Unit 7: Graphs

A simple graph G is composed of a finite set of vertices and a finite set of edges. A graph is denoted by G. A graph is denoted as G = (V,E). The elements of V are the vertices and those of E are the edges. The vertex set of V are denoted by V_G and edge set is denoted by E_G. Thus G = (V_G, E_G). The number of vertices is denoted by |V| and number of edges is denoted by |E|.

A directed graph or digraph G = (V, E) consists of a nonempty set V of vertices and a set of edges where each edge is a pair of vertices from V. The difference is that one edge of a simple graph is of the form $\{v_i,v_j\}$ and for such an edge $\{v_i,v_j\} = \{v_j,v_i\}$. In digraph, the edge is of the form (v_i,v_j) and in this case, $(v_i,v_j)\neq(v_j,v_i)$. Unless necessary, this distinction in notation will be disregarded and edge between vertices v_i and v_j is will be referred to as edge (v_i,v_j).

A multigraph is a graph in which two vertices can be joined by multiple edges.

A path from v_i to v_n is a sequence of edges edge(v_1,v_2), edge(v_2,v_3),….(v_{n-1},v_n) and is denoted as path $v_1,v_2,v_3,\ldots\ldots\ldots\ldots..v_{n-1},v_n$. If $v_1 = v_n$ and no edge is repeated, then the path is called a circuit. If all vertices in a circuit are different, then it is called a cycle.

The graph is called a weighted graph if each edge has an assigned number. Depending on the context in which such graphs are used, the number assigned to an edge is called its weight, cost, distance, length or some other name.

A graph with n vertices is called complete and is denoted by K_n if for each pair of distinct vertices there is exactly one edge connecting them: that is, each vertex can be connected to any other vertex.

Graph Representation

There are various ways to represent a graph. A simple representation is given by an adjacency list which specifies all vertices adjacent to each vertex of the graph. This list can be implemented as table, in which case it is called a star representation which can be forward or reverse.

Another representation is a matrix, which comes in two forms: an adjacency matrix and an incidence matrix. A adjacency matrix of graph g = (V, E) is a binary |V|×|V| matrix such that each entry of this matrix:

aij = 1 if there exists an edge (vi,vj)

0 Otherwise.

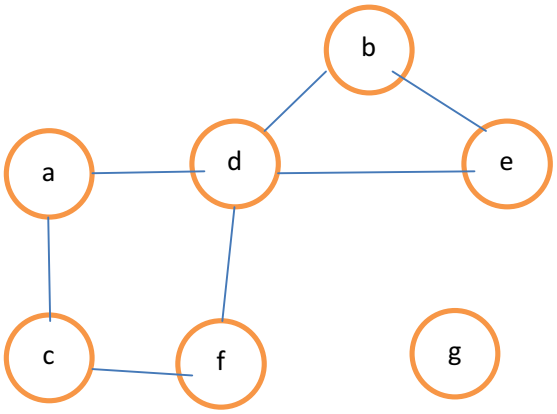
Generalization of this definition to multigraph is obtained by transforming the definition into the following form:

aij = number of edges between vi and vj.

Another matrix representation of a graph is based on the incidents of vertices and edges and is called an incidence matrix. An incidence matrix of graph G = (V, E) is a |V|×|E| matrix such that:

a_{ij} = 1 if edge e_j is incident with vertex vi

0 otherwise



Graph Traversal

As in the trees, traversing a graph consists of visiting each vertex only one time. The simple traversal algorithms used for trees cannot be applied because graphs may include cycles; hence tree traversal algorithms would result in infinite loops. To prevent that from happening, each visited vertex can be marked to avoid revisiting it. However, graphs can have isolated vertices, which means that some part of the graph are left out if unmodified tree traversal methods are applied.

There are two types of traversal.

Depth First Traversal: This algorithm was developed by John Hop croft and Robert Tarjan. In this algorithm, each vertex v is visited and then unvisited vertex adjacent to v is visited. If a vertex v has no adjacent vertices or all of its adjacent vertices have been visited, we backtrack to the predecessor of v. The traversal is finished if this visiting and backtracking process leads to the first vertex where the traversal started. If there are still some unvisited vertices in the graph, the traversal continues restarting for one of the unvisited vertices.

The algorithm assigns a unique number to each accessed vertex so that vertices are now renumbered.

```
DFS(v)
        num(v) = i++;
        for all vertices u adjacent to v
        if num(u) is 0
        attach edge (uv) to edges;
        DFS(u)
DepthFirstSearch ()
For all vertices v
num(v) = 0;
edges = null;
i=1;
While there is a vertex v such that num (v) is 0
```

DFS(v)
output edges;
The complexity of depth first search algorithm is O(|V|+|E|) because (a) initializing num(v) for each vertex requires |V| steps. (b) DFS(v) is called deg(v) times for each v- that is, once for each edge of v. Hence, the total number of calls is 2|E|. (c) searching for vertices as required by the statement

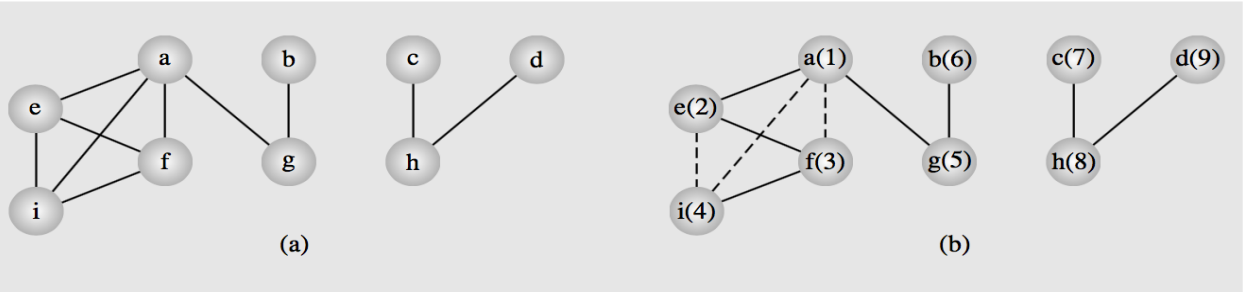
While there is a vertex v such that num (v) is 0

can be assumed to require |V| steps.

