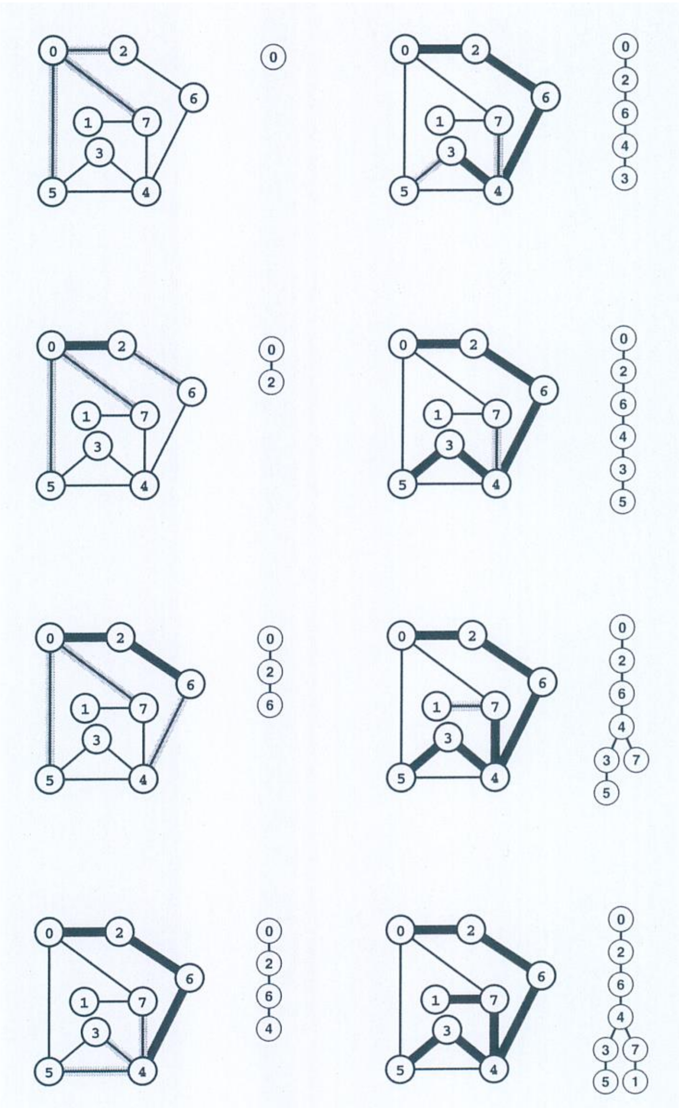
Some Problems Related to Graph Search

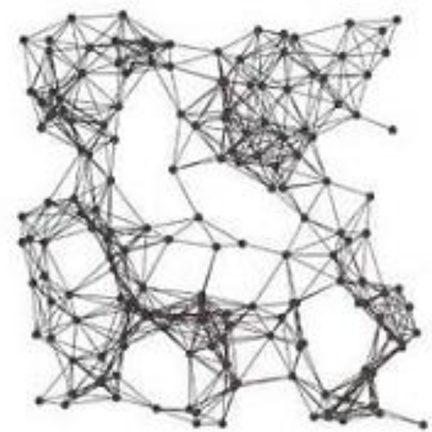
- **Cycle detection**
  - Does a given graph have any cycle?
- **Simple path**
  - Given two vertices, is there a path in the graph that connects them?
- **Simple connectivity**
  - Is a graph connected?
- **Two-way Euler tour**
  - Find a path that uses all the edges in a graph exactly twice – once in each direction.
- **Spanning tree**
  - Given a connected graph with n vertices, find a set of n-1 edges that connects the vertices.
- **Vertex search**
  - How many vertices are in the same connected component as a given vertex?
- **Two-colorability, bipartiteness, odd cycle**
  - Is there a way to assign one of two colors to each vertex of a graph such that no edge connects two vertices of the same color?

