Algorithms and Lab (CSE130)

Graph Algorithms

Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

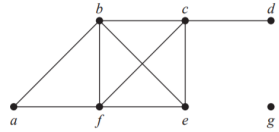
Note: These notes are prepared from the following resources.

- (main text)Foundations of Algorithms, by Richard Neapolitan and Kumarss Naimipour
- Python Algorithm (파이썬 알고리즘) by Y.K. Choi (2021) (Korean)
- Introduction to the Design and Analysis of Algorithms by Anany Levitin
- Introduction to Algorithms, by By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
- https://www.geeksforgeeks.org

Contents

Definitions and Terminologies

- Graph : A graph is a data structure G(V, E) that consists of a collection of vertices V and a collection of edges E, represented as ordered pairs of vertices (u,v)
- Vertices $V = V(G)$ is the set of vertices of G. Vertices are also called nodes and points.
- Edge $E = E(G)$ is the set of edges of G. Each edge connects two different vertices, Edges are also called arcs or lines.
- The order of a graph is its number of vertices |V|. The size of a graph is its number of edges |E|.



- Example:Graph H

  $V = \{a, b, c, d, e, f\}$

  $E = \{(a, b), (a, f), (a, e), (a, f), (b, f), (b, e), (b, d), (b, c), (c, d), (c, f)\}$

  $G = \{V, E\}$

  $|V| = 6, \quad |E| = 10$

- Example: Graph G

  $V = \{a, b, c, d, e, f, g\}$

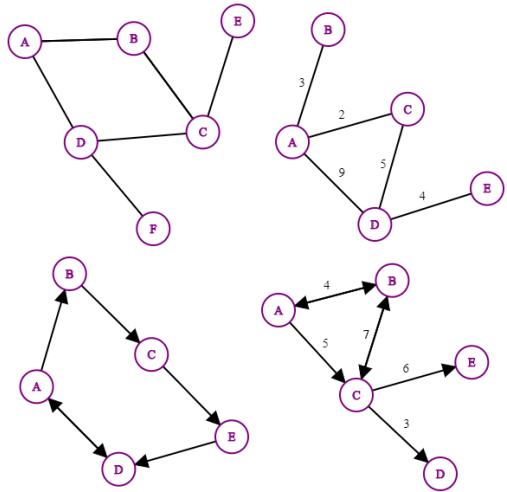
  $E = \{(a, b), (b, c), (c, d), (a, f), (b, f), (c, f), (c, e)\}$

  $G = \{V, E\}$

  $|V| = 7, \quad |E| = 7$

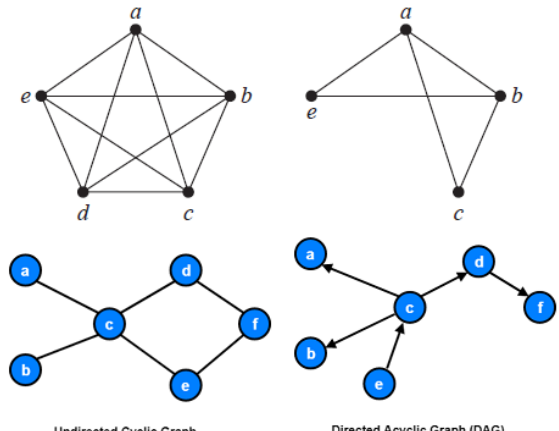
Definitions and Terminologies (cont...)

- Undirected graphs: The graph G is undirected, without any direction on the edges. The edge (A,B) is the same as the edge (B,A). The graphs in the figures (row-1-1, row-1-2) are undirected graph
- Directed graphs have edges with direction. The edges indicate a one-way relationship, i.e. each edge can only be traversed in a single direction. The graphs in figures (row-2-1 and row-2-2) are directed graphs. The digraph in the figure row 2-1 has the vertex set $\{A, B, C, D, E\}$ and the edge set $\{(A, B), (B, C), (C, E), (E, D), (D, A)\}$.
- Unweighted graph: In unweighted graphs, we consider all the edges have same weight, that is equal to 1. Graphs in figures (row 1-1, row-2-1) are unweighted graph
- Weighted graph A weighted graph is a graph G(V,E) where each $e \in E$ is consisting of a pair of vertices and a real number called the weight. Graphs in figures (row 1-2, row 2-2) are weighted graphs



Definitions and Terminologies (cont...)

- Acyclic graph A graph having no cycles is an acyclic graph. A tree is a connected acyclic graph. A directed graph with no cycles is called a Direct Acyclic Graph (DAG) and has many use cases in computer science including the scheduling problems.
- Cyclic graph: A graph with one or more cycles is called a cyclic graph.
- Dense graph A graph can have a quadratic number of edges. If V is the number of vertices in a graph, it can have up to $O(V^2)$ edges. A graph having edges in this order is called a dense graph.
- Sparse graph a graph having a fewer number of edges is called a sparse graph. If V is the number of vertices in a graph, it can have up to $O(V)$ edges.

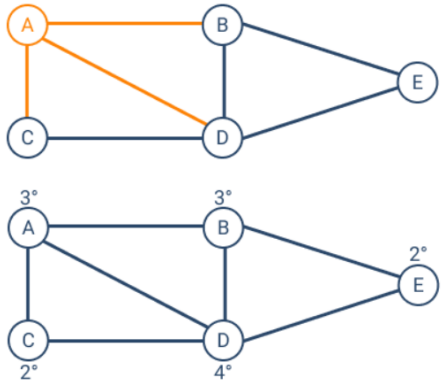


Definitions and Terminologies (cont...)

Adjacent/incident/neighbor vertices

- If $(v_1, v_2) \in E$ is an edge of a graph $G = (V, E)$, we say that v_1 and v_2 are *adjacent* vertices.
- Let $adj(v)$ be the set of all vertices that are adjacent to v. Then we have

$$\begin{aligned}
\text{adj}(A) &= \{B, C, D\} \\
\text{adj}(B) &= \{A, D, E\} \\
\text{adj}(C) &= \{A, D\} \\
\text{adj}(D) &= \{A, B, C, D\} \\
\text{adj}(E) &= \{B, D\}.
\end{aligned}$$



- The edge (v_1, v_2) is also said to be *incident* with the vertices v_1 and v_2.
- The vertices adjacent to v are also referred to as its *neighbors*.

Definitions and Terminologies (cont...)

- Degree of a Vertex v: For any vertex v in a graph $G = (V, E)$, the cardinality of $|adj(v)|$ is called the *degree* of v and written as $\deg(v) = |adj(v)|$.

- The degree of v counts the number of vertices in G that are adjacent to v. If $\deg(v) = 0$, then v is not incident to any edge and we say that v is an *isolated* vertex.

- There is a weighted degree and an unweighted degree. Let G be a graph The *unweighted indegree* of a vertex $v \in V$ counts the edges going into v:

$$\deg_+(v) = \sum_{\substack{e \in E \\ h(e)=v}} 1.$$

- The *unweighted outdegree* of a vertex $v \in V$ counts the edges going out of v:

$$\deg_-(v) = \sum_{\substack{e \in E \\ v \in i(e) = \{v, v'\} \\ h(e)=v'}} 1.$$

- The *unweighted degree* $\deg(v)$ of a vertex v of a weighted multigraph is the sum of the unweighted indegree and the unweighted outdegree of v:

$$\deg(v) = \deg_+(v) + \deg_-(v).$$

Definitions and Terminologies (cont...)

- The *weighted outdegree* of a vertex $v \in V$ counts the weights of edges going out of v:

$$deg_-(v) = \sum_{\substack{e \in E \\ v \in i(e) = \{v, v'\} \\ h(e) = v'}} w_v.$$

- The *weighted degree* of a vertex of a weighted multigraph is the sum of the weighted indegree and the weighted outdegree of that vertex,

$$deg(v) = deg_+(v) + deg_-(v).$$

- In other words, it is the sum of the weights of the edges incident to that vertex, regarding the graph as an undirected weighted graph.

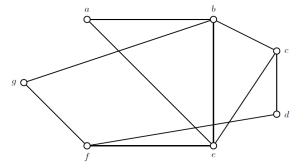
Definitions and Terminologies (cont...)

- Walks, trails, and paths : If u and v are two vertices in a graph G, a *u-v walk* is an alternating sequence of vertices and edges starting with u and ending at v. Consecutive vertices and edges are incident. Formally, a *walk* W of length $n \geq 0$ can be defined as

$$W : v_0, e_1, v_1, e_2, v_2, \ldots, v_{n-1}, e_n, v_n$$

where each edge $e_i = v_{i-1}v_i$ and the length n refers to the number of (not necessarily distinct) edges in the walk.

- A *trail* is a walk with no repeating edges. For example, the *a-b* walk a, b, c, d, f, g, b in figure is a trail. It does not contain any repeated edges, but it contains one repeated vertex, i.e. b.