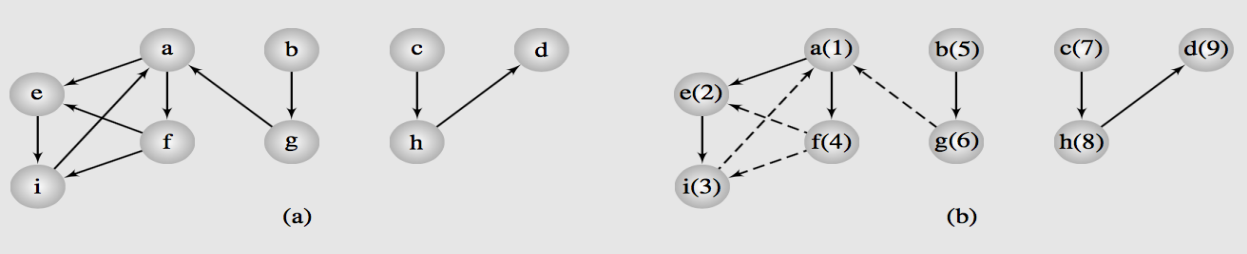
The complexity of depthFirstSearch () is O(|V|+|E|) because initializing num(v) for each vertex v requires |V| steps;

The depth first search of above graph is:

(a)Before DFS **(b) After DFS**



DFS for Digraph:



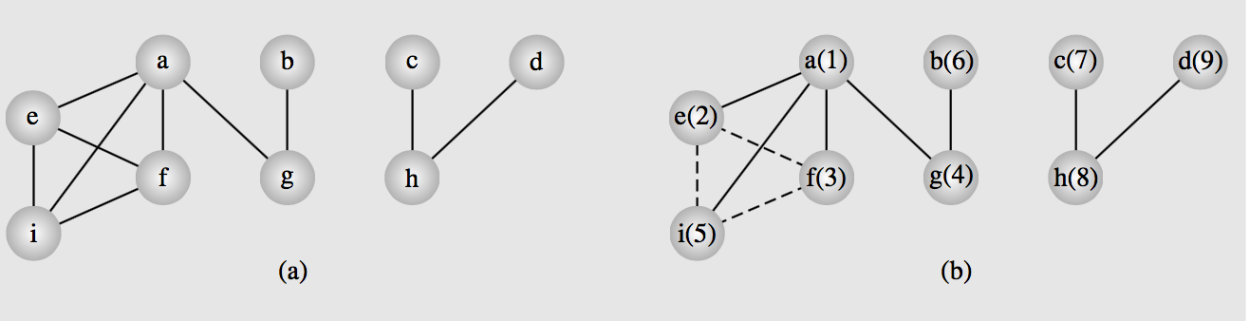
<u>Breadth First Search(BFS):</u>

breadthFirstSearch() first tries to mark all neighbors of a vertex *v* be- fore proceeding to other vertices, whereas DFS() picks one neighbor of a *v* and then proceeds to a neighbor of this neighbor before processing any other neighbors of *v.*

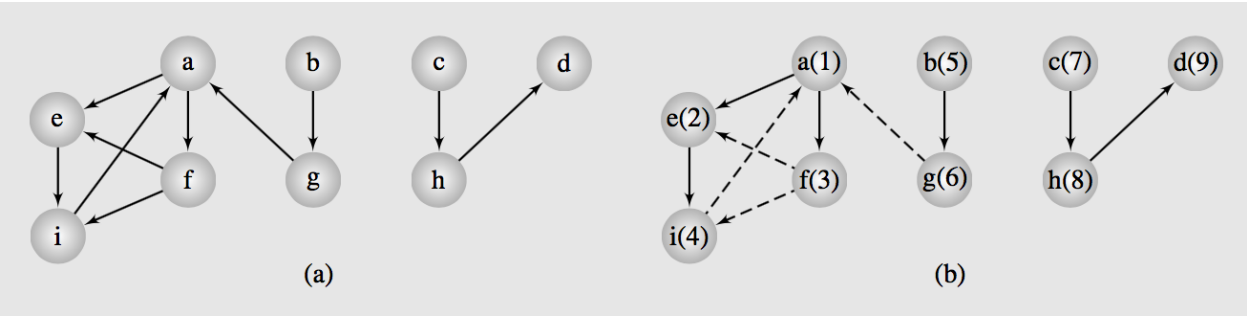
The breadth first search algorithm:

BreadthFirstSearch()
for all vertices u
num(u) = 0;
edge = null;
i = 1;
while(there is a vertex v such that num(v) ==0
num(v) = i++;

enqueue (v);
while queue is not empty
v = dequeue ()
for all vertices u adjacent to v
if num(u) is 0
num(u) = i++;
enqueue(u)
attach edges;



BFS for Digraph:



Shortest Paths

Finding the shortest path is a classical problem in graph theory and a large number of different solutions have been proposed. Edges are assigned certain weights representing, for example, distances between cities, times separating the execution of certain tasks, costs of transmitting information between locations. When determining the shortest path a from vertex v to u, information about distances between intermediate w has to be recorded. This information can be recorded as a label associated with these vertices, where the level is only the distance from v to w or the distance along with the predecessor of w in this path. The methods of finding the shortest path rely on these labels. Depending on how many times these labels are updated, the methods of solving the shortest path problem are divided into two classes: label setting methods and label-correcting methods.

For label setting methods, in each pass through the vertices still to be processed, one vertex is set to a value that remains unchanged to the end of the execution. This, however, limits such methods to processing graphs with only positive weights. The second category includes label correcting methods which allow for the changing of any label during application of the methods. The later method can be

applied to graphs negative weights and with no negative cycles- a cycle composed of edges with weights adding up to negative number- but they guarantee that, for all vertices, the current distances indicate the shortest path only after the processing of the graph is finished. Most of the label setting and label correcting methods, however, can be subsumed to the same form, which allows finding the shortest paths from one vertex to all other vertices.