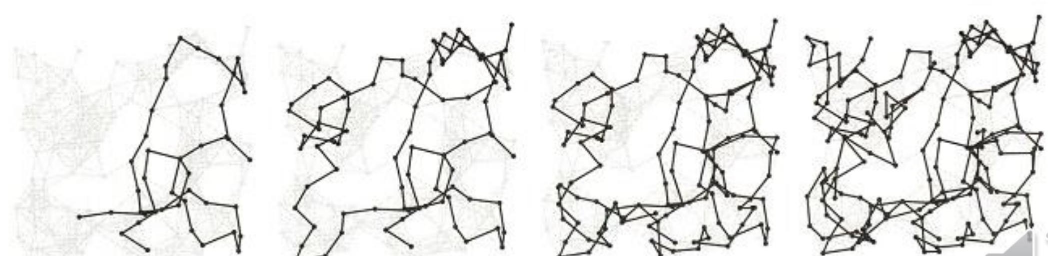
  - Is a given graph bipartite?
  - Does a given graph have a cycle of odd length?
- ***st*-connectivity**
  - What is the minimum number of edges whose removal will separate two given vertices s and t in a graph?
- **General connectivity**
  - Is a graph *k*-connected?
  - Is a given graph *k*-edge connected?
  - What is the edge connectivity and the vertex connectivity of a given graph?
- **Shortest path**
  - Find a shortest path in the graph from *v* to *w*.
- **Single-source shortest paths**
  - Find shortest paths connecting a given vertex v with each other vertex in the graph.

자료구조 과목에서 배웠음

Graph Search 1: Depth-First Search (DFS)

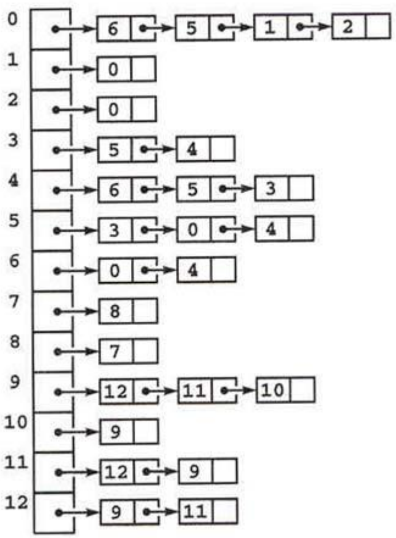


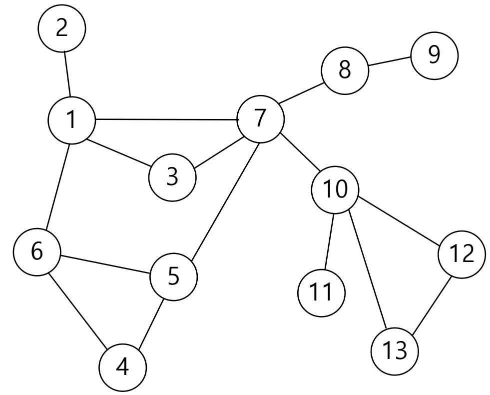




Depth-First Search: Review

- **A graph structure definition**






```
******* GRAPH *******
[1]:  2 3 6 7
[2]:  1
[3]:  1 7
[4]:  5 6
[5]:  4 6 7
[6]:  1 4 5
[7]:  1 3 5 8 10
[8]:  7 9
[9]:  8
[10]:  7 11 12 13
[11]:  10
[12]:  10 13
[13]:  10 12
```

```c
typedef struct _edgenode {
  int y;                     /* adjacency info */
  int weight;                /* edge weight, if any */
  struct _edgenode *next; /* next edge in list */
} edgenode;

typedef struct _graph {
  // The vertices are numbered starting from 1 not 0.
  edgenode *edges[MAXV + 1];/* adjacency info */
  int degree[MAXV + 1];/* outdegree of each vertex */
  int nvertices;/* number of vertices in the graph */
  int nedges;/* number of edges in the graph */
  int directed;/* is the graph directed? */
} graph;
```