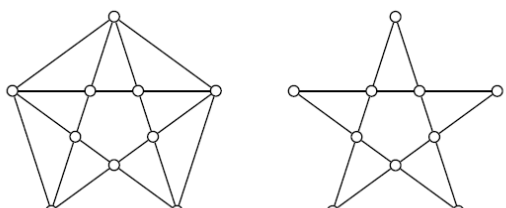
- A walk with no repeating vertices, except possibly the first and last, is called a path.
- A walk of length $n \geq 3$ whose start and end vertices are the same is called a *closed walk*.
- A path of length $n \geq 3$ whose start and end vertices are the same is called a *closed path* For example, the walk a, b, c, e, a in Figure is a closed path.
- A path whose length is odd is called *odd*, otherwise it is referred to as *even*.

Definitions and Terminologies (cont...)

Sub-graph

- Let G be a graph with vertex set $V(G)$ and edge set $E(G)$. Suppose we have a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Then H is a *subgraph* of G and, G is referred to as a *supergraph* of H.

- Starting from G, one can obtain its subgraph H by deleting edges and/or vertices from G. Note that when a vertex v is removed from G, then all edges incident with v are also removed. If $V(H) = V(G)$, then H is called a *spanning subgraph* of G.

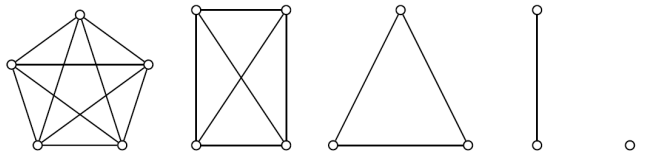


(a) (b)

Definitions and Terminologies (cont...)

- The *complete graph* K_n on n vertices is a graph such that any two distinct vertices are adjacent. As $|V(K_n)| = n$, then $|E(K_n)|$ is equivalent to the total number of 2-combinations from a set of n objects:

$$|E(K_n)| = \binom{n}{2} = \frac{n(n-1)}{2}.$$



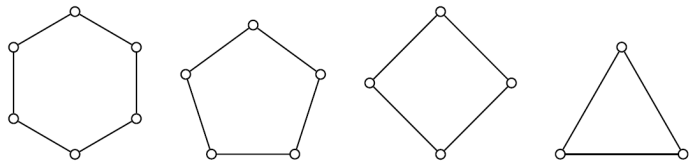
(a) K_5 (b) K_4 (c) K_3 (d) K_2 (e) K_1

Figure shows complete graphs each of whose total number of vertices is bounded by $1 \leq n \leq 5$. The complete graph K_1 has one vertex with no edges. It is also called the *trivial graph*.

- Thus for any simple graph G with n vertices, its total number of edges $|E(G)|$ is bounded above by

$$|E(G)| \leq \frac{n(n-1)}{2}.$$

- The *cycle graph* on $n \geq 3$ vertices, denoted C_n, is the connected 2-regular graph on n vertices. Each vertex in C_n has degree exactly 2 and C_n is connected.



(a) C_6 (b) C_5 (c) C_4 (d) C_3

- Figure shows cycles graphs C_n where $3 \leq n \leq 6$. The *path graph* on $n \geq 1$ vertices is denoted P_n. For $n = 1, 2$ we have $P_1 = K_1$ and $P_2 = K_2$. Where $n \geq 3$, then P_n is a spanning subgraph of C_n obtained by deleting one edge.

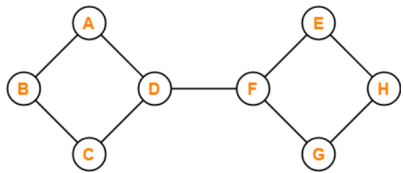
Theorem

Let G be a simple graph with n vertices and k components. Then G has at most $\frac{1}{2}(n-k)(n-k+1)$ edges.

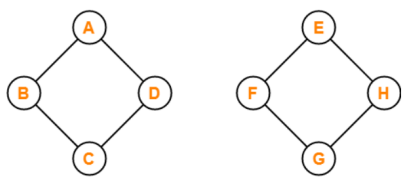
Definitions and Terminologies (cont...)

Connected and non-connected graphs

- A graph is said to be connected if for every pair of distinct vertices u, v there is a u-v path joining them.
- A graph that is not connected is referred to as disconnected/non-connected. The empty graph is disconnected and so is any nonempty graph with an isolated vertex.
- Let H be a connected subgraph of a graph G such that H is not a proper subgraph of any connected subgraph of G. Then H is said to be a *component* of G. We also say that H is a maximal connected subgraph of G. Any connected graph is its own component. The number of connected components of a graph G will be denoted $\omega(G)$



Example of Connected Graph



Example of Disconnected Graph

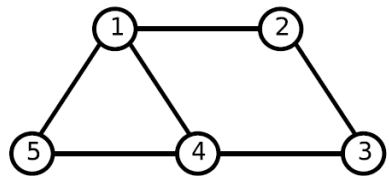
Adjacency matrix

- Let G be an undirected graph with vertices $V = \{v_1, \ldots, v_n\}$ and edge set E.
- The *adjacency matrix* of G is the $n \times n$ matrix $A = [a_{ij}]$ defined by

$$a_{ij} = \begin{cases} 1, & \text{if } (v_i v_j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

- More generally, if G is an undirected multigraph with edge $e_{ij} = (v_i, v_j)$ having multiplicity w_{ij}, or a weighted graph with edge $e_{ij} = (v_i, v_j)$ having weight w_{ij}, then we can define the (weighted) *adjacency matrix* $A = [a_{ij}]$ by

$$a_{ij} = \begin{cases} w_{ij}, & \text{if } v_i v_j \in E, \\ 0, & \text{otherwise.} \end{cases}$$



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

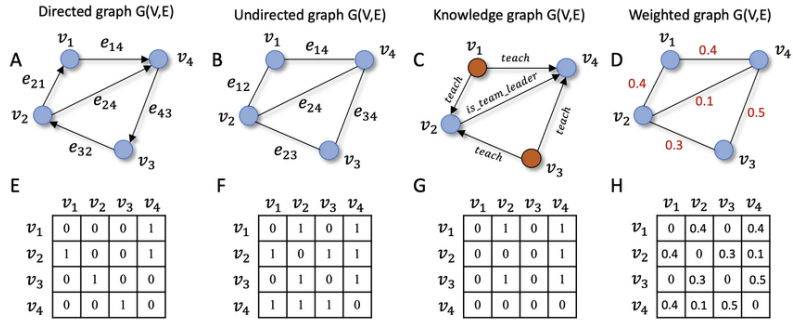
- if G is an undirected graph, then A is a symmetric matrix. That is, A is a square matrix such that $a_{ij} = a_{ji}$.

Representing graphs (cont...)

- Let G be a directed graph with vertices $V = \{v_1, \ldots, v_n\}$ and edge set E. The $(0, -1, 1)$-*adjacency matrix* of G is the $n \times n$ matrix $A = [a_{ij}]$ defined by

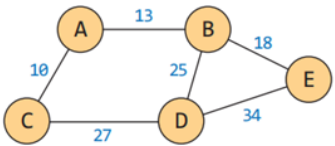
$$a_{ij} = \begin{cases} 1, & \text{if } v_i v_j \in E, \\ -1, & \text{if } v_j v_i \in E, \\ 0, & \text{otherwise.} \end{cases}$$

- In general, the adjacency matrix of a digraph is not symmetric, while that of an undirected graph is symmetric.



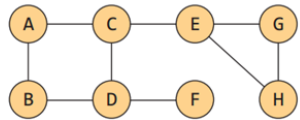
Representing graphs (cont...)