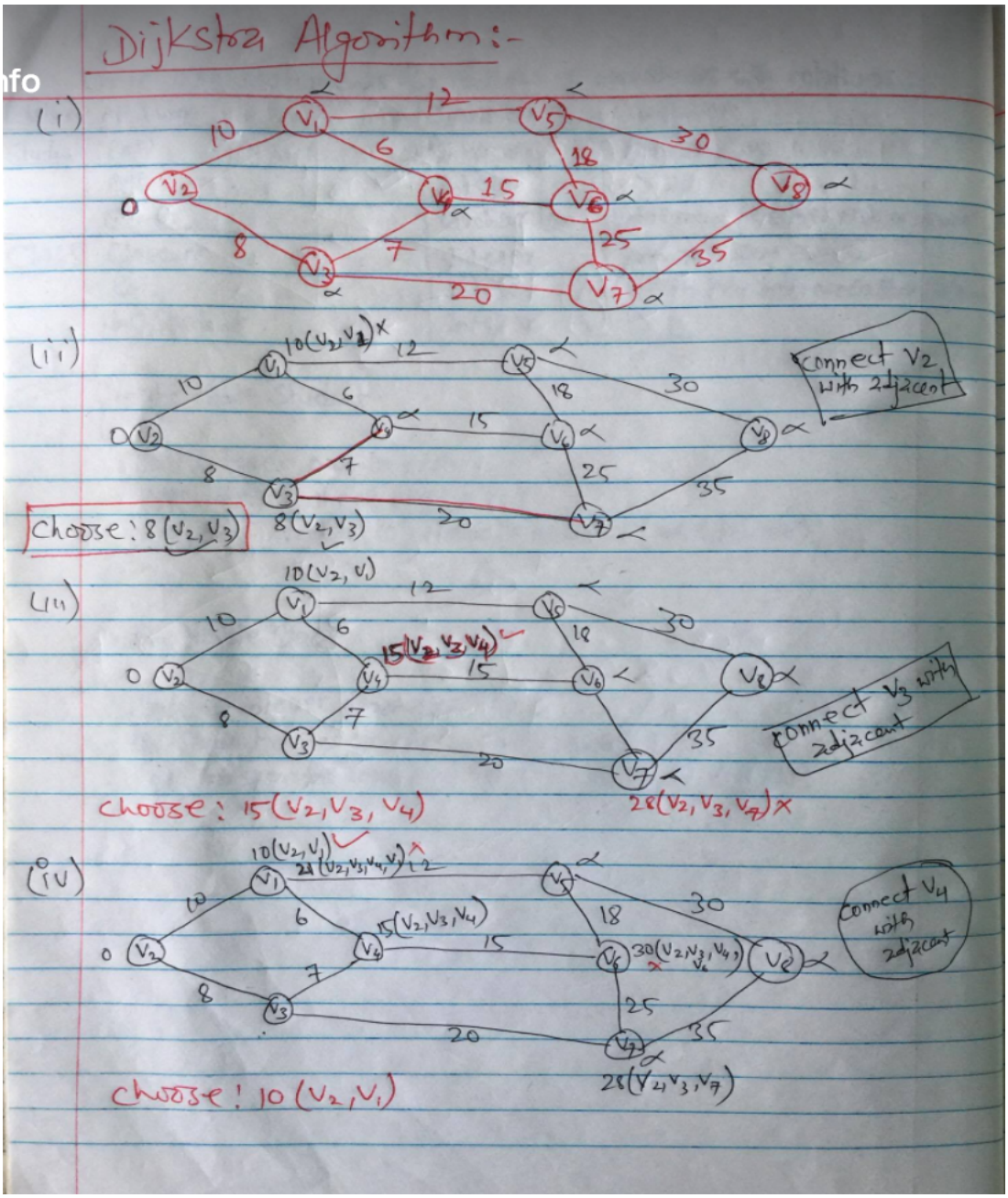
GenericShortestPathAlgorithm (weighted simple diagraph, vertex first)

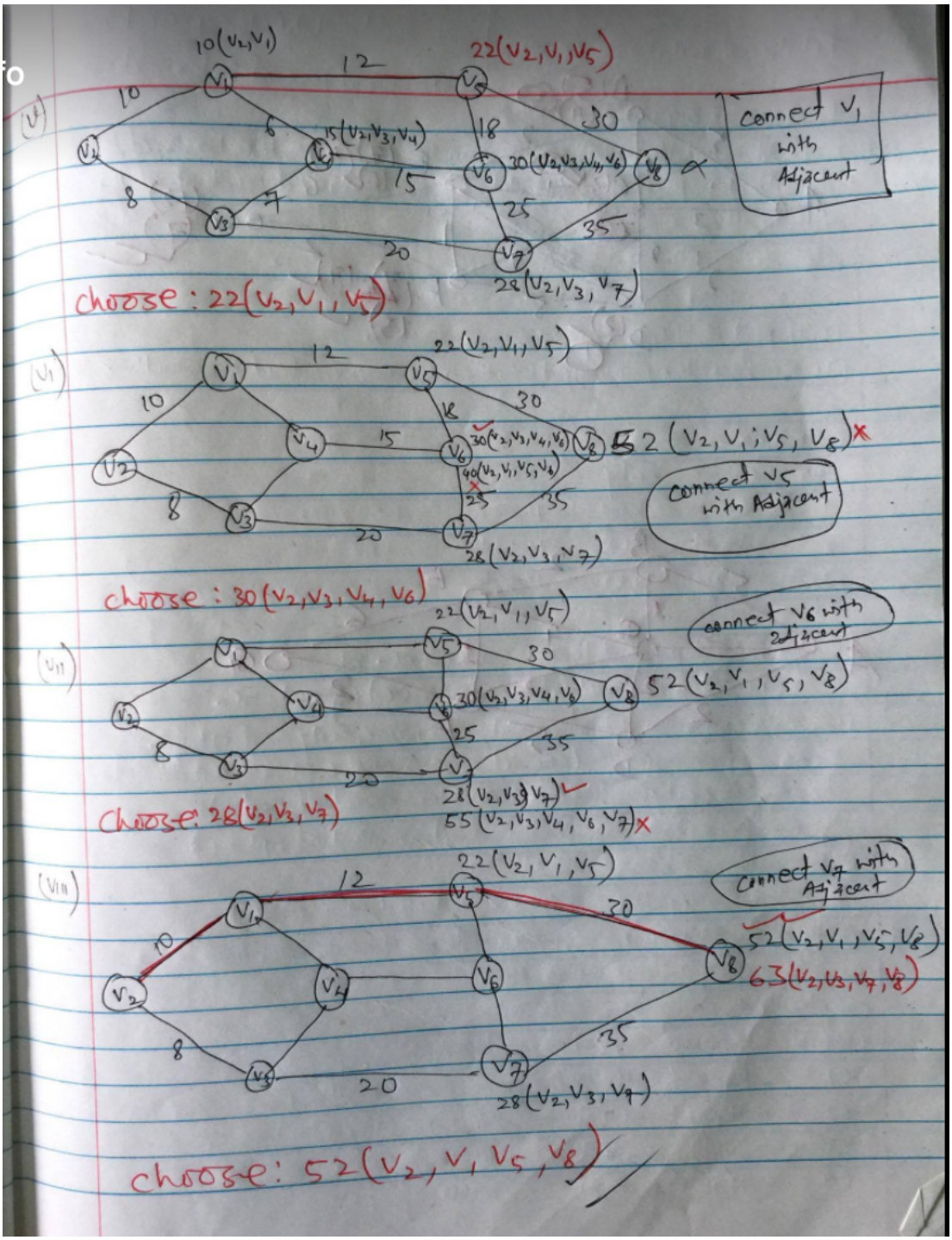
for all vertices v
currDist(v) = ∞
initialize toBeChecked;
while toBeChecked is not empty
v = a vertex in toBeChecked;
for all vertices u adjacent to v
if curDist(u)>curDist(v)+weight(edge(vu))
currDist(u) = currDist(v)+weight(edge(vu));
predecessor(u) = v;
add u to toBeChecked if it not there;