A Recursive implementation in C

parent = predecessor
entry time = discovery time
exit time = finish time

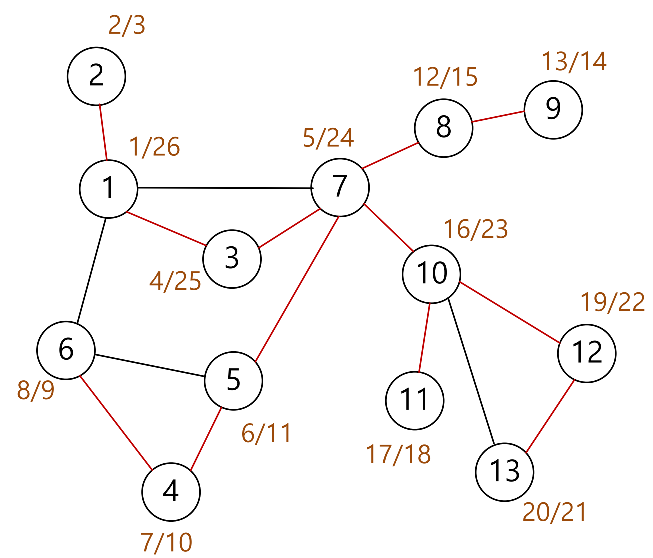
```
void dfs(graph *g, int v) {
  edgenode *p;/* temporary pointer */
  int y;/* successor vertex */

  entry_time[v] = ++time;
  PROCESS_VERTEX_EARLY(v);
  discovered[v] = TRUE;

  p = g->edges[v];
  while (p != NULL) {
    y = p->y;
    if (discovered[y] == FALSE) {
      parent[y] = v;
      PROCESS_EDGE(v, y, g);
      dfs(g, y);
    }
    else
      PROCESS_EDGE(v, y, g);
    p = p->next;
  }

  exit_time[v] = ++time;
  PROCESS_VERTEX_LATE(v);
  processed[v] = TRUE;
}
```

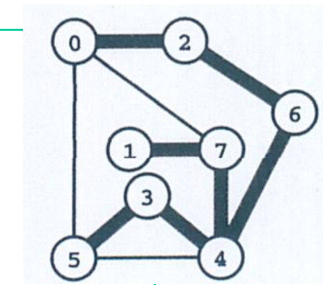
v	parent	entry_time	exit_time
1	-1	1	26
2	1	2	3
3	1	4	25
4	5	7	10
5	7	6	11
6	4	8	9
7	3	5	24
8	7	12	15
9	8	13	14
10	7	16	23
11	10	17	18
12	10	19	22
13	12	20	21



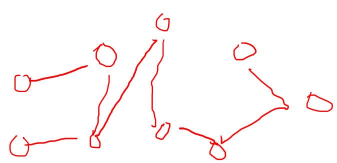
기존에 배운 recursion 기반 구현과 비교할 것

An Abstract Implementation Using a Stack

```
DFS(G, s) { // s is the vertex where the DFS starts.
   Initialize a stack S to be empty;
   visited[v] = F for all vertices in G;
   Push(S, s);
   while (S is not empty) do {
      v = Pop(S);
      if (visited[v] = F) {
         visited[v] = T;
         for (each vertex u that is adjacent to v)
            if (visited[u] = F)
               Push(S, u);
      }
   }
}
```



- 편의상 connected graph로 가정 (아닐 경우에는?)
- 어떤 연산이 전체 탐색을 dominate하는가?
- 각 꼭지점은 unvisited 상태에서 스택에 몇 번 push되는가?
- 전체적으로 각 edge는 몇 번 access되는가?

