Chapter 19

Graphs

(Revision Date: 16 November 1997)

19.1 Graph Basics

19.1.1 Definitions

◇ **Definition**: A *graph* $G = (V, E)$ consists of a finite set of *vertices*, V and a set of *edges*, E. Each edge is a pair (v, w), where $v, w \in V$.

• V and E are *sets*, so each vertex $v \in V$ is unique, and each edge $e \in E$ is unique.

• Edges are sometimes called *arcs* or *lines*.

• Vertices are sometimes called *nodes* or *points*.

◇ **Definition**: A *directed graph* is a graph in which the edges are ordered pairs. That is, $(u, v) \neq (v, u)$, $u, v \in E$. Directed graphs are sometimes called *digraphs*.

◇ **Definition**: An *undirected graph* is a graph in which the edges are unordered pairs. That is, $(u, v) = (v, u)$.

• Since E is a set, an undirected graph cannot have both edge (u, v) and edge (v, u), $u, v \in E$. They would not be unique. (An undirected graph-type data structure which allows such edges is called a *multi-graph*).

• In a directed graph, $(u, v) \neq (v, u)$, $u, v \in E$. Therefore, both edges (u, v) and (v, u) may be in a directed graph.

• Graphs are often shown pictorially. The vertices are represented as small circles or dots. The edges are represented as lines. Figure 19.1 shows two graphs of 5 vertices. One of the graphs is a directed graph, the other is undirected. The vertices are labelled just for identification purposes. The edges are unlabelled.

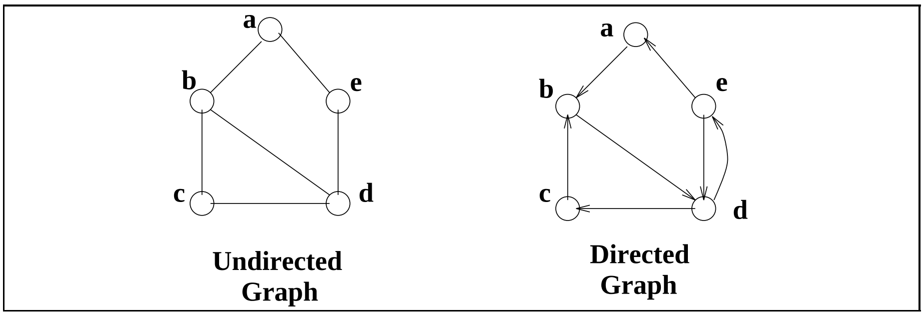


Figure 19.1: Two Simple Graphs (Directed and Undirected)

◇ **Definition**: Vertex v is *adjacent to* vertex w if and only if $(v, w) \in E$.

◇ **Definition**: Vertex v is *adjacent from* vertex w if and only if $(w, v) \in E$.

Note: It can be important to distinguish between *adjacent to* and *adjacent from*. In an undirected graph, it doesn't matter (since $(u, v) = (v, u)$, $u, v \in E$). In a directed graph it does matter and the distinction between *from* and *to* is important.

EXAMPLE: In the undirected graph of Figure 19.1, vertex `a` is adjacent to vertex `b`; but is not adjacent to vertex `d`. In the directed graph, vertex `a` is adjacent to vertex `b`; is adjacent from vertex `e`; but is not adjacent to or from vertex `d`.

• An edge in a graph may also have other values associated with it. These values are commonly called *weight* or *cost*. Edges may also be *labelled* (given a unique name).

• The *degree* of a vertex u in an undirected graph is the number of vertices adjacent to u. Degree is sometimes called *valence*.

• The *in-degree* (*outdegree*) of a vertex u in a directed graph is the number of vertices adjacent to (from) u.

EXAMPLE: In the undirected graph of Figure 19.1, vertex `a` has degree 2. In the directed graph, vertex `a` has indegree 1 and outdegree 1; vertex `b` has indegree 2 and outdegree 1.

19.1.2 Paths in Graphs

◇ **Definition**: A *path* in a graph is a sequence of vertices $w_1, w_2, w_3, \ldots, w_n$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < n$.