It should be clear that the organization of toBeChecked can determine the order of choosing new values for v, but it also determines the efficiency of the algorithm.
What distinguishes label-setting methods from label correcting methods in the method of choosing the value for v, always a vertex in toBeChecked with the smallest current distance. One of the first label setting algorithms was developed by Dijkstra.
In Dijkstra's algorithm, a number of paths p_1,p_2,…….. .. p_n from a vertex v are tried, and each time, the shortest path is chosen among them, which may mean that the same path pi can be continued by adding one more edge to it. But if pi turns to be longer than any other path that can be tried, pi is abandoned and this other path is tried by resuming from where it was left and by adding one more edge to it. Because paths can lead to vertices with more than one outgoing edge, new paths for possible exploration are added for each outgoing edge. Each vertex is tried once, all paths leading from it are opened, and the vertex itself is put away and not used anymore. After all vertices are visited, the algorithm is finished. Dijkstra's algorithm is as follows:
DijkatraAlgorithm (weighted simple diagraph, vertex first)
For all vertices v
CurDist (v) = o
ToBeChecked = all vertices
while toBeChecked is not empty
v = a vertex in toBeChecked with minimal currDist(v);
while toBeChecked is not empty
v = a vertex in toBeChecked and minimal currDist(v);
remove v from toBeChecked;
for all vertices u adjacent to v and in toBeChecked

if currDist(u)> currDist(v)+weight(edge(vu))
CurrDist (u) = currDist (v)+weight(edge(vu));
predecessor (u) = v;

Dijkstra's algorithm is obtained from the generic method by being more specific about which vertex is to be taken from toBeChecked so that the line
v = a vertex toBeChecked;
is replaced by the line
v = a vertex in toBeChecked with minimal curDist (v);




10(V2,V1)
22(V2,V1,V5)
connect V1 with Adjacent
choose: 22(V2,V1,V5)
22(V2,V1,V5)
52(V2,V1,V5,V8)
connect V5 with Adjacent
choose: 30(V2,V3,V4,V6)
22(V2,V1,V5)
connect V6 with Adjacent
52(V2,V1,V5,V8)
choose: 28(V2,V3,V7)
55(V2,V3,V4,V6,V7)
22(V2,V1,V5)
connect V7 with Adjacent
52(V2,V1,V5,V8)
63(V2,V3,V7,V8)
choose: 52(V2,V1,V5,V8)

All to All Shortest Path Problems

The task of finding all shortest paths from any vertex to any other vertex is more complex than the task of dealing with the one source only. But the method was designed by Stephen Warshall and implemented by Robert W Floyd and P, Z using adjacency matrix. The graph can include negative weights. The algorithm is as follows:

```
WFIalgorithm (matrix weight)
for i=1 to |V|
For j=1 to |V|
For k = 1 to |V|
If weight[j][k]> weight[j][k]+weight[i][k]
weight[j][k] = weight[j][i]+weight[i][k];
```

The outermost loop refers to vertices that may be on a path between the vertex with index j and the vertex k.

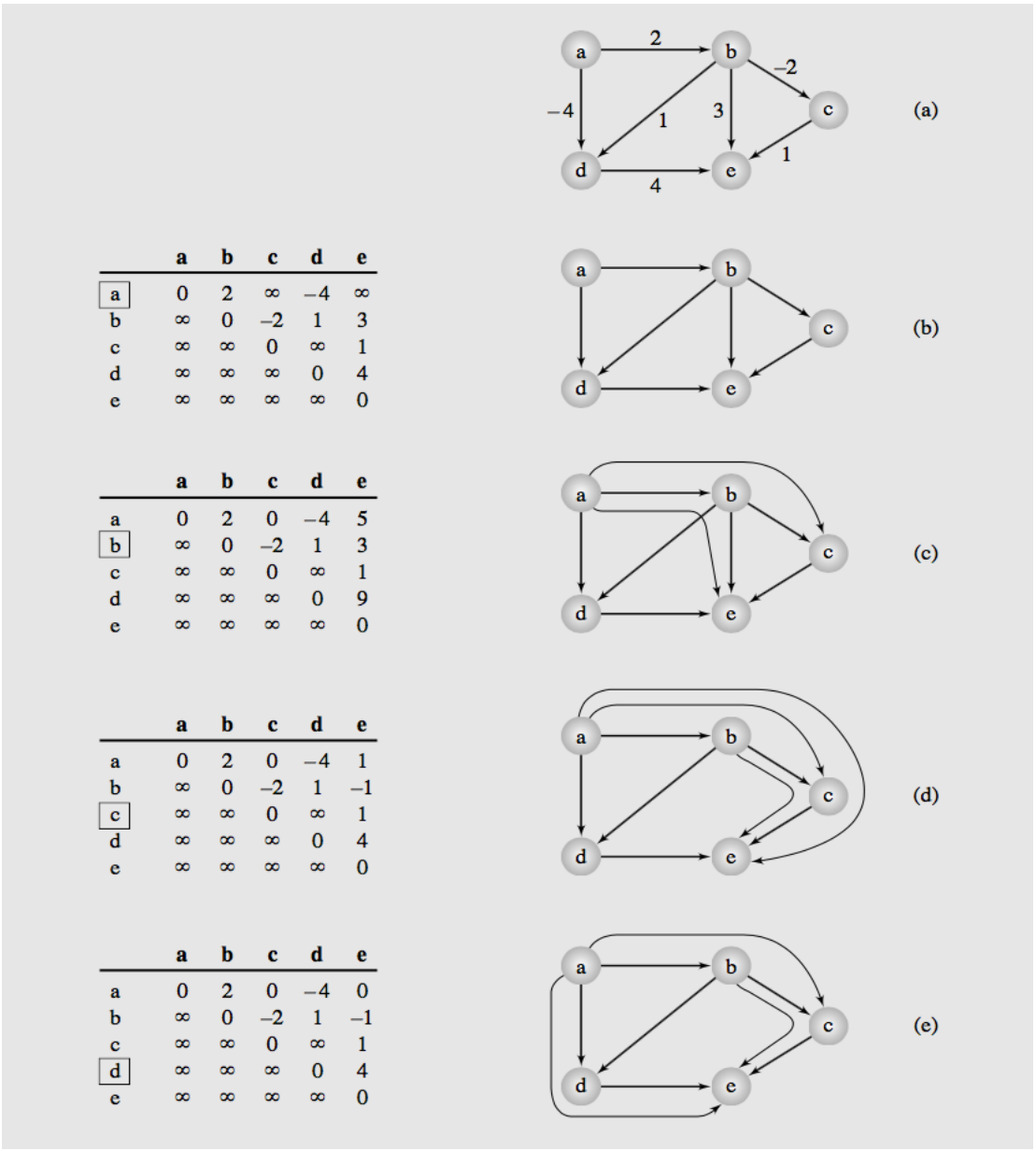
This algorithm also allows us to detect cycles if the diagonal is initialized to ∞ and not to zero. If any of the diagonal values are changed, the graph contains a cycle. Also, if an initial value of ∞ between two vertices in the vertex is not changed to a finite value, it is an indication that one vertex cannot be reached from another.