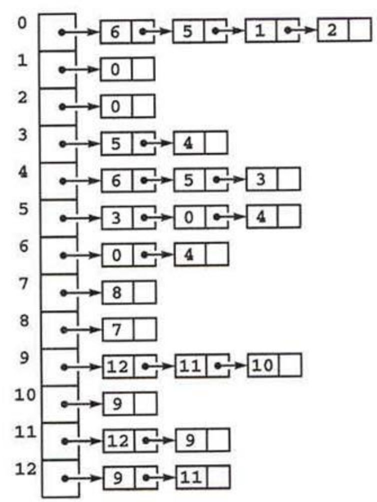


$|V| = 9$

$|E| = 9-1$
$= 8$

$G = (V, E)$

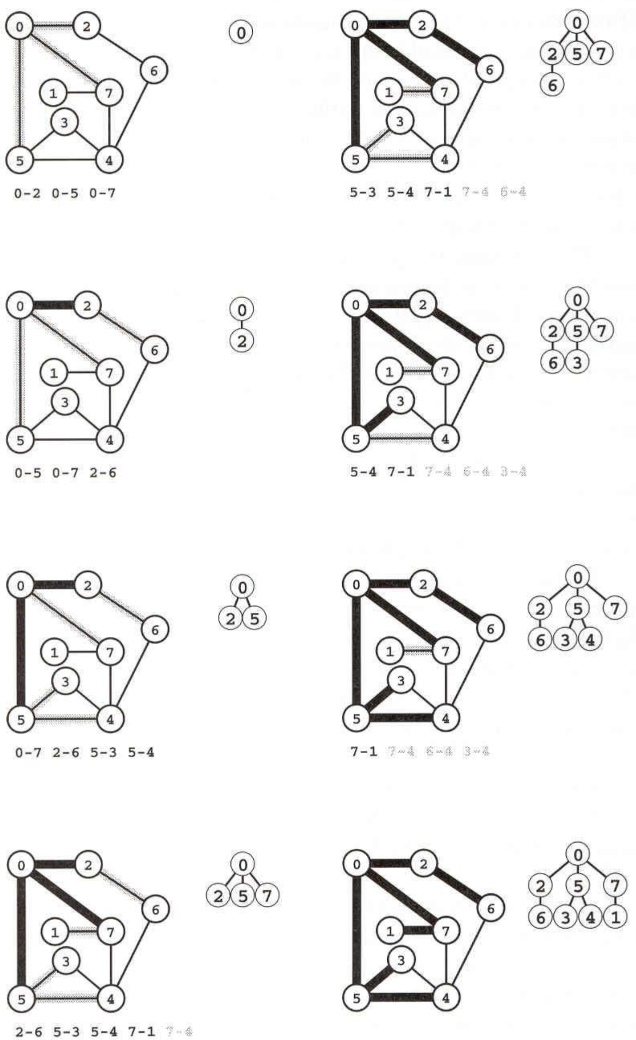
- Time complexity
  - **Adjacency list:** O(|V|+|E|)
  - **Adjacency matrix:** O(|V|^2).

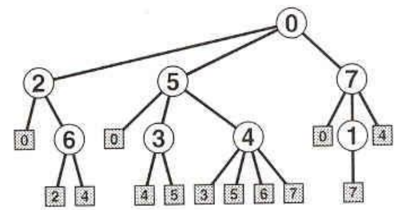


	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

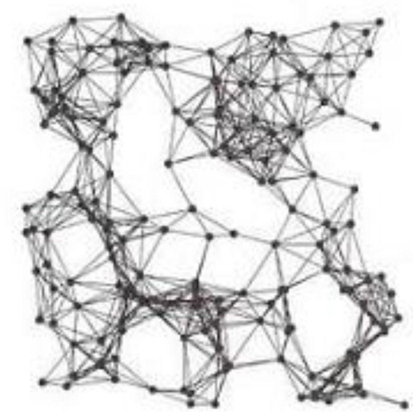
자료구조 과목에서 배웠음

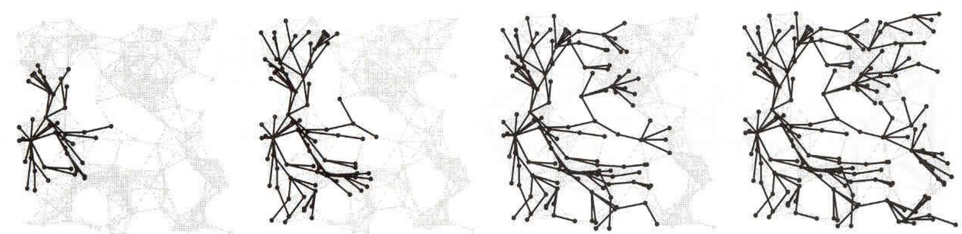
Graph Search 2: Breadth-First Search (BFS)

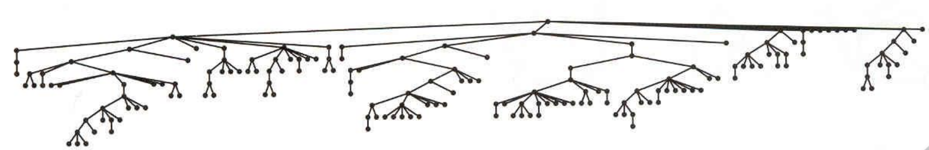




	0	1	2	3	4	5	6	7
pre	0	7	1	5	6	2	4	3
st	0	7	0	5	5	0	2	0







An Abstract Implementation Using a Queue

```
void BFS(G, s) {// s is the vertex where the DFS starts.
   Initialize a queue Q to be empty;
   visited[v] = F for all vertices in G;
   visited[s] = T;
   Insert(Q, s);
   while (Q is not empty) {
      v = delete(Q);
      for (each vertex u that is adjacent to v) {
         if (visited[u] = F) {
            visited[u] = T;
            Insert(Q, u);
         }
      }
   }
}
```