- Example-2: Adjacency matrix representation in Python




```
vertex = ['A',    'B',    'C',    'D',    'E' ]
adjMat = [[0,      13,     10,     None,   None],
          [13,     0,      None,   25,     18 ],
          [10,     None,   0,      27,     None],
          [None,   25,     27,     0,      34 ],
          [None,   18,     None,   34,     0  ]]
```

- Example-3: Adjacency matrix representation in Python

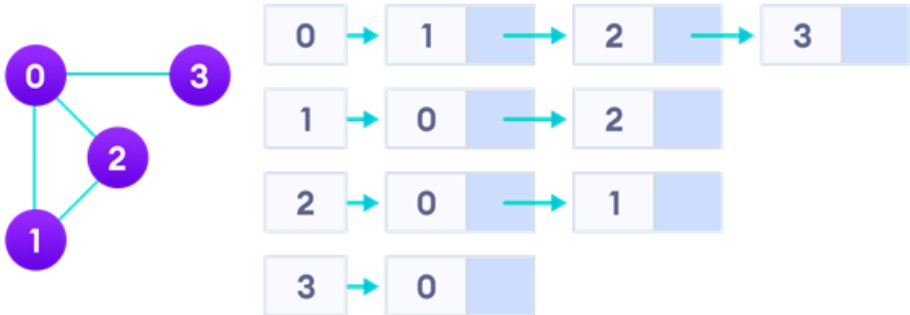



```
vertex =  [ 'A','B','C','D','E','F','G','H']
adjMat = [[ 0, 1, 1, 0, 0, 0, 0, 0 ],
          [ 1, 0, 0, 1, 0, 0, 0, 0 ],
          [ 1, 0, 0, 1, 1, 0, 0, 0 ],
          [ 0, 1, 1, 0, 0, 1, 0, 0 ],
          [ 0, 0, 1, 0, 0, 0, 1, 1 ],
          [ 0, 0, 0, 1, 0, 0, 0, 0 ],
          [ 0, 0, 0, 0, 1, 0, 0, 1 ],
          [ 0, 0, 0, 0, 1, 0, 1, 0 ]]
```

Adjacency List

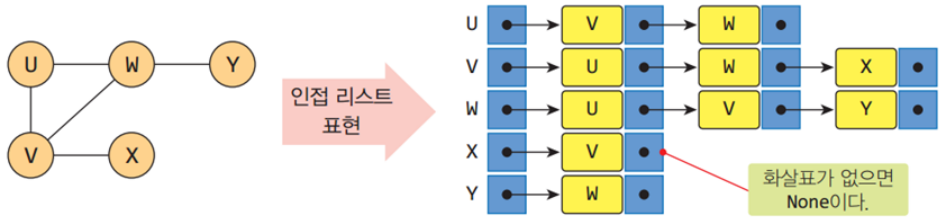
Adjacency List

- An adjacency list represents a graph as an array of linked lists.
- The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.
- Example-1

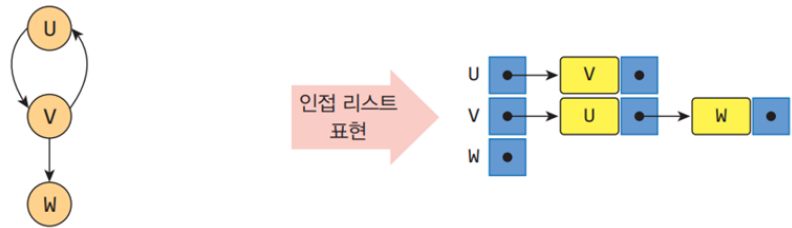


Adjacency List (cont...)

- Example-2

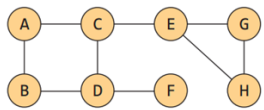


- Example-3



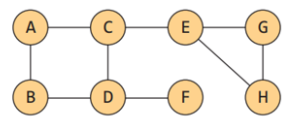
Adjacency List (cont...)

- Example: Adjacency List in Python




```
vertex =  [ 'A','B','C','D','E','F','G','H']
adjList = [[ 1, 2 ],        # 'A'의 인접정점 인덱스
           [ 0, 3 ],        # 'B'의 인접정점 인덱스
           [ 0, 3, 4 ],     # 'C'
           [ 1, 2, 5 ],     # 'D'
           [ 2, 6, 7 ],     # 'E'
           [ 3 ],           # 'F'
           [ 4, 7 ],        # 'G'
           [ 4, 6 ] ]       # 'H'
```

- Example: Graph in Python



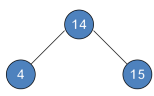

```
graph = { 'A': set(['B','C']),      # 또는 'A': {'B', 'C'}
          'B': set(['A','D']),
          'C': set(['A','D','E']),
          'D': set(['B','C','F']),
          'E': set(['C','G','H']),
          'F': set(['D']),
          'G': set(['E','H']),
          'H': set(['E','G']) }
```

Graph Algorithms

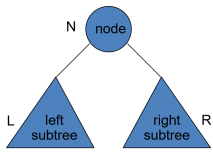
- Searching and Traversing Graphs
  1. Depth-First Search (DFS)
  2. Breadth-First Search (BFS)
- Minimum Spanning Tree
  1. Prim's Minimum Spanning Tree
  2. Kruskal's Minimum Spanning Tree Algorithm
  3. Total number of Spanning Trees in a Graph
  4. Boruvka's algorithm for Minimum Spanning Tree
- Shortest Paths Algorithms in Undirected Graphs
  1. Print all Hamiltonian paths present in a graph
  2.
- Shortest Paths Algorithms in Directed Graphs
  1. Single-Source Shortest Paths Dijkstra's Algorithm
  2. Find the path between given vertices in a directed graph
  3. Find the longest path in a Directed Acyclic Graph (DAG)
- Maximum Flow (minimum cut)
- Miscellaneous Graph Algorithms
  1. isConnected

Searching and Traversing Graphs

- To solve many problems modeled with graphs, we need to visit all the vertices and edges in a systematic fashion called graph traversal.
- Traversal of a graph is commonly used to search a vertex or an edge through the graph; hence, it is also called a search technique.
- Remember: possible ways to print binary tree

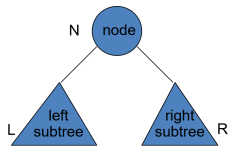


(4, 14, 15), (4,15,14)
(14,4,15), (14,15,4)
(15,4,14), (15,14,4)



(L,N,R), (L,R,N)
(N,L,R), (N,R,L)
(R,L,N), (R,N,L)

- Three common ways



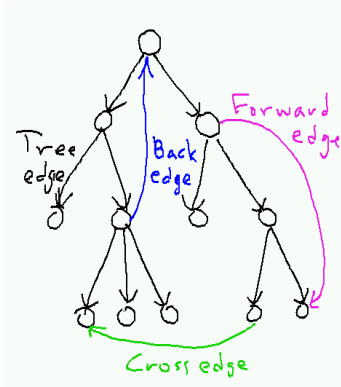
Preorder:	(N,L,R)
Inorder:	(L,N,R)
Postorder:	(L,R,N)

- Two standard methods of searching for such vertices are depth-first search (DFS) and breadth-first search (BFS).

Basic Graph Algorithms (cont...)

Classification of edges: A search done on a graph G results in some solution forest (set of trees) consisting of a subset of edges of G. The edges of G can be classified as follows:

- Tree edges – edges in the solution forest.
- Back edges – edges of the original graph that connects a vertex to its ancestor in a solution tree.
- Forward edges – edges not in the solution forest that connect a vertex to a decendent in a solution tree.
- Cross edges – all other edges. They can go between vertices in the same solution tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different solution trees.



Depth First Search (DFS)

- In DFS, as the name indicates, from the currently visited vertex in the graph, we keep searching deeper whenever possible.
- All the vertices are visited by processing a vertex and its descendents before processing its adjacent vertices.
- This procedure can be written either recursively or non-recursively.
- For recursive code, the internal stack would be used, and for non-recursive code, we would use a stack.
- The recursive algorithm for DFS can be outlined as

```
1: procedure DFS(v)
2:     visited[v] = true
3:     for (each child u of v) do
4:         DFS(u)
5:     end for
6: end procedure
```

- This is called a Depth-First-Search, or DFS for short, because of the way it explores the graph; it sort of "goes deep" before it "goes wide".

Depth First Search (DFS) (cont...)

- Depth First Search with an explicit Stack
  1. Create a stack object
  2. push the source vertex onto the stack and mark it visited
  3. While the stack is not empty
     1. Pop a Vertex v from the stack
     2. Get neighbors/children of the Vertex v
     3. If a child u of V is not marked visited before push it onto the stack and mark it visited

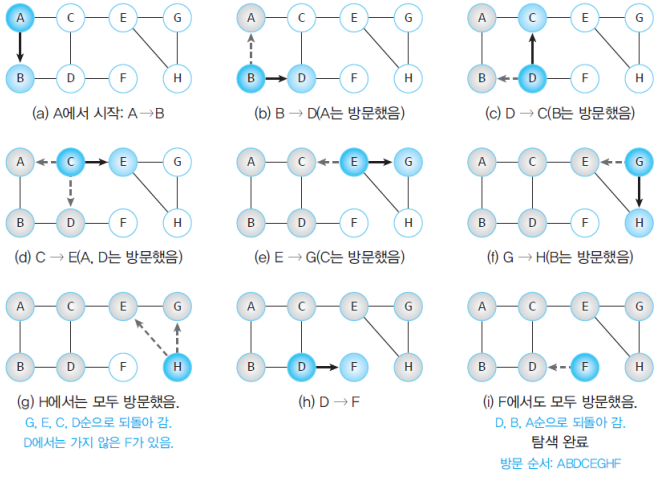
```
1:  procedure DFS(Graph G, Vertex v)
2:      Stack S
3:      visited[v] = true
4:      push(S, v)
5:      while (!empty(S)) do
6:          v=pop(S)
7:          for (each child u of v) do
8:              if (visited[u] = false) then
9:                  visited[u] = true,  push(S, u)
10:             end if
11:         end for
12:     end while
13: end procedure
```

- Complexity: DFS runs in $O(|V| + |E|)$ time. Proof: It explores every vertex once. Once a vertex is marked, it's not explored again. It traverses each edge twice. Overall, $O(|V| + |E|)$.