◇ **Definition**: The *length* of a path in a graph is the number of edges on the path.

- The length of the path from a vertex to itself is 0.

◇ **Definition**: A *simple path* is a path such that all vertices are distinct, except that the first and last may be the same.

EXAMPLE: In the undirected graph of Figure 19.1, there are two simple paths of length 2 and one simple path of length 3 from vertex `a` to vertex `d`. The paths of length 2 are a, b, d and a, e, d. The path of length 3 is a, b, c, d.

EXAMPLE: In the directed graph of Figure 19.1, there is one simple path from `a` to `d`, namely the path a, b, d. The length of this path is 2.

◇ **Definition**: A *cycle* in a graph is a path $w_1, w_2, \cdots, w_n$, $w \in V$ such that:

1. there are at least two vertices on the path (the path length is at least equal to 1), and
2. $w_1 = w_n$ (the path starts and ends on the same vertex), and
3. if any part of the path contains the sub-path w_i, w_j, w_i, then each of the edges in the subpath is distinct, (no doubling back along the same edge is allowed).

◇ **Definition**: A *simple* cycle is one in which the path is simple.

EXAMPLE: Some simple cycles in the undirected graph of Figure 19.1 are a, b, d, e, a and b, d, c, b. The cycle a, b, c, d, b, a is an example of a cycle which is not simple because not all the vertices are distinct (b appears twice).

EXAMPLE: Some simple cycles in the directed graph of Figure 19.1 are a, b, d, e, a and b, d, c, b. The cycle b, d, c, b, d, e, a, b is an example of a cycle which is not simple because not all the vertices are distinct (b and d are repeated at other than the endpoints).

◇ **Definition**: An undirected graph is *connected* if there is a path from every vertex to every other vertex.

◇ **Definition**: A directed graph is *strongly connected* if there is a path from every vertex to every other vertex.

◇ **Definition**: A directed graph is *weakly connected* if there would be a path from every vertex to every other vertex, disregarding the direction of the edges.

◇ **Definition**: A *complete* graph is one in which there is an edge between every pair of vertices.

◇ **Definition**: A *connected component* of a graph is any maximal connected subgraph. Connected components are sometimes simply called *components*.

- A connected graph has just one component. A disconnected graph has at least two components.

◇ **Definition**: For digraphs, a *strongly-connected component* is a component that is strongly connected. Similarly for *weakly-connected component*.

19.2 Some Simple Graph Theorems

The following theorem, due to Euler, is perhaps the very first graph theorem:
Theorem 19.1 Let $G = (V, E)$ be an undirected graph. Let $v1, v2, \ldots, v_p \in V$ be the vertices in G and let $q = |E|$ be the number of edges in E. Let $D(v)$ be the degree of vertex v. Then,

$$\sum_{i=1}^{p} D(v_i) = 2q$$

(the sum of the degrees of all the vertices is twice the number of edges).

▷ **Proof**: Since every edge is incident on two vertices, each edge contributes 2 to the sum of the degrees of the vertices. ◁

A corollary of Theorem 19.1 is:
Theorem 19.2 In any undirected graph, the number of vertices of odd degree is even.

▷ **Proof**: Let $S = D(v_1) + D(v_2) + \ldots + D(v_m)$, where $v1, \ldots, v_m$ have even degree. Let $T = D(v_{m+1}) + D(v_{m+2}) + \ldots + D(v_p)$ where $v_{m+1}, \ldots, v_p$ have odd degree.
From Theorem 19.1, $S + T = 2q$ and $2q$ is even. But, S is even since it is the sum of even numbers. Therefore, T must also be even (even + even = even). Therefore, T must contain an even number of terms (each term in T is odd, but T is even; an odd number of odds would be odd, an even number of odds is even). ◁

19.3 A Graph ADT

We can define a graph ADT as:

1. the object is a graph, and
2. for any graph $G = (V, E)$ with vertices $u, v \in V$ the operations are:
   - `getDegree(u)` returns the degree of vertex `u` (undirected graph).
   - `getInDegree(u)` returns the in-degree of vertex `u` (directed graph).
   - `getOutDegree(u)` returns the out-degree of vertex `u` (directed graph).
   - `getAdjacentTo(u)` returns a list of the vertices adjacent to vertex `u` (directed and undirected graphs).