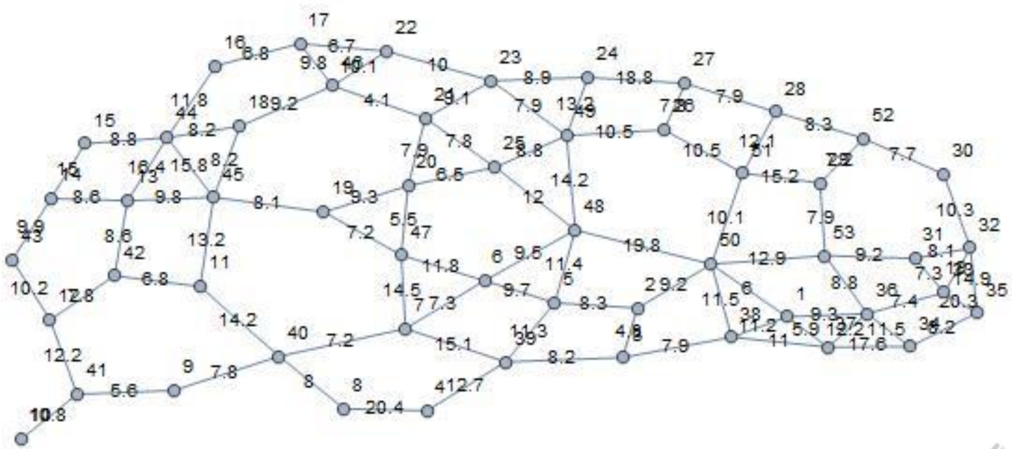
- 편의상 connected graph로 가정 (아닐 경우에는?)
- 어떤 연산이 전체 탐색을 dominate하는가?
- 각 꼭지점은 unvisited 상태에서 스택에 몇 번 push되는가?
- 전체적으로 각 edge는 몇 번 access되는가?

- Time complexity
  - **Adjacency list:** O(|V|+|E|)
  - **Adjacency matrix:** O(|V|^2).

Floyd-Warshall All-Pairs Shortest Path Algorithm

- **Problem**
  - Given a weighted graph $G = (V, E)$ with cost function $cost[i][j]$, find the **shortest paths between all pairs of vertices**. ($V = \{v_0, v_1, v_2, \ldots, v_{n\text{-}1}\}$ with $|V| = n$)

$$cost[i][j] = \begin{cases} 0 & \text{if } i = j, \\ c_{ij} & \text{if } i \neq j \text{ and } (i, j) \in E(G), \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E(G) \end{cases}$$