- Undirected DFS can be used to solve in $O(|V| + |E|)$ time the following problems:
  - is G connected?
  - compute the number of CC of G
  - compute a spanning forest of G
  - compute a path between two vertices of G, or report that such a path does not exist
  - compute a cycle, or report that no cycle exists

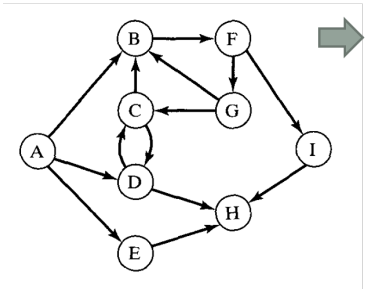
Depth First Search (DFS) (cont...)

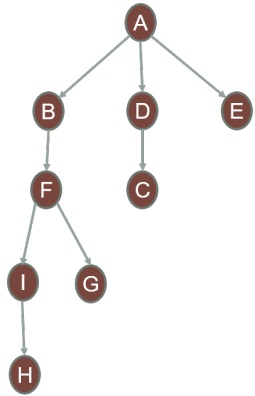
- Example-1



Depth First Search (DFS) (cont...)

- Example-2
  - Vertices with their children without any cycle

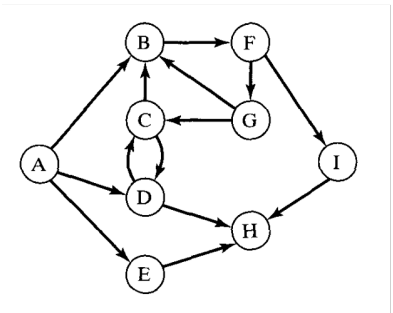


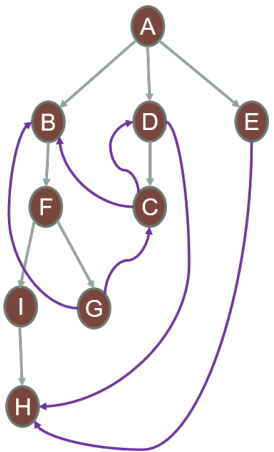


Vertices with their children without any cycle

Depth First Search (DFS) (cont...)

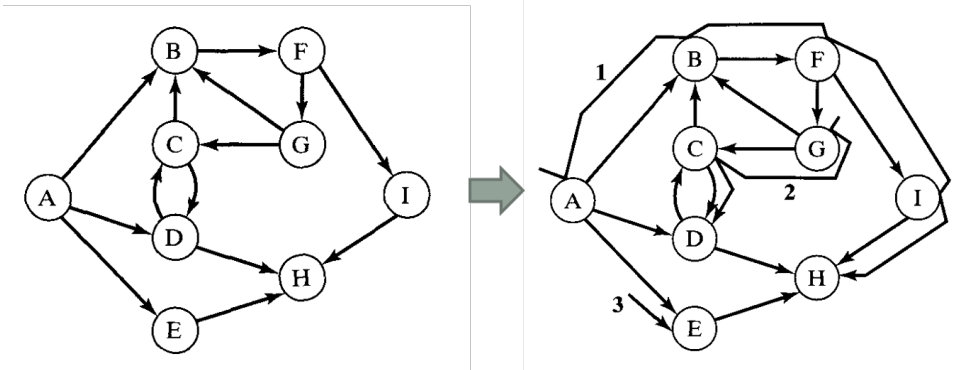
- Two same graphs





Depth First Search (DFS) (cont...)

- DFS results



Breath First Search (BFS)

Breath First Search (BFS)

- Breadth-first search (BFS) is a search technique that expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
- That is, the algorithm discovers all vertices at depth/level k from s before discovering any vertices at depth/level $k+1$.
- This search algorithm uses a queue to store the vertices of each level of the graph as and when they are visited.
- These vertices are then taken out from the queue in sequence, and their adjacent vertices are visited until all the vertices have been visited.

Breath First Search (BFS) (cont...)

- Algorithm: Breath First Search with Queue data structure
  1. Create a queue object
  2. enqueue() the source vertex into the queue and mark it visited
  3. While the stack is not empty
     1. dequeue() a Vertex v from the queue
     2. Get neighbors/children of the Vertex v
     3. If a child u of V is not marked visited before enqueue() it into the queue and mark it visited

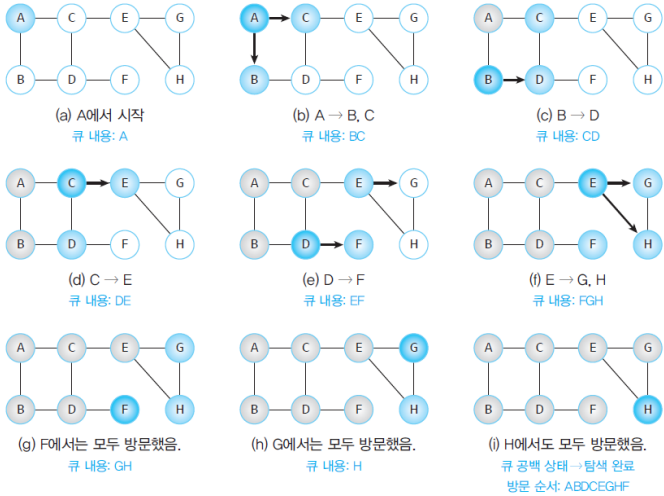
```
1:  procedure BFS(Graph G, Vertex v)
2:      Queue Q
3:      visited[v] = true
4:      enqueue(Q, v)
5:      while (!empty(Q)) do
6:          v=dequeue(Q)
7:          for ((each child u of v)) do
8:              if (visited[u] = false) then
9:                  visited[u] = true
10:                 enqueue(Q, u)
11:             end if
12:         end for
13:     end while
14: end procedure
```

- Complexity: BFS runs in $O(|V| + |E|)$ time. Proof: It explores every vertex once. Once a vertex is marked, it's not explored again. It traverses each edge twice. Overall, $O(|V| + |E|)$.

- Undirected BFS can be used to solve in $O(|V| + |E|)$ time the following problems:
  - is G connected?
  - compute the number of CC of G
  - compute a spanning forest of G
  - compute shortest path between two vertices of G
  - compute a cycle, or report that no cycle exists

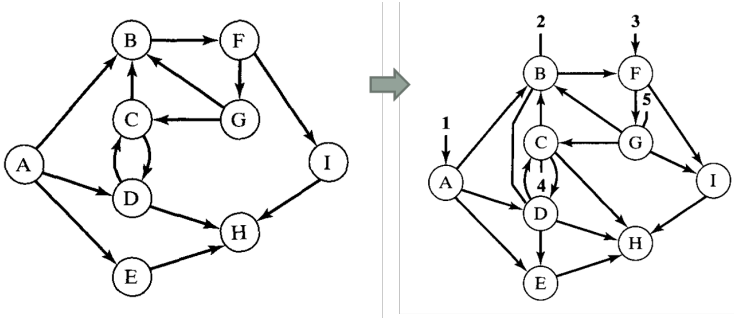
Breath First Search (BFS) (cont...)

- Example-1

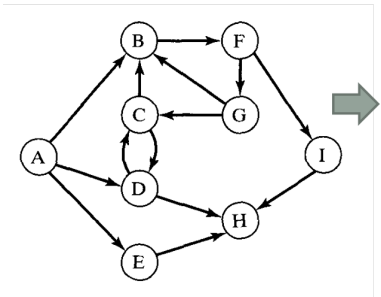


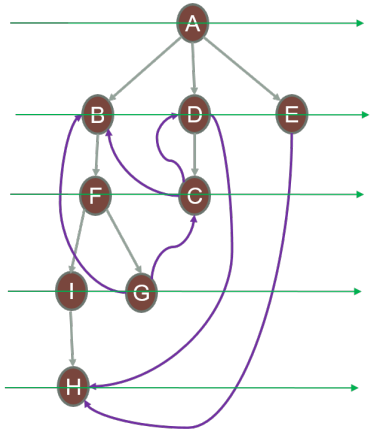
- Example-2

Breath First Search (BFS) (cont...)



Breath First Search (BFS) (cont...)