- `getAdjacentFrom(u)` returns a list of the vertices adjacent from vertex `u` (directed graph).
- `isConnected(u,v)` returns TRUE if vertices `u` and `v` are connected, FALSE otherwise (directed and undirected graphs).
- `doInsert(u)` inserts vertex `u` and connects it to current vertex with an edge from `u`.
- `doDelete()` deletes the current vertex and removes any edges impinging on it. It's an error to attempt to delete a non-existent vertex.
- `getSize()` returns the number of vertices in the graph.

19.4 Graph Traversals

• As with trees, graphs can be traversed breadth-first or depth-first. Recall that breadth-first tree traversal involved a queue and depth-first traversal involved a stack.

• In tree traversals, there was no danger of repeating a path because there are no cycles in a tree. This is not true for graph traversals.

• To avoid going over the same path multiple times, we mark each vertex as "visited" when we first encounter it. We do not consider visited vertices more than once.

Compare the following breadth-first and depth-first graph traversals to the corresponding traversals for binary trees.

Breadth-First:

```
Queue q = new Queue();
graphvertex u;

q.enqueue(startvertex); // Choose any vertex to start with
markVisited(startvertex)
while (!q.isEmpty())
 {
   u = q.dequeue();
   for (each vertex w adjacent to u)
      if (w is not marked as visited)
        {
          markVisited(w);
          q.enqueue(w);
        }
 }
```

Depth-First:

```
Stack s = new Stack();
GraphVertex u;
// Choose any vertex to start with
GraphVertex startvertex = graph.getStartVertex();

s.push(startvertex);
markVisited(startvertex)
while (!s.isEmpty())
 {
   u = s.Pop();
   for (each vertex w adjacent to u)
      if (w is not marked as visited)
        {
          markVisited(w);
          s.push(w);
        }
 }
```

EXAMPLE: For the graph of Figure 19.2,a breadth-first order of visit, starting from vertex `1` is `1 2 3 4`. A depth-first order of visit, starting from vertex `1` is `1 2 4 3`. Note that in neither case is vertex `5` visited.

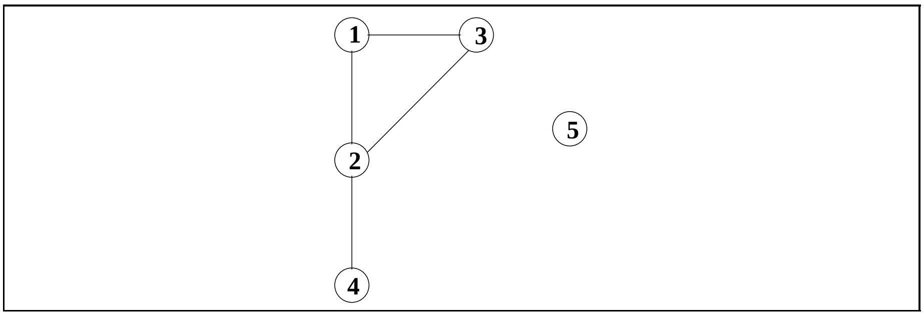


Figure 19.2: Example Graph for DF and BF Traversals

- Many graph algorithms assign numbers to the vertices. The numbers are usually assigned in arbitrary order.

- Each vertex is visited at most once.

- The edges traversed form a tree which includes all the visited vertices in the graph. This tree is called a *spanning tree*.

- For connected graphs, depth-first and breadth-first traversals will visit every vertex, starting from any vertex. In a graph which is not connected (such as Figure 19.2), visitation of every vertex is guaranteed only if a traversal is done starting from each vertex.

▷ **QUESTION**:

What is the performance of df and bf traversal?