$v_0, v_1, \cdots, v_{n-1}$

$i \rightsquigarrow j$

0

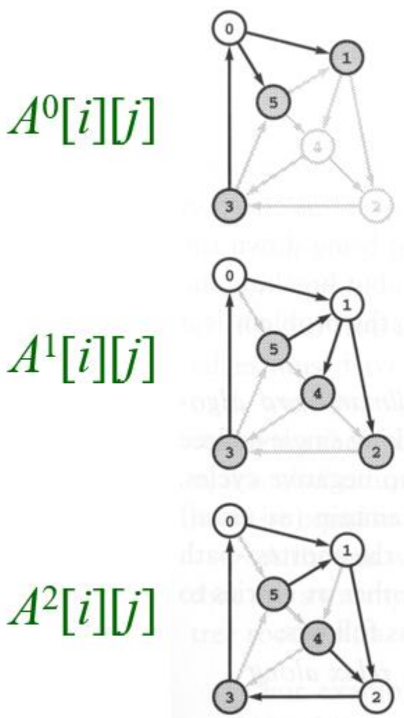
0 1

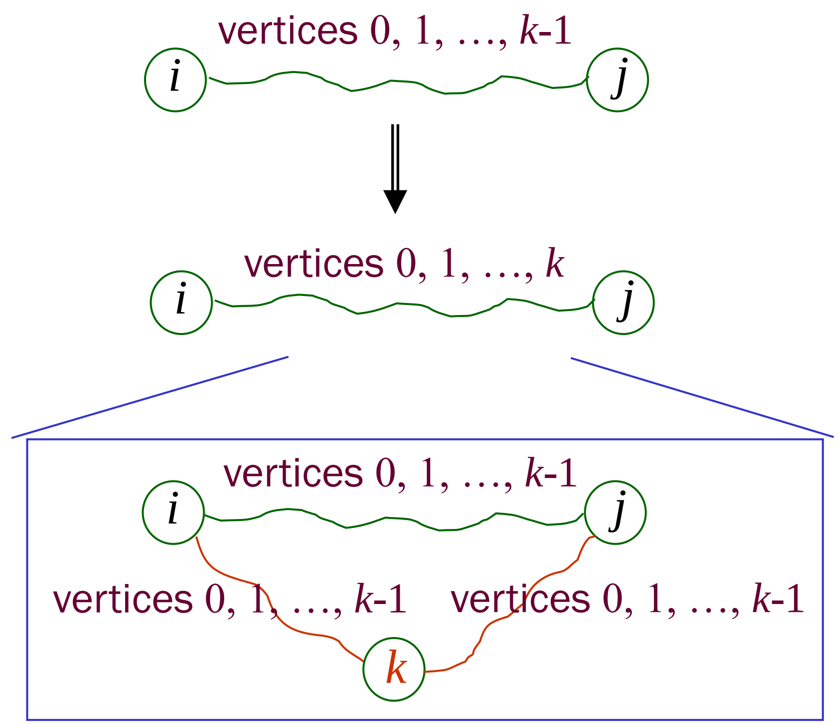
0 1 2

0 1 2 3

0 1 2 ⋯ n-1

- **A dynamic programming approach**
  - Let $A^k[i][j]$ be the cost of the shortest path from i to j, using only those intermediate vertices with an index $\leq k$.
    - ✓ The goal is to compute $A^{n-1}[i][j]$ for all $i, j = 0, 1, 2, \ldots, n-1$.





– **Optimal substructure for computing $A^k[i][j]$ from $A^{k\text{-}1}[i][j]$**

① If the shortest path from i to j going through no vertex with index greater than k **does not** go through the vertex with index k : $A^k[i][j] = A^{k\text{-}1}[i][j]$

② If the shortest path from i to j going through no vertex with index greater than k **does** go through the vertex with index k : $A^k[i][j] = A^{k\text{-}1}[i][k] + A^{k\text{-}1}[k][j]$

$$\boxed{A^k[i][j] = \begin{cases} min\{\ A^{k-1}[i][j],\ A^{k-1}[i][k] + A^{k-1}[k][j]\ \}, & \text{if } k \geq 0, \\ cost[i][j], & \text{if } k = -1 \end{cases}}$$

$A^0[i][j],\ A^1[i][j],\ A^2[i][j],\ \cdots,\ A^{n-1}[i][j]\ \forall i,j$

– **The table computation**

- Initialization / Table traversal order
- **Example:** k = 4 ($A^k[i][j] \leftarrow A^{k-1}[i][j]$)

i \ j	0	1	2	3	4	5	6	7	8
0	0								
1		0							
2			0						
3				0					
4					0				
5						0			
6							0		
7								0	
8									0

An in-place implementation is possible.

```
void allcosts(int cost[][MAX_VERTICES],
                        int distance[][MAX_VERTICES], int n)
{
/* determine the distances from each vertex to every other
vertex,
 cost is the adjacency matrix, distance is the matrix of
distances */
   int i,j,k;
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         distance[i][j] = cost[i][j];
   for (k = 0; k < n; k++)
      for (i = 0; i < n; i++)
         for (j = 0; j < n; j++)
            if (distance[i][k] + distance[k][j] <
                                          distance[i][j])
               distance[i][j] =
               distance[i][k] + distance[k][j];
}
```

$O(n^3)$ time

$$A^k[i][j] = \begin{cases} min\{\ A^{k-1}[i][j],\ A^{k-1}[i][k] + A^{k-1}[k][j]\ \}, & \text{if } k \geq 0, \\ cost[i][j], & \text{if } k = -1 \end{cases}$$