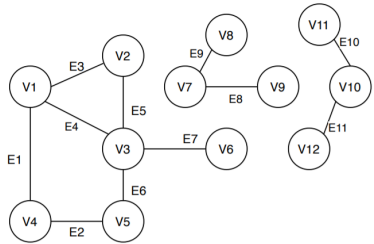


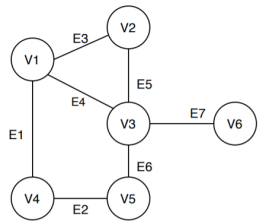


Connected Components

Connected and non-connected graphs

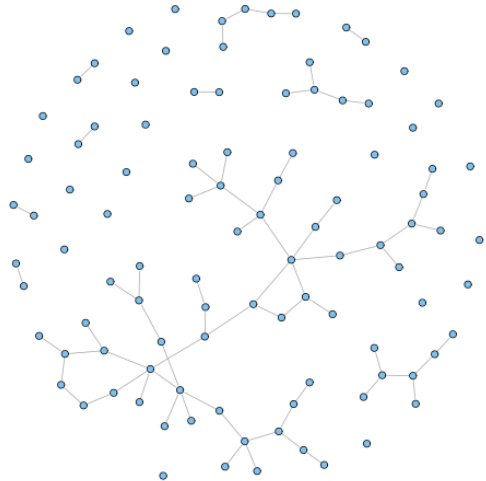
- A graph is said to be *connected* if for every pair of distinct vertices u, v there is a u-v path joining them.
- A graph that is not connected is referred to as *disconnected/non-connected*.
- The empty graph is disconnected and so is any nonempty graph with an isolated vertex.
- In this example, the given undirected graph has one connected component:





Connected Components (cont...)

- Let H be a connected subgraph of a graph G such that H is not a proper subgraph of any connected subgraph of G. Then H is said to be a *component* of G.
- We also say that H is a maximal connected subgraph of G.
- Any connected graph is its own component.
- The number of connected components of a graph G will be denoted $\omega(G)$
- Equivalently, we can say that the relation that associates two nodes if and only if they belong to the same connected component is an equivalence relation, that is it is reflexive, symmetric, and transitive.



Connected Components (cont...)

Finding Connected Components

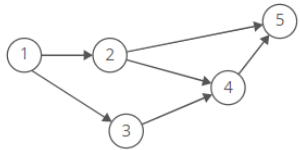
- Either breadth-first or depth-first search can be used to identify the connected components of an undirected graph and label each vertex with the identifier of its components.
- we can modify the DFS-graph algorithm to increment a counter for the current component number and label each vertex accordingly as it is discovered in DFS.
- The key point to observe in the algorithm is that the number of connected components is equal to the number of independent DFS function calls.

Algorithm 1: Finding Connected Components using DFS

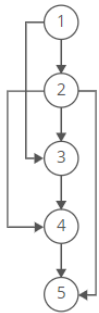
```
Data: Given an undirected graph G(V, E)
Result: Number of Connected Components
Component_Count = 0;
for each vertex k ∈ V do
    Visited[k] = False;
end
for each vertex k ∈ V do
    if Visited[k] == False then
        DFS(V,k);
        Component_Count = Component_Count + 1;
    end
end
Print Component_Count;
Procedure DFS(V,k)
Visited[k] = True;
for each vertex p ∈ V.Adj[k] do
    if Visited[p] == False then
        DFS(V,p);
    end
end
```

Topological Sort

- The topological sort algorithm takes a directed graph and returns an array of the nodes where each node appears before all the nodes it points to.



- Topological sorting for a graph is not possible if the graph is not a DAG.



- The ordering of the nodes in the array is called a topological ordering.

- cyclic graphs don't have valid topological orderings.

Topological Sort (cont...)

The pseudocode of topological sort:

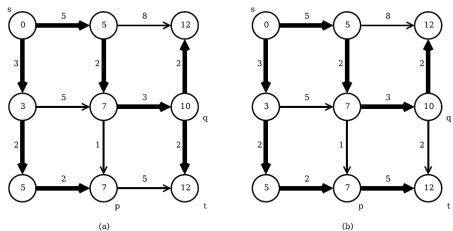
- Step 1: Create the graph g
- Step 2: Call the doTopologicalSort(g)
  1. Step 2.1: Create a result[] and visited[] arrays;
  2. Step 2.2: Mark all the vertices as not visited i.e. initialize visited[ ] with 'false' value.
  3. Step 2.3: Call the recursive helper function dfsTS(vertex, adjList,visited,result) to store Topological Sort starting from all vertices one by one.
- Step 3: dfsTS(vertex,adjList,visited,result):
  1. Step 3.1: Mark the current vertex as visited.
  2. Step 3.2: Recur for all the vertices adjacent to this vertex.
  3. Step 3.3: insert current vertex to result .
- Step 4: At the end, print contents of result.

Single Source Shortest Paths

- Given a connected weighted directed graph $G(V, E)$, associated with each edge $\langle u, v \rangle \in E$, there is a weight $w(u, v)$.
- The single source shortest paths (SSSP) problem is to find a shortest path from a given source r to every other vertex $v \in V - \{r\}$.
- The weight (length) of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

- The weight of a shortest path from u to v is defined by $\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}$.
- Representation of a shortest path tree : The shortest path tree rooted at s is a directed subgraph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$ such that every $v \in V'$ is reachable from s; and for all $v \in V'$, the unique simple path in G' from s to v is a shortest path from s to v in G.



- Shortest Path Tree. The shortest path tree rooted at s has its edges in bold. Two variants (a) and (b) are shown for the same graph. In (a) vertex t is reached through the edge (q, t), whereas in (b), edge (p, t) is used instead.

Single Source Shortest Paths (cont...)

- Initialization The algorithms introduced here make use of a common initialization that is as follows:

```
INITIALIZE(G, s)
1  d[s] ← 0
2  p[s] ← NIL
3  for all v ∈ V − {s}
4       do d[v] ← ∞
5          p[v] ← NIL
```

- Shortest Paths and Relaxation: The main technique used by the shortest path algorithms introduced here is relaxation, a method that repeatedly decreases an upper bound on the length of an actual shortest path for each vertex until the upper bound equals the length of the shortest path.

- For each vertex v, we maintain an attribute $d[v]$ which is an upper bound on the length of a shortest path from source s to v. $d[v]$ is also called the *shortest path estimate*.

```
RELAX(u, v, w)
1  if d[v] > d[u] + w(u, v)
2     then d[v] ← d[u] + w(u, v)
3          p[v] ← u
```

Dijkstra's algorithm

- Dijkstra's algorithm assumes that all the edges in G are non-negative.
- The algorithm maintains a set S of vertices whose final shortest path weights from the source s have already been determined. That is, for all the vertices $v \in S$, we have $d[v] = \delta(s, v)$.
- The algorithm repeatedly selects a vertex $u \in V - S$ with the minimum shortest-path estimate, inserts u to S, and relaxes all the edges leaving u.
- Also, a priority queue Q that contains all the vertices in $V - S$ is maintained, keyed by their d values.

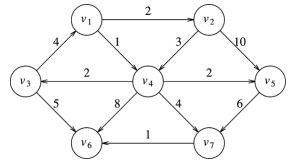
- The Dijkstra-Short algorithm listed below.

```
DIJKSTRA-SHORT(G, w, s)
1  INITIALIZE(G, s)
2  S ← ∅
3  Q ← V
4  while Q ≠ ∅
5        do u ← EXTRACT-MIN(Q)
6           S ← S ∪ {u}
7           for each vertex v ∈ Adj[u]
8                do RELAX(u, v, w)
```

- Complexity :
  - Use a linear array to implement $Q \rightarrow O(n^2 + m) = O(n^2)$.
  - Use a binary heap to implement Q, $\rightarrow O(n + m \log n)$.
  - Use the Fibonacci heap to implement Q, $\rightarrow O(m + n \log n)$.

Dijkstra's algorithm (cont...)

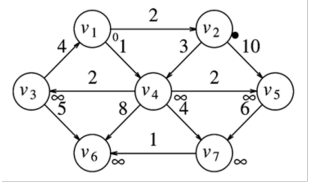
- Example-1: A weighted, directed graph is given. with a set of 7 vertices and a set of 12 edges



$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$

$$E = \begin{bmatrix} 2\,(v_1, v_2), 4\,(v_1, v_3), 1\,(v_1, v_4), 3\,(v_2, v_4), \\ 10\,(v_2, v_5), 5\,(v_3, v_6), 2\,(v_4, v_3), 2\,(v_4, v_5), \\ 8\,(v_4, v_6), 4\,(v_4, v_7), 6\,(v_5, v_7), 1\,(v_7, v_6) \end{bmatrix}$$

- Let the source vertex is $s = v_1$.



- Let's create an array $d_v[]$ where for each vertex v we store the current length of the shortest path from s to v in $d_v[v]$. Initially $d_v[s] = 0$, and for all other vertices this length equals infinity.

v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

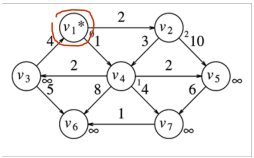
Initial configuration

- A Boolean array *known*[] is maintained which stores for each vertex v whether it's marked. Initially all vertices are unmarked i.e. false:

- An array of predecessors $p_v[]$ is maintained in which for each vertex $u \neq s$. $p_v[v]$ is the penultimate vertex (immediate parent) in the shortest path from s to u.