▷ **ANSWER**:

Each vertex appears in the stack or queue exactly once. Therefore, the traversals are at least $O(|V|)$. However, at each vertex, we must find the adjacent vertices. Therefore, df- and bf-traversal performance depends on the performance of the `getAdjacentTo` operation.

19.4.1 The `getAdjacentTo` operation

There are two ways to implement the `getAdjacentTo(u)` operation:

1. Look at every vertex (except `u`), asking "are you an adjacent node of `u`?"

```
List L = new List(<class of vertex>);
for (each vertex v, except u)
  if (isConnected(u,v))
     L.doInsert(v);
```

   - As will be seen, `isConnected` is in $O(1)$ for the adjacency table representation, so `getAdjacentTo` is in $O(|V|)$.

2. Look only at the edges which impinge on `u`. Therefore, at each vertex, the number of vertices to be looked at is $D(u)$, the degree of the vertex. This approach is $O(D(u))$.

• Using the first approach, since `getAdjacentTo` is done $O(|V|)$ times, the performance of df and bf traversal is in $O(|V|^2)$.

• Using the second approach, the performance of df and bf traversal is in $O(\sum_{i=1}^{|V|} D(v_i)) = O(2q) = O(|E|)$ (see Theorem 19.1). However, in a disconnected graph we must still look at every vertex, so the performance of df and bf traversals is $\mathtt{max}(O(|V|), O(|E|)) = O(|V| + |E|)$.

Theorem 19.3 The number of edges in an undirected graph $G = (V, E)$ is $O(|V|^2)$

▷ **Proof**: Suppose G is fully connected, so there is an edge between each pair of vertices. Let $p = |V|$. We have the following situation:

vertex	connected to
1	$2, 3, 4, \ldots, p$
2	$1, 3, 4, \ldots, p$
...	...
p	$1, 2, 3, 4, \ldots, (p-1)$

There are $\frac{p(p-1)}{2} = O(|V|^2)$ edges.

◁

• In a directed graph, the result is $p(p-1) = O(|V|^2)$ edges.

• In any undirected graph, $0 \leq |E| \leq |V|^2$.

• A "typical" undirected graph has more edges than vertices, so, generally, $|V| < |E| < |V|^2$.

• The second approach to the operation `getAdjacentTo` causes df and bf traversal to be $O(|V| + |E|)$. Since from Theorem 19.3, $|E| = O(|V|^2)$, df and bf traversal is $O(|V|^2)$ under both approaches.

▷ **QUESTION**:

If df and bf traversal is $O(|V|^2)$ under both approaches to the operation `getAdjacentTo`, does it matter which approach is used?

▷ **ANSWER**:

Yes. Although the two approaches have the same asymptotic performance, the second approach will be faster for the typical graph.

19.5 Graph Implementations

The two basic methods for implementing graphs are the *adjacency table* and the *adjacency list* methods.

19.5.1 Adjacency Table

The adjacency table implementation (also known as *adjacency matrix*) uses a table of size $|V| \times |V|$. Each entry (i, j) in the table is boolean – TRUE if there is an edge from vertex i to vertex j, FALSE otherwise.

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

EXAMPLE: Here is an undirected graph and its adjacency table:

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

EXAMPLE: Here is a directed graph and its adjacency table:

• When edges are weighted, store the weights in the table. Use ∞ for "no edge."

• Note that for undirected graphs, the adjacency table is symmetric. It's generally not symmetric for directed graphs.

19.5.2 Analysis of Adjacency Tables

• The storage requirement is $O(|V|^2)$.

The performance of the ADT operations on graphs implemented by adjacency tables is:

`getDegree(u)`	$O(\|V\|)$	Must look at $(p-1)$ elements on the u-th row.
`getInDegree(u)`	$O(\|V\|)$	Must look down the u-th column.
`getOutDegree(u)`	$O(\|V\|)$	Must look across the u-th row.
`getAdjacentTo(u)`	$O(\|V\|)$	Must look across the u-th row.
`getAdjacentFrom(u)`	$O(\|V\|)$	Must look down the u-th column.
`isConnected(u,v)`	$O(1)$	Random access to element $[u,v]$.