– Path reconstruction

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
let next be a |V| × |V| array of vertex indices initialized to null

procedure FloydWarshallWithPathReconstruction() is
    for each edge (u, v) do
        dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
        next[u][v] ← v
    for each vertex v do
        dist[v][v] ← 0
        next[v][v] ← v
    for k from 1 to |V| do // standard Floyd-Warshall implementation
        for i from 1 to |V|
            for j from 1 to |V|
                if dist[i][j] > dist[i][k] + dist[k][j] then
                    dist[i][j] ← dist[i][k] + dist[k][j]
                    next[i][j] ← next[i][k]
```

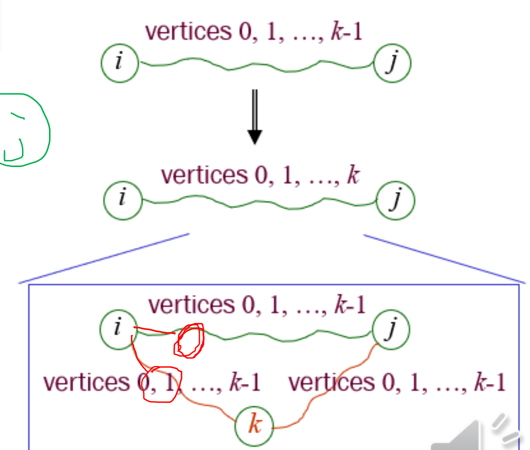
u → ○ → v

next[u][v]

```
procedure Path(u, v)
    if next[u][v] = null then
        return []
    path = [u]
    while u ≠ v
        u ← next[u][v]
        path.append(u)
    return path
```

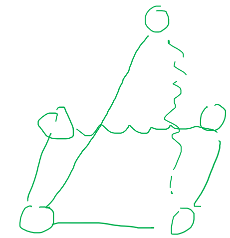
i — l → k → j

next[i][j]



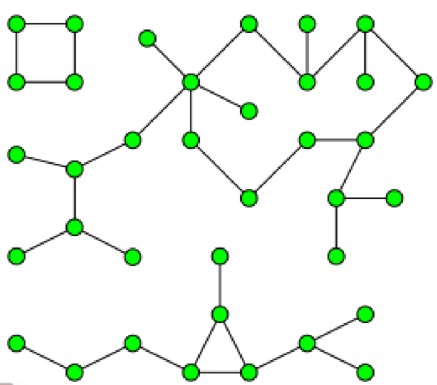
Minimum Spanning Trees

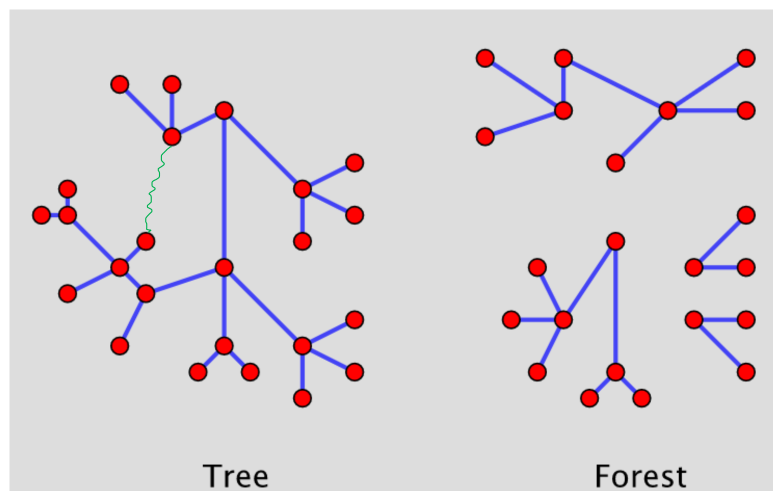
- **Tree**
  - A tree is a connected graph T that contains no cycle.
  - Other equivalent statements (T = (V, E) where |V| = n)
    - T contains no cycles, and has n-1 edges.
    - T is connected, and has n-1 edges.
    - Any two vertices of T are connected by exactly one path.
    - T contains no cycles, but the addition of any new edge creates exactly one cycle.



$n = 5$

- **Forest**
  - A forest is a graph with no cycles.





https://en.wikipedia.org/wiki

https://www.mathreference.org/