Dijkstra's algorithm (cont...)

- First Iteration $known[v_1] = true$

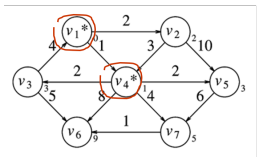


v	$known$	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

After v1 is declared known

- (Do relaxation) distances from v_1 to neighbors (adjacent vertices) are updated as
  $d_v[v_2] = min(2, \infty) = 2, d_v[v_4] = min(1, \infty) = 1$
- parent(v_1) for neighbors (adjacent vertices) is updated, who's path is changed: $p_v[v_2] = v_1, p_v[v_4] = v_1$

- Next Iteration: vertex v_4 is selected based on shortest distance from v_1 to other unmarked vertices.
  $known[v_4] = true$



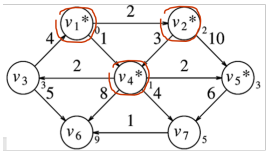
v	$known$	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

After v4 is declared known

- (Do relaxation) distances from v_4 to unmarked neighbors (adjacent vertices) are updated as
  $d_v[v_5] = min(1+2, \infty) = 3, d_v[v_7] = min(1+4, \infty) = 5, d_v[v_6] = min(1+8, \infty) = 9, d_v[v_6] = min(1+2, \infty) = 3$
- parent(v_4) for neighbors (adjacent vertices) is updated, who's path is changed i.e.
  $p_v[v_5] = v_4, p_v[v_7] = v_4, p_v[v_4] = v_4, p_v[v_3] = v_4$

Dijkstra's algorithm (cont...)

- Next Iteration: vertex v_2 is selected based on shortest distance from v_1 to other unmarked vertices
- $known[v_2] = true$

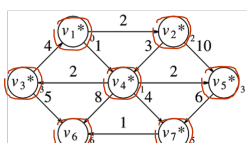


v	$known$	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

After v2 is declared known

- (Do relaxation) distances from v_4 to unmarked neighbors (adjacent vertices) are updated as $d_v[v_5] = min(2 + 10, 3) = 3$
- parent(v_2) for neighbors (adjacent vertices) is updated, who's path is changed : no update

- Same treatment is done for remaining vertices
  - next vertex v is selected based on shortest distance from $v_1 = s$ to other unmarked vertices
  - $known[v] = true$
  - distances from v to unmarked neighbors u (adjacent vertices) are updated as $d_v[u] = min(w(u, v) + d_v[v], d_v[u])$
  - update parent ($p_v[u] = v$), if path is changed.



v	$known$	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

After v5, v3, v7, v6 are known and algorithm terminates

The Bellman-Ford Algorithm

- Like Dijkstra's algorithm, the Bellman-Ford algorithm uses the technique of relaxation, progressively decreasing an estimate $d[v]$ on the weight of a shortest path from a source s to each other vertex $v \in V$ until it reaches the actual shortest path weight $\delta(s, v)$.
- The algorithm is capable of detecting negative cycles and returns true if and only if the graph contains no negative cycles that are reachable from the source.
- If Bellman-Ford returns true, then we have $d[u] = \delta(s, v)$ for all vertices v. Also, for all $v \in V$ not reachable from s, $\delta(s, v) = \infty$.
- The Bellman-Ford algorithm listed below

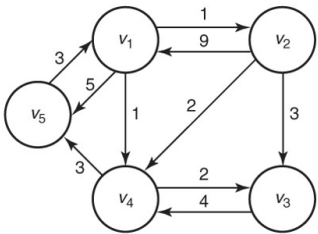
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE(G, s)
2  for i ← 1 to |V| − 1
3       do for each edge (u, v) ∈ E
4            do RELAX(u, v, w)
5  for each edge (u, v) ∈ E
6       do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

All pair Shortest Path

- Given a directed, connected weighted graph $G(V, E)$, for each edge $\langle v_i, v_j \rangle \in E$, a weight $w(v_i, v_j)$ is associated with the edge. The **all pairs of shortest paths problem** (APSP) is to find a shortest path from u to v for every pair of vertices v_i and v_j in V.

- The Shortest Paths problem is an optimization problem.

- Each candidate solution has a **value** associated with it, and a solution to the instance is any **candidate solution** that has an **optimal value**.



- In the case of the Shortest Paths problem
  - A **candidate solution** is a path from one vertex to another
  - The **value** is the length of the path, and
  - The **optimal value** is the minimum of these lengths.
- There can be more than one candidate solution to an instance of an optimization problem.
- Algorithms for the APSP problem
  - Matrix Multiplication / Repeated Squaring
  - The Floyd-Warshall Algorithm
  - Transitive Closure of a Graph
  - Johnson's Algorithm

All pair Shortest Path (cont...)

- **Weighted graph representation :** A weighted graph, containing n vertices, is represented by adjacency matrix W

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

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

D

- D matrix contains the solution of all pair shortest path as shown in Figure 1
- D matrix can be computed using Floyd's Algorithm for Shortest Path (Dynamic Programming Approach)

Figure 1: W and D matrices

- The representation of G: The input for the algorithms is an $n \times n$ matrix W.

$$W[i][j] = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of the directed edge } \langle i, j \rangle & \text{if } i \neq j \text{ and } \langle i, j \rangle \in E \\ \infty & \text{if } i \neq j \text{ and } \langle i, j \rangle \notin E \end{cases}$$

Matrix Multiplication Algorithm

- For any two vertices u and v,
  1. if $u = v$, then the shortest path p from u to v is 0.
  2. otherwise, decompose p into $u \to x \to v$, where p' is a path from u to x and contains at most k edges and it is the shortest path from u to x.

- A recursive solution for the APSP problem is defined. Let $d_{ij}^{(k)}$ be the minimum weight of any path from i to j that contains at most k edges.
  1. If $k = 0$, then

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

  2. Otherwise, for $k \geq 1$, $d_{ij}^{(k)}$ can be computed from $d_{ij}^{(k-1)}$ and the adjacency matrix w.
     $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, \min_{1 \leq l \leq n}\{d_{il}^{(k-1)} + w_{lj}\}\}$

SPECIAL-MATRIX-MULTIPLY(A, B)

```
1  n ← length[A]
2  C ← new n × n matrix
3  for i ← 1 to n
4      do for j ← 1 to n
5             do c_ij ← ∞
6                for k ← 1 to n
7                    do c_ij ← min(c_ij, a_ik + b_kj)
8  return C
```

- The optimal solution can be computed by calling SPECIAL-MATRIX-MULTIPLY$(D^{(k)}, W)$ for $1 \leq k \leq n - 2$.
- Since SPECIAL-MATRIX-MULTIPLY is called $n - 2$ times, the total running time (Complexity) is $O(n^4)$.

Matrix Multiplication Algorithm (cont...)

The repeated squaring method

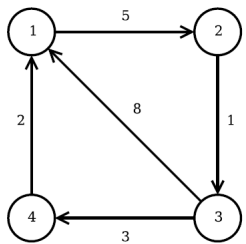
- Since each $D^{(k)}$ matrix contains the shortest paths of at most k edges, and W really is $D^{(1)}$,
- all we were doing in the earlier solution was going: "Given the shortests paths of at most length k, and the shortests paths of at most length 1, what is the shortests paths of at most length $k+1$?"
- The repeated squaring method rephrases the question to: Given the shortests paths of at most length k, what is the shortests paths of at most length $k+k$?
- To improve the time complexity, a repeated squaring method is introduced.
- The running time of the improved matrix multiplication is $O(n^3 \log n)$.

$$
\begin{aligned}
D^{(1)} &= W \\
D^{(2)} &= W^2 = W.W \\
D^{(4)} &= W^4 = W^2.W^2 \\
&\vdots \\
D^{2^{\lceil \log(n-1) \rceil}} &= W^{2^{\lceil \log(n-1) \rceil}} \\
&= W^{2^{\lceil \log(n-1) \rceil - 1}}.W^{2^{\lceil \log(n-1) \rceil - 1}}
\end{aligned}
$$

- Using repeated squaring, we need to run SMM $(n-1)$ times.

```
ALL-PAIRS-SHORTEST-PATHS(W)
1  n ← length[W]
2  D^(1) ← W
3  m ← 1
4  while m < n − 1
5        do D^(2m) ← SMM(D^(m), D^(m))
6           m ← 2m
7  return D^(n)
```

- Repeat Squaring algorithm: Example



- The weights for this graph in matrix form...

$$W = \begin{pmatrix} 0 & 5 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ 8 & \infty & 0 & 3 \\ 2 & \infty & \infty & 0 \end{pmatrix} = D^{(1)}$$

- Repeatedly squaring this gives us...

$$D^{(1)}.D^{(1)} = \begin{pmatrix} 0 & 5 & 6 & \infty \\ 9 & 0 & 1 & 4 \\ 5 & 13 & 0 & 3 \\ 2 & 7 & \infty & 0 \end{pmatrix} = D^{(2)}$$

$$D^{(2)}.D^{(2)} = \begin{pmatrix} 0 & 5 & 6 & 9 \\ 6 & 0 & 1 & 4 \\ 5 & 10 & 0 & 3 \\ 2 & 7 & 8 & 0 \end{pmatrix} = D^{(4)}$$

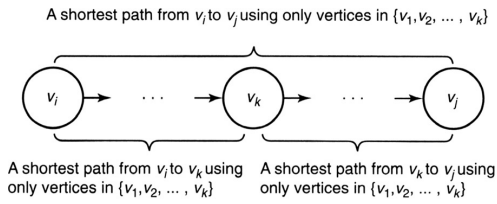
Floyd's Algorithm for Shortest Path

- Floyd-Warshall's algorithm is based upon the observation that a path linking any two vertices u and v may have zero or more intermediate vertices.
- Recursive property is accomplished by considering two cases:
  1. **case-1:** At least one shortest path from v_i to v_j that does not use intermediate vertices v_k, then recursive relation will be

$$D^{(K)}[i][j] = D^{(K-1)}[i][j]$$

  2. **case-2:** All shortest paths from v_i to v_j, do use v_k.