19.5.3 Adjacency List

In the adjacency list methods, a list of adjacent vertices is kept for each vertex. Figure 19.3 shows some of the many variations on this theme.

Array of Lists of Indices: An array of lists of indices. Each element of `array[i]` is a list of the indices of the vertices adjacent to vertex `i`.

List of Lists: Let `L` be a list. The `i`-th element in `L` is associated with vertex `Vi` and is a List `Li` of the elements of `L` adjacent to `Vi`. See the figure.

Lists in Array (`NIL` sentinels): Each entry `a[i,j]` is either the index of the `j`-th vertex adjacent to vertex `i` or a `NIL` sentinel indicating end-of-list.

Lists in Array (with valence array): Instead of using `NIL` sentinels to mark the end of the list in the array, a separate array `Valence` is kept indicating the number of entries in each row of the array.

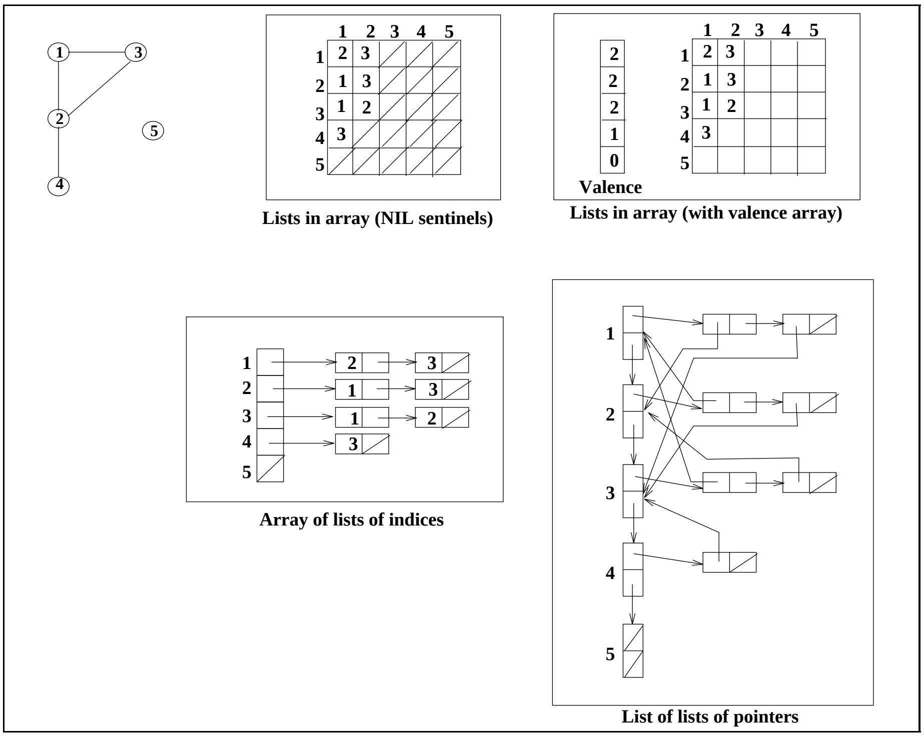


Figure 19.3: A Variety of Adjacency Lists for the Graph Shown

19.5.4 Analysis of Adjacency Lists

- The storage requirement is $O(|V| + |E|)$.

The performance of the ADT operations on graphs implemented by adjacency lists is (note $D(u)$ is the degree of vertex u):

	Array of Lists of Indices	List of Lists of Pointers	Lists in Array (`NIL` sentinels)	Lists in Array (with `valence`)
`getDegree(u)`	$O(D(u))$	$O(D(u))$	$O(D(u))$	$O(1)$
`getInDegree(u)`	$O(\|V\|+\|E\|)$	$O(\|V\|+\|E\|)$	$O(\|V\|+\|E\|)$	$O(\|V\|+\|E\|)$
`getOutDegree(u)`	$O(D(u))$	$O(D(u))$	$O(D(u))$	$O(1)$
`getAdjacentTo(u)`	$O(D(u))$	$O(D(u))$	$O(D(u))$	$O(D(u))$
`getAdjacentFrom(u)`	$O(\|V\|+\|E\|)$	$O(\|V\|+\|E\|)$	$O(\|V\|^2)$	$O(\|V\|^2)$
`isConnected(u,v)`	$O(D(u))$	$O(D(u))$	$O(D(u))$	$O(D(u))$
`Storage`	$O(\|V\|+\|E\|)$	$O(\|V\|+\|E\|)$	$O(\|V\|^2)$	$O(\|V\|^2)$