We can observe that, for any vertex v, the length of the shortest path to v is never greater than the length of the shortest path to any of its predecessors w plus the length of edge from w to v or

dist (v)≤dist(w) + weight(edge(wv))

For any vertices v and w. This inequality is equivalent to the inequality]

0≤weight' (edge (wv)) = weight (edge (vw)+dist(w)-dist(v))



	a	b	c	d	e
[a]	0	2	∞	−4	∞
b	∞	0	−2	1	3
c	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

	a	b	c	d	e
a	0	2	0	−4	5
[b]	∞	0	−2	1	3
c	∞	∞	0	∞	1
d	∞	∞	∞	0	9
e	∞	∞	∞	∞	0

	a	b	c	d	e
a	0	2	0	−4	1
b	∞	0	−2	1	−1
[c]	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

	a	b	c	d	e
a	0	2	0	−4	0
b	∞	0	−2	1	−1
c	∞	∞	0	∞	1
[d]	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

Hence, changing weight(e) to weight'€ for all edges e renders a graph with nonnegative edge weights. Now the shortest path v_1, v_2, v_3,……vk is

$$\sum_{i=0}^{k-1} \text{weight}'\big(\text{edge}(\text{vivi}+1)\big) = \sum_{i=0}^{i=k-1} \text{weight}\big(\text{edge}(\text{vivi}+1)\big) + \text{dist}(\text{vi}) - \text{dist}(\text{vk})$$

Therefore, if the weight L' of the path from v_1 to v_k is found in terms of nonnegative weights, then length L of the same path in the same graph using the original weights, some possibly negative, is L = L'- dist(v1) +dist(v_k).

Cycle Detection

Many algorithms rely on detecting cycles in graphs. We have just seen that as a side effect, WFIalgorithm () allows for detecting cycles in graphs. However, it is a cubic algorithm, which in many situations is too inefficient. Therefore, other cycles detection methods have to be explored.

One such algorithm is obtained directly from depthFirstSearch (). For undirected graphs, small modifications in DFS(v) are needed to detect cycles and report them

```
CycleDetectionDFS (v)
num(v) = i++;
for all vertices u adjacent to v
if num(u) is 0
pred(u) = v
cycleDetectionDFS (u);
else if u ≠pred(v)
pred(u) = v;
cycle detected;
```

For diagraphs, the situation is a bit more complicated, because there may be edges between different spanning subtrees, called side edges. An edge (a back edge) indicates a cycle if it joins two vertices already included in the same spanning subtree. To consider only this case, a number higher than any number generated in the subsequent searches is assigned to a vertex being currently visited after all its descendents have also been visited. In this way, if a vertex is about to be joined by an edge with a vertex having a lower number, we declare a cycle detection. The algorithm is now

```
DigraphCycleDetectionDFS (v)
num (v)  = i+1;
for all vertices u adjacent to v
if num(u) is 0 pred(u)  = v
digraphCycleDetectionDFS (u);
else if num(u) is not ∞
pred(u)=v
cycle detected;
num(v) = ∞ ;
```

Spanning Trees: Consider the graph representing the airline's connections between seven cities. If the economic situation forces this airline to shut down as many connections as possible, which of them should be retained to make sure that it is still possible to reach any city from any other city, if only indirected?

One popular algorithm was devised by Joseph Kruskal. In this, all edges are ordered by weight, and then each edge in this ordered sequence is checked to see whether it can be considered part of the tree under construction. It is added to the tree if no cycle arises after its inclusion. This simple algorithm can be summarized as follows:

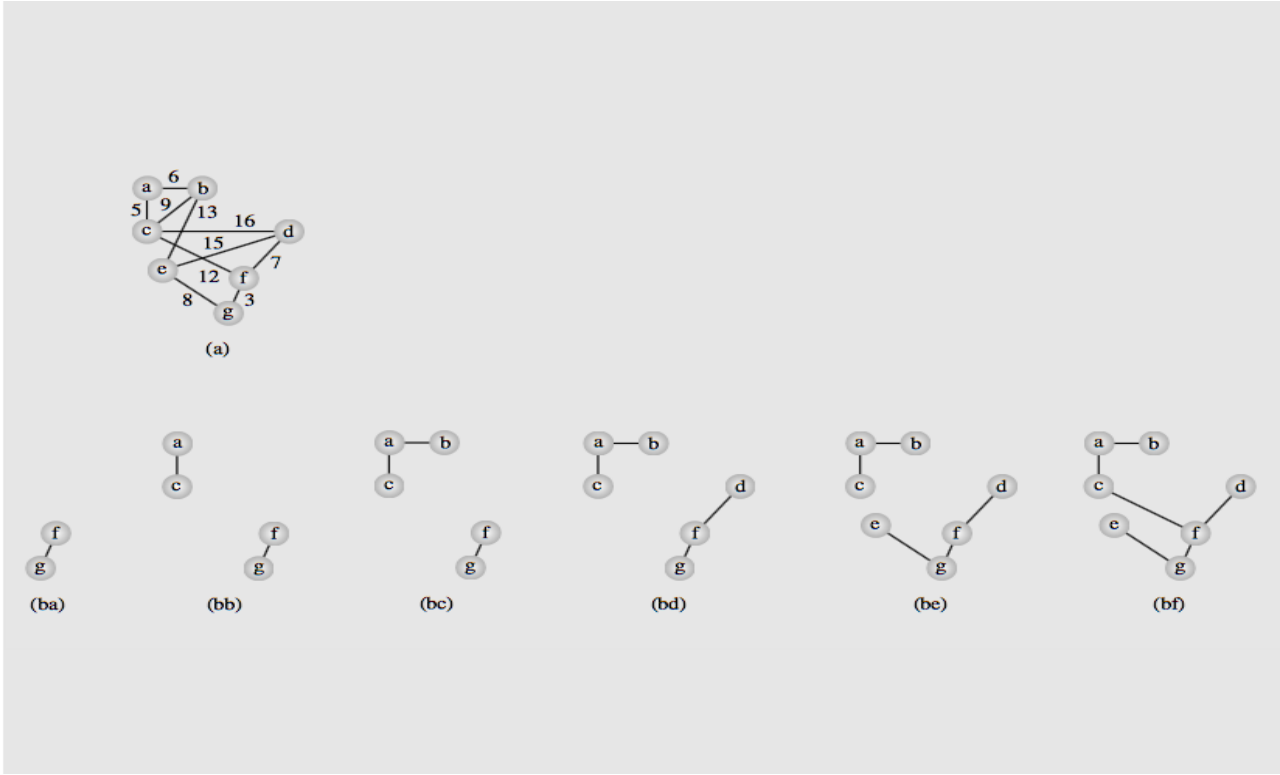
KruskalAlgorithm (weighted connected undirected graph)

tree = null;

edges = sequence of all edges of graph sorted by weight;

for (i=1; i≤|E| and |tree|<|V|-1; i++)