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



- The recursive relation for shortest path will be

$$D^{(K)}[i][j] = \text{minimum}\left(\underbrace{D^{(K-1)}[i][j]}_{Case-1}, \underbrace{D^{(K-1)}[i][k] + D^{(K-1)}[k][j]}_{Case-2}\right).$$

- We need to compute n matrices recursively, $D^{(0)}, D^{(1)}, D^{(2)}, \cdots, D^{(n)}$ where; $D^{(0)} = W$ and $D = D^{(n)}$

Floyd's Algorithm for Shortest Path (cont...)

Algorithm: Floyd's Algorithm for Shortest Paths

- **Problem:** Compute the shortest paths from each vertex in a weighted graph to each of the other vertices. The weights are nonnegative numbers.
- **Inputs:** A weighted, directed graph and n, the number of vertices in the graph. The graph is represented by a two-dimensional array W which has both its rows and columns indexed from 1 to n.
- **Outputs:** A two-dimensional array D, which has both its rows and columns indexed from 1 to n, where $D[i][j]$ is the length of a shortest path from the i^{th} vertex to the j^{th} vertex.
- **Pseudo-code**

```
1:  procedure FLOYD(integer n, number W[][], number D[][])
2:      integer i, j, k                          ▷ variables i, j, k of type integer
3:      D = W
4:      for (k = 1; k <= n; k++) do
5:          for (i = 1; i <= n; i++) do
6:              for for(j = 1; j <= n; j++) do
7:                  D[i][j] = minimum(D[i][j], D[i][k] + D[k][j])
8:              end for
9:          end for
10:     end for
11: end procedure
```

- Time complexity function $T(n) = n \times n \times n = n^3$
- Complexity in terms of asymptomatic notations: $T(n) = n \times n \times n = n^3 \in \Theta\left(n^3\right)$

Floyd's Algorithm for Shortest Path (cont...)

Algorithm: modified Floyd's Algorithm with additional matrix containing Shortest Paths index

- **Problem:** same as in the previous Algorithm, except shortest paths are also created.
- **Additional outputs:** an matrix $P[i][j]$

```
1:  procedure FLOYD(W[][])
2:      integer i, j, k,  D[][] P[][]
3:      for (i = 1; i <= n; i++) do
4:          for (j = 1; j <= n; j++) do
5:              P[i][j] = 0
6:          end for
7:      end for
8:      D = W
9:      for (k = 1; k <= n; k++) do
10:         for (i = 1; i <= n; i++) do
11:             for for(i = 1; j <= n; j++) do
12:                 if (D[i][k] + D[k][j] < D[i][j]) then
13:                     D[i][j[= D[i[[k] + D[k][j]
14:                     P[i][j] = k
15:                 end if
16:             end for
17:         end for
18:     end for
19: end procedure
```

- The array P produced when Algorithm 3.4 is applied to the graph

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

Floyd's Algorithm for Shortest Path (cont...)

Print Shortest Path

- **Problem:** Print the intermediate vertices on a shortest path from one vertex to another vertex in a weighted graph.
- **Inputs:** the array $P[][]$ produced by Algorithm 3.4, and two indices, q and r, of vertices in the graph that is the input to Algorithm 3.4.
- **Outputs:** the intermediate vertices on a shortest path from v_q to v_r.
- **Pseudo-code**

```
1: procedure PATH(integer q, r)
2:     if (P[q][r]! = 0) then
3:         path(q, P[q][r])
4:         print(v, P[q][r])
5:         path(P[q][r], r)
6:     end if
7: end procedure
```

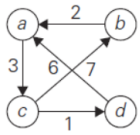
- Time complexity : $T(n) \in \Theta(n)$

Floyd's Algorithm for Shortest Path (cont...)

Example:

- The application of Floyd's algorithm to the graph shown in Figure 2 is illustrated using the recursive property

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



(a)

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

(b)

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(c)

Figure 2: (a) Directed graph. (b) Its weight matrix (Adjacency Matrix) W (c) Its distance matrix D

- for $k = 1$, row-1 (i=1, j= 1,2,3,4)

$$D^{(1)}[1][1] = \min\left[D^{(0)}[1][1],\ D^{(0)}[1][1] + D^{(0)}[1][1]\right] = \min[0, (0+0)] = 0$$
$$D^{(1)}[1][2] = \min\left[D^{(0)}[1][2],\ D^{(0)}[1][1] + D^{(0)}[1][2]\right] = \min[0, (0+\infty)] = \infty$$
$$D^{(1)}[1][3] = \min\left[D^{(0)}[1][3],\ D^{(0)}[1][1] + D^{(0)}[1][3]\right] = \min[3, (0+3)] = 3$$
$$D^{(1)}[1][4] = \min\left[D^{(0)}[1][4],\ D^{(0)}[1][1] + D^{(0)}[1][4]\right] = \min[\infty, (0+\infty)] = \infty$$

Floyd's Algorithm for Shortest Path (cont...)

- for $k = 1$, row-2 (i=2, j= 1,2,3,4)

$$
\begin{aligned}
D^{(1)}[2][1] &= \min\left[D^{(0)}[2][1],\ D^{(0)}[2][1] + D^{(0)}[1][1]\right] = \min[2, (2+0)] = 2\\
D^{(1)}[2][2] &= \min\left[D^{(0)}[2][2],\ D^{(0)}[2][1] + D^{(0)}[1][2]\right] = \min[0, (2+\infty)] = 0\\
D^{(1)}[2][3] &= \min\left[D^{(0)}[2][3],\ D^{(0)}[2][1] + D^{(0)}[1][3]\right] = \min[\infty, (2+3)] = 5\\
D^{(1)}[2][4] &= \min\left[D^{(0)}[2][4],\ D^{(0)}[2][1] + D^{(0)}[1][4]\right] = \min[\infty, (2+\infty)] = \infty
\end{aligned}
$$

- for $k = 1$, row-3 (i=3, j= 1,2,3,4)

$$
\begin{aligned}
D^{(1)}[3][1] &= \min\left[D^{(0)}[3][1],\ D^{(0)}[3][1] + D^{(0)}[1][1]\right] = \min[\infty, (\infty+0)] = \infty\\
D^{(1)}[3][2] &= \min\left[D^{(0)}[3][2],\ D^{(0)}[3][1] + D^{(0)}[1][2]\right] = \min[7, (2+\infty)] = 7\\
D^{(1)}[3][3] &= \min\left[D^{(0)}[3][3],\ D^{(0)}[3][1] + D^{(0)}[1][3]\right] = \min[0, (2+3)] = 0\\
D^{(1)}[3][4] &= \min\left[D^{(0)}[3][4],\ D^{(0)}[3][1] + D^{(0)}[1][4]\right] = \min[1, (2+\infty)] = 1
\end{aligned}
$$

- for $k = 1$, row-4 (i=2, j= 1,2,3,4)

$$
\begin{aligned}
D^{(1)}[4][1] &= \min\left[D^{(0)}[4][1],\ D^{(0)}[4][1] + D^{(0)}[1][1]\right] = \min[6, (6+0)] = 6\\
D^{(1)}[4][2] &= \min\left[D^{(0)}[4][2],\ D^{(0)}[4][1] + D^{(0)}[1][2]\right] = \min[\infty, (6+\infty)] = \infty\\
D^{(1)}[4][3] &= \min\left[D^{(0)}[4][3],\ D^{(0)}[4][1] + D^{(0)}[1][3]\right] = \min[\infty, (6+3)] = 9\\
D^{(1)}[4][4] &= \min\left[D^{(0)}[4][4],\ D^{(0)}[4][1] + D^{(0)}[1][4]\right] = \min[0, (6+\infty)] = 0
\end{aligned}
$$

Floyd's Algorithm for Shortest Path (cont...)