• `getDegree(u)` requires counting list length in each case except "`Lists in Array (with valence)`."

• `getInDegree(u)` requires searching each list for the presence of vertex `u`.

• `getOutDegree(u)` requires counting list length in each case except "`Lists in Array (with valence)`."

• `getAdjacentTo(u)` requires construction of a list of each vertex on the adjacency list of vertex `u`. Note that if just the list is to be returned, this can be considered to be $O(1)$.

• `getAdjacentFrom(u)` requires construction of a list of vertices by searching the adjacency lists of each vertex.

• `isConnected(u,v)` requires searching the adjacency list of vertex `u` for the presence of vertex `v`.

• `Storage` for the "Lists in Array" requires an entry (even if `NIL`) for each vertex and is therefore $O(|V|^2)$. `Storage` for the other methods requires an entry for each vertex and a list of entries for each adjacent vertex. No list entry is needed for `NIL` entries. Pointer storage is required. $O(|V|+|E|)$

19.6 Directed Acyclic Graphs (DAG)

◇ **Definition**: A *directed acyclic graph* is a directed graph with no cycles.

A major use for DAGs is representation of partial orders.

◇ **Definition**: A *partial order* R on a set S is a binary relation such that

1. for all $a \in S$, aRa is false (irreflexive property)
2. for all $a, b, c \in S$, if aRb and bRc then aRc is true (transitive property).

To represent a partial order with a DAG, follow these steps:

1. represent each member of S as a vertex.
2. for each pair of vertices (a, b), insert an edge from a to b if and only if aRb.

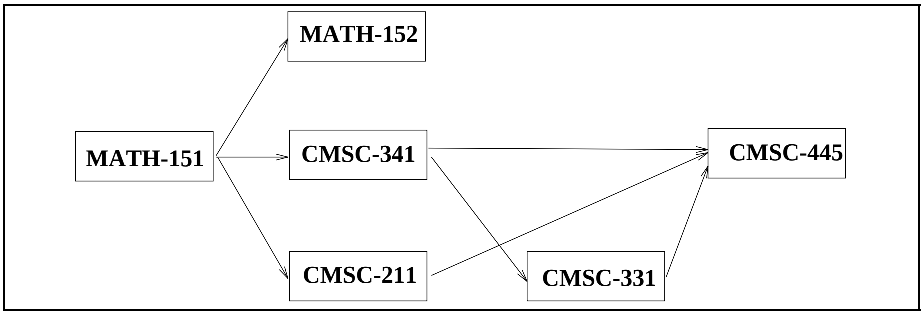


Figure 19.4: A DAG for some Computer Science Courses

EXAMPLE: The usual textbook example of a DAG representing a partial order is the representation of course prerequisites. We define the partial order $\prec$ on the set of courses C to mean that for courses $a, b \in C$, $a \prec b$ if and only if a is a prerequisite for b.

EXAMPLE: In figure 19.4, CMSC–151 $\prec$ CMSC–341 $\prec$ CMSC–331 $\Rightarrow$ CMSC–151 $\prec$ CMSC–331 (the transitive property). Note that MATH–152 $\not\prec$ CMSC–445.

19.6.1 Some More Definitions

◇ **Definition**: Vertex i is a *predecessor* of vertex j if and only if there is a path from i to j.

◇ **Definition**: Vertex i is an *immediate predecessor* of vertex j if and only if (i, j) is an edge in the graph.

- Similar definitions for *successor* and *immediate successor*.