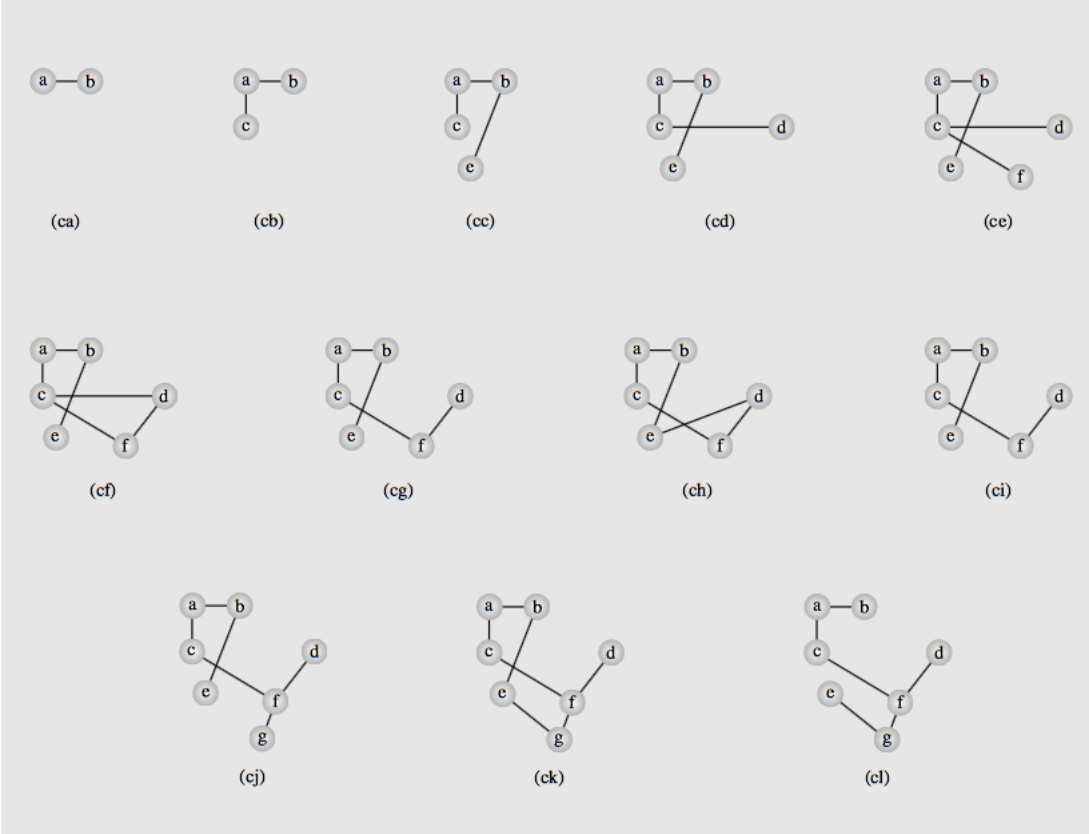
if e_i from edges does not form a cycle with edges in tree add e_i to tree;



The complexity of this algorithm is determined by the complexity of the sorting method applied, which for an efficient sorting is O(|E|log|E|). It also depends on the complexity of the method used for cycle detection.

Kruskal's algorithm requires that all the edges be ordered before beginning to build the spanning tree. Thus, however, is not necessary; it is necessary; it is possible to build a spanning tree by using any order of edges. A method was proposed by Dijkstra and independently by Robert Kalaba.

DijkstraMethod (weighted connected undirected graph)
tree = null;
edges = an undirected sequence of all edges of graph;
for j = 1 to |E|
and e_j to tree;
is there is a cycle in tree
remove an edge with maximum weight from this only cycle;



In this algorithm, the tree is being expanded by adding to it edges one by one, and if a cycle is detected, then an edge in this cycle with maximum weight is discarded.

Connectivity

In many situations, we are interested in finding a path in the graph from one vertex to any other vertex. For undirected graphs, this means that there are no separate pieces, or subgraphs, of the graph; for a digraph, it means that there are some places in the graph to which we can get from some directions but not necessarily able to return to the starting points.

Connectivity in Undirected Graphs

An undirected graph is called connected when there is a path between any two vertices of the graph. The depth first search algorithm can be used for recognizing whether a graph is connected provided that the loop heading

while there is a vertex v such that num (v) is 0

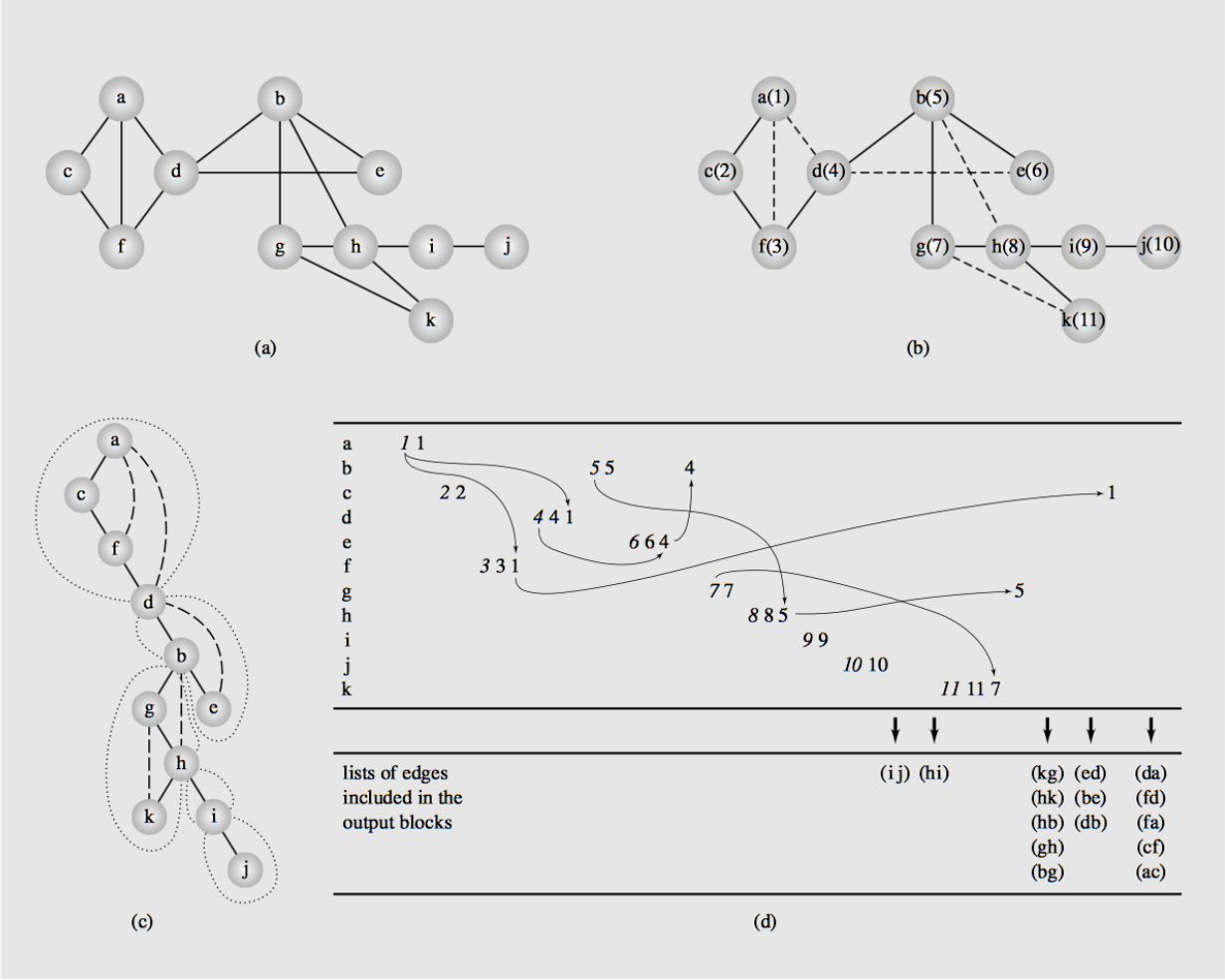
is removed. Then, after the algorithm is finished, we have to check whether the list edges include all vertices of the graph or simply check if i is equal to the number of vertices.