$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

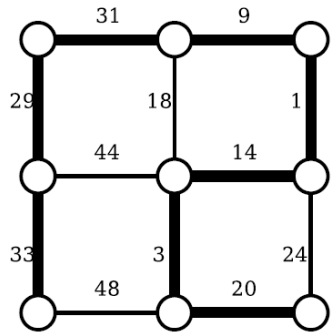
Figure 3: Application of Floyd's algorithm to the digraph is shown. Updated elements are shown in bold.

Minimum Spanning Trees

- Given a connected, weighted, undirected graph $G(V, E)$, for each edge $(u, v) \in E$, there is a weight $w(u, v)$ associated with it.
- The Minimum Spanning Tree (MST) problem in G is to find a spanning tree $T(V, E')$ such that the weighted sum of the edges in T is minimized, i.e.

$$\text{minimize } w(T) = \sum_{(u,v) \in E'} w(u, v), \text{ where } E' \subseteq E$$

- Approaches to Finding MSTs
  - Kruskal's algorithm (Boruvka, 1926)
  - Prim's algorithm (1956)
  - Guan's algorithm (1974)



- For instance, the diagram below shows a graph, G, of nine vertices and 12 weighted edges. The bold edges form the edges of the MST, T. Adding up the weights of the MST edges, we get $w(T) = 140$.

Generic Algorithm for finding MSTs

- The generic algorithm for finding MSTs maintains a subset A of the set of edges E.
- At each step, an edge $\langle u, v \rangle \in E$ is added to A if it is not already in A and its addition to the set does not violate the condition that there may not be cycles in A.
- The algorithm also considers edges according to their weights in non-decreasing order.
- This aspect makes Generic-MST an example of a greedy algorithm.

GENERIC-MST(G, w)

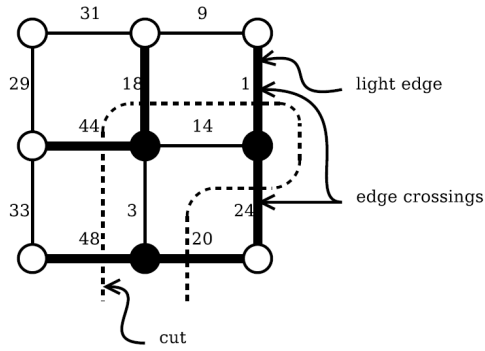
```
1  A ← ∅
2  while the edges in A do not form an MST
3        do find an edge (u, v) ∈ E − A that has
.           the minimum weight and this edge,
.           along with the edges in A,
.           does not form a cycle.
4  A ← A ∪ {(u, v)}
5  return A
```

Evidently, the gist of the MST algorithm is finding an edge $\langle u, v \rangle \in E - A$ that has the minimum weight and that adding this edge to A does not result in a cycle in A.

Generic Algorithm for finding MSTs (cont...)

Terminologies:

- **cut** – A cut is a partition of vertices into two parts; those in a set S and those in the set $V - S$ (i.e. not in S).
- **edge crossing** – An edge that connects a vertex in S to a vertex in $V - S$.
- **light edge** – The edge crossings with the minimum weight. Note that there may be more than one such light edges if two or more edge crossings have the minimum weight.



- Graph showing a cut. Vertices in S are shaded black. Edge crossings are indicated by bold lines. The light edge in this example has a weight of 1.

Theorem Let $G(V, E)$ be a connected undirected weighted graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some MST for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Since (u, v) connects two vertices from two distinct sets S and $V - S$, adding (u, v) to A will not result in loops.

Kruskal's algorithm

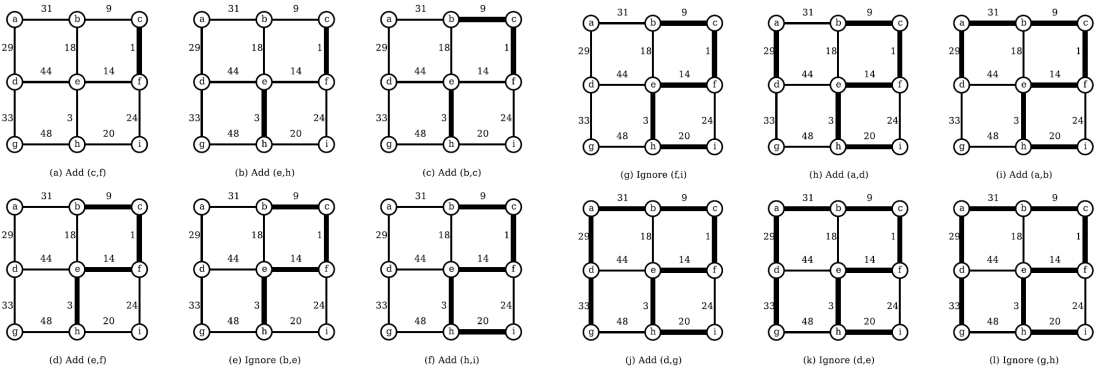
- The set of edges A is initialised as an empty set.
- To keep track of which trees vertices belong to, the algorithm makes use of disjoint-sets.
- Kruskal's algorithm then proceeds by considering each edge of E in order of non-decreasing weight until all vertices in V are in the same set.

```
KRUSKAL-MST(G, w)
1  A ← ∅
2  for each vertex v ∈ V
3      do MAKE-SET(v)
4  Sort the edges of E by nondecreasing weight w
5  for each edge (u, v) ∈ E in order by nondecreading weight
6      do if FIND-SET(u) ≠ FIND-SET(v)
7           then A ← A ∪ {(u, v)}
8                UNION(u, v)
9  return A
```

- Complexity
  - The sorting of edges will take $O(|E| \log |E|)$ time.
  - Next for each edge, disjoint-set functions are called. This requires $O(|E| \log |V|)$ time.
  - Since $|E|$ is at most $|V|^2$ and $\log |V|^2 = 2 \log |V|$, $O(|E| \log |E|) = O(|E| \log |V|)$.
  - The running time for Kruskal's algorithm is $O(|E| \log |V|)$.

Kruskal's algorithm (cont...)

- Snapshots for each step of the Kruskal-MST function on a graph with nine vertices and 12 edges. Edges in A are marked using bold lines.



- Note that edges are considered in order of nondecreasing weights. Edges that link two vertices in the same component are ignored (Steps (e), (g), (k), and (l)). A vertex u is in the same component as v if there is a path from u to v using only the edges in A.

Prim's Algorithm

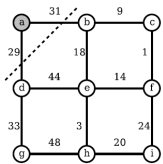
- Prim's algorithm has the property that the edges in A always form a single tree.
- The tree starts at an arbitrary vertex r and grows until it covers all the vertices in V.
- Let S be the vertex set of T so far, at each step we try to find an edge $(u, v) \in S \times (V - S) \cap E$ and the weight of the edge is the minimum one, add the edge into A and add v to S.

```
PRIM-MST(G, w, r)
 1   Q ← V
 2   for each u ∈ Q
 3       do key[u] ← ∞
 4   ikey[r] ← 0
 5   p[r] ← NIL
 6   while Q ≠ ∅
 7         do u ← EXTRACT-MIN(Q)
 8            for each v ∈ adj[u]
 9                do if v ∈ Q and w(u, v) < key[v]
10                      then p[v] ← u
11                           key[v] ← w(u, v)
```

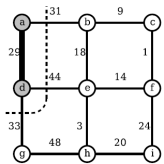
- The performance of Prim's algorithm depends on how the priority queue Q is implemented.

Prim's Algorithm (cont...)

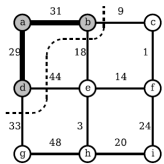
- Growing a MST using Prim-MST. At step (a), the root of the MST is initialised to vertex a.



(a) Add a

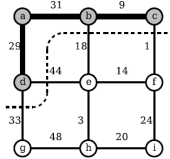


(b) Link to d via (a,d)

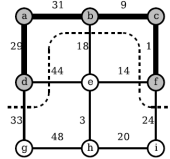


(c) Link to b via (a,b)

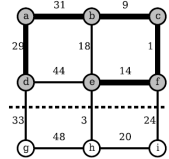
- At each intermediate step, the dotted-line indicates the cut dividing the vertices in the MST (shaded) and those not in the MST (not shaded). Notice that the MST is grown along light edge edges only.



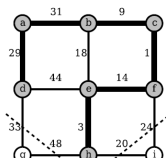
(d) Link to c via (b,c)



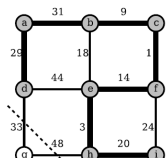
(e) Link to f via (c,f)



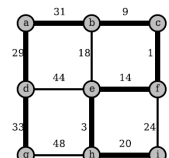
(f) Link to e via (f,e)



(g) Link to h via (e,h)



(h) Link to i via (h,i)



(i) Link to g via (d,g)

Summary