Connectivity comes in degrees. A graph can be more or less connected, and it depends on the number of different paths between its vertices. A graph is called n-connected if there are at least n different paths between any two vertices; that is, there are n paths between any two vertices that have no vertices in common.

A special type of graph is a *2-connected,* or *biconnected,* graph for which there are at least two nonoverlapping paths between any two vertices. A graph is not biconnected if a vertex can be found that always has to be included in the path between at least two vertices *a* and *b.* In other words, if this vertex is removed from the graph (along with incident edges), then there is no way to find a path from *a* to *b,* which means that the graph is split into two separate subgraphs. Such vertices are called *articulation points,* or *cut-vertices.*

If an edge causes a graph to be split into two subgraphs, it is called a *bridge* or *cut-edge.* Connected subgraphs with no articulation points or bridges are called *blocks,* or—when they include at least two vertices—*biconnected components.* It is important to know how to decompose a graph into biconnected components.

Articulation points can be detected by extending the depth-first search algorithm. This algorithm creates a tree with forward edges (the graph edges included in the tree) and back edges (the edges not included). A vertex *v* in this tree is an articulation point if it has at least one subtree unconnected with any of its predecessors by a back edge; because it is a tree, certainly none of *v*'s predecessors is reachable from any of its successors by a forward link. For example, the graph in Figure 8.16a is transformed into a depth-first search tree (Figure 8.16c), and this tree has four articulation points, *b, d, h,* and *i,* because there is no back edge from any node below *d* to any node above it in the tree, and no back edge from any vertex in the right subtree of *h* to any vertex above *h.* But vertex *g* cannot be an articulation point because its successor *h* is connected to a vertex above it. The four vertices divide the graph into the five blocks indicated in Figure 8.16c by dotted lines.



The algorithm uses a stack to store all currently processed edges. After an articulation point is identified, the edges corresponding to a block of the graph are output. The algorithm is as follows:

blockDFS(v)
pred(v) = num(v) = i++;
for all vertices u adjacent to v
if edge (vu) has not been processed
push(edge(vu));
if num(u) is 0;
blockDFS(u);
if pred(u)≥num(v)
e = pop()
while e/edge(vu)
output e;
e = pop();
output e;

```
else
pred(v) = min (pred(v), pred(u));
else if u is not the parent of v
pred(v)  = min(pred(v), num(u));

blockSearch ()
for all vertices v
num(v) = 0
i = 1;
While there is a vertex v such that num(v)==0
blockDFS (v);
```

Connectivity in Directed Graphs

For directed graphs, connectedness can be defined in two ways depending on whether or not the direction of the edges is taken into account. An directed graph is weakly connected if the undirected graph with the same vertices and same edges is connected. A directed graph is strongly connected if for each pair of vertices there is a path between them in both directions. The entire digraph is not always strongly connected, but it may be composed of strongly connected components (SCC), which are defined as subsets of vertices of the graph such that each of these subsets induces a strongly connected digraphs.

To determine SCCs. We also refer to depth first search. Let vertex v be the first vertex of an SCC for which depth first search is applied. Such a vertex is called the root of the SCC. Because each vertex u in this SCC is reachable from v, num(v) < num(u), and only after all such vertices u have been visited, the depth first search backtracks to v. In this case, which is recognized by the fact that pred(v) = num(v), the SCC accessible from the root that can be output.

The problem now is how to find all such roots of the digraph, which is analogous to finding articulation points in an undirected graph. To that end, the parameter pred (v) is also used, where pred(v) is the lower number chosen out of num(v) and pred(u), where u is a vertex reachable from v and belonging to the same SCC as v.

The algorithm attributed to Tarjan is as follows:

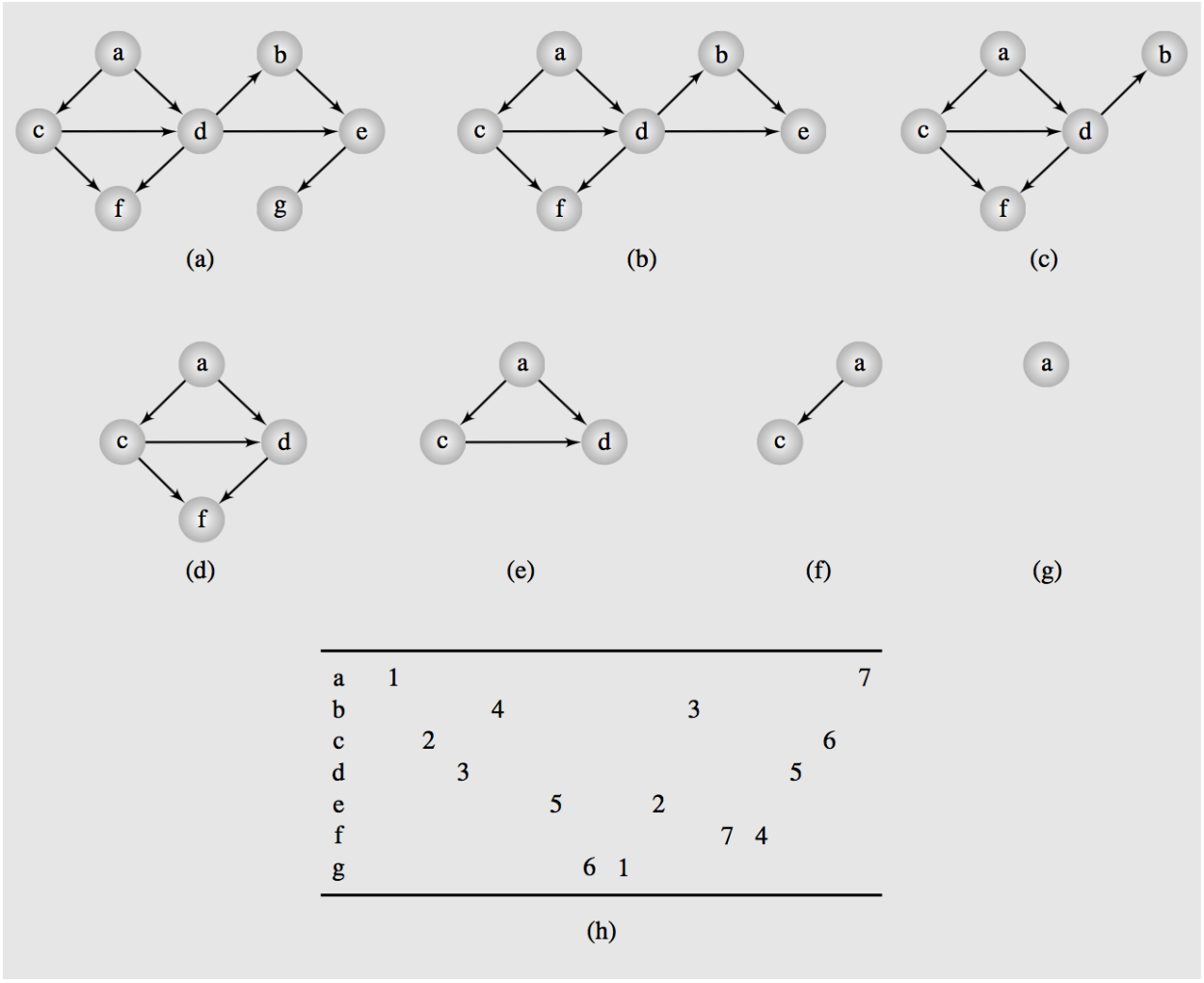
```
strongDFS (v)
pred(v) = num(v) = i++;
push(v);
for all vertices u adjacent to v
if num(u) is 0
strongDFS(u)
pred(v) = min(pred(v),num(u));
if pred(v) == num(v)
w = pop();
while w≠v
output w
w = pop();  output w;
```

```
strongConnectedComponentSearch ()
for all vertices v
num(v) = 0;
i = 1;
while there is a vertex v such that num(v)==0
strongDFS(v);
```

Topological Sort

In many situations, there is a set of tasks to be performed. For some pairs of tasks, it matters which task is performed first, where for other pairs, the order of execution is unimportant.
The dependencies between tasks can be shown in the form of a diagraph. A topological sort linearizes a digraph; that is, it labels all its vertices with numbers 1 ……..|V| so that i<j only if there is a path from vertex vi to vertex vj. The diagraph must not include a cycle; otherwise, a topological sort is impossible.
The algorithm for topological sort is rather simple. We have to find a vertex v with no outgoing edges, called a sink or a minimal vertex, and then disregard all edges leading from any vertex to v. The summary of the topological sort algorithm is as follows:

topologicalSort(digraph)
for i=1 to |V|
find a minimal vertex v;
num(v)=i;
remove from diagraph vertex v and all edges incident with v



a	1						7
b		4			3		
c	2					6	
d		3				5	
e			5	2			
f					7	4	
g				6	1		

(h)

Actually, it is not necessary to remove the vertices and edges from the digraph while it is processed if it can be ascertained that all successors of the vertex being processed have already been processed, so they can be considered as deleted. And once again, depth first search comes to the rescue. By the nature of this method, if the search backtracks to a vertex v, then all successors of v can be assumed to have already been searched. Here is how depth first search can be adapted to topological sort:

```
TS(v)
num(v) = i++;
for all vertices u adjacent to v
if num(u)==0
TS(u);
else if TSNum(u)==0
error
TSNum(v) = j++;
topologcalSorting(digraph)
for all vertices v
num(v) = TSNum(v)=0;
i = j= 1;
while there is a vertex v such that num (v) ==0
TS(v);
Output vertices according to their TSNum's
```

Networks

Maximum Flows

An important type of graph is a network. A network can be exemplified by a network of pipelines used to deliver water from one source to one destination. However, water is not simply pumped through one pipe but through many pipes with many pumping stations in between. The pipes are of different diameters and the stations of different power so that the amount of water that can be pumped may differ from one pipeline to another

Let us consider the following pipeline with eight pipes and six pumping stations.

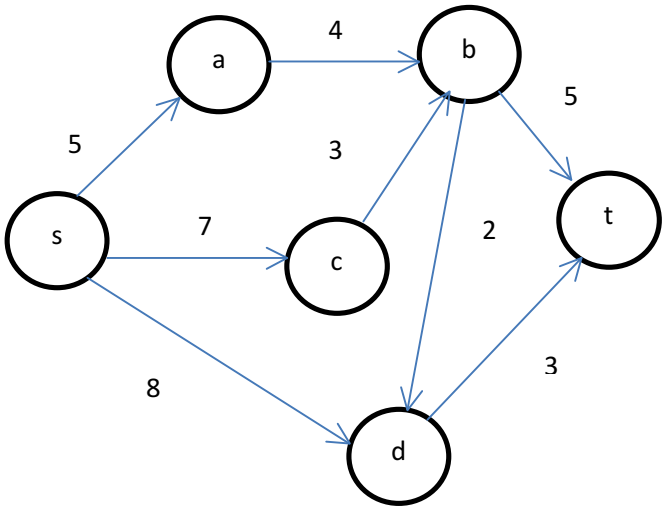


Figure: A pipeline with eight pipes and six pumping stations.

The network has eight pipes and six pumping stations. The numbers shown in the figure are the maximum capacities of each pipeline. The problem is to maximize the capacity of the entire network so that it can transfer the maximum amount of water. It may not be obvious how to accomplish this goal. It is to be noted that we cannot put 5 units through pipe sa, because pipe ab cannot transfer it. Also, the amount of water coming to station b has to be controlled as well because if incoming pipes, ab and cb are used to full capacity, then the outgoing pipe, bt, cannot process it either. It is far from obvious, especially for large networks, what the amounts of water put through each pipe should be to utilize the network maximally. Computational analysis of this particular network problem was initialized by Leser R. Ford and D ray Fulkerson. Since their work, scores of algorithms have been published to solve this problem.

A network is a diagraph with one vertex s, called the source, with no incoming edges, and one vertex t, called sink, with no outgoing. With each edge e we associate a number cap(e) called capacity of edge. A flow is a real function f:E→R that assigns a number to each edge of the network and meets these two conditions:

1. The flow through an edge e cannot be greater than its capacity , or $0 \leq f(e) \leq cap\ (e)$
2. The total flow coming to a vertex v is the same as the total flow coming from it. Or $\Sigma_u f(edge(uv)) = \Sigma_w f(edge(vw)$. Where v is neither the source nor the sink.

The problem now is to maximize the flow f so that the sum $\Sigma\ f_u(edge(ut))$ has a maximum value for any possible function f. This is called a maximum flow problem.