19.7 Topological Ordering

◇ **Definition**: A *topological ordering* of the vertices of a DAG $G = (V, E)$ is a linear ordering such that, for vertices $i, j \in V$, if i is a predecessor of j, then i precedes j in the linear order.

In general, there is more than one topological ordering for a given DAG.

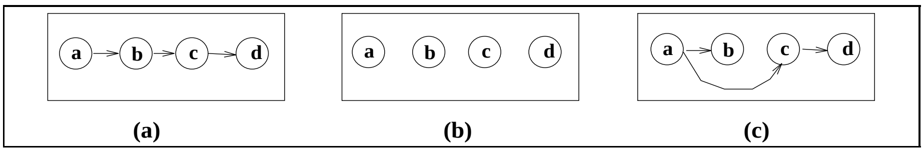


Figure 19.5: Examples for Topological Ordering

EXAMPLE: Figure 19.5 show three graphs. Graph (a) has only one topological ordering, namely `a b c d`. Graph (b) has a large number of topological ordering, namely $4! = 24$. Graph (c) has more than 1 and fewer than 24 topological orderings (two of them are `a b c d` and `a c b d`.

19.7.1 An Algorithm for Topological Ordering

Here is pseudo-code for performing a topological ordering on a DAG `G = (V,E)`. We assume that `Num_Vertex`, the size of `V`, is known beforehand.

```
void TopSort(Graph G)
{
  unsigned int counter = 1;
  Vertex v, w;
  Queue q = new Queue();
  Vertex [] indegree = new Vertex[G.numbVertices());

  for each Vertex v
   { indegree[v] = getInDegree(v); // fill indegree matrix and
     if (indegree[v] == 0)      // enqueue zero indegree vertices.
       q.enqueue(v);

  while (!q.Is_Empty())
  {
    v = q.dequeue();
    Put v on the topological ordering;
    counter++;
    for each Vertex w adjacent to v
      {
        indegree[w] -= 1;
        if (indegree[w] == 0)
          q.enqueue(w);
      }
  if (counter <= G.numbVertices())
    declare an error -- G has a cycle
}
```

• `Vertex` might just be an integer index of an array representation. In this case we would have done `typedef int Vertex;`

• The `counter` variable is used to check for the existence of a cycle in the graph. `counter` is incremented after each `dequeue` operation. If the indegree of a vertex is "too high" due to a cycle, it will not be enqueued, so `counter` will be too low.

▷ **QUESTION**:

Under what circumstances can `counter` be equal to `Num_Vertex`?

▷ **ANSWER**:

One circumstance would be a graph with a single vertex and a directed edge from the vertex to itself.

- Rather than wait till the algorithm is completed, a partial test for cycles can be made immediately after filling the `indegree` array. A sufficient, but not necessary, condition for the presence of a cycle is that no vertex have zero indegree.

  ▷ **QUESTION**:

  What is the asymptotic performance of `TopSort` as a function of the size of the graph?

  ▷ **ANSWER**:

  It depends on the graph representation. The performance will depend on the computation of the in-degree of each vertex and on the adjacent-to computation.

Adjacency Matrix Representation

The first `for` loop is done $|V|$ times. Since `getInDegree` is a $O(|V|)$ operation, the for loop is $O(|V|^2)$. The `while` loop is also done $|V|$ times, each time doing $O(|V|)$ work for a total of $O(|V|^2)$. Thus, with the adjacency matrix representation, topological sort is a $O(|V|^2)$ algorithm.

Adjacency List Representation

The first `for` loop is done $O(\sum_{i=1}^{|V|} D(i)) = O(|E|)$ times. However, if the graph has no edges, it is $O(|V|)$. Thus, the first loop is $O(|V| + |E|)$. The `while` loop takes

$$O(\sum_{i=1}^{|V|} D(i) + |V|) = O(2|E| + |V|) = O(|E| + |V|)$$

operations. Thus, with the adjacency list representation, topological sort is a $O(|V| + |E|)$ algorithm.

EXAMPLE: TopSort on the DAG in Figure 19.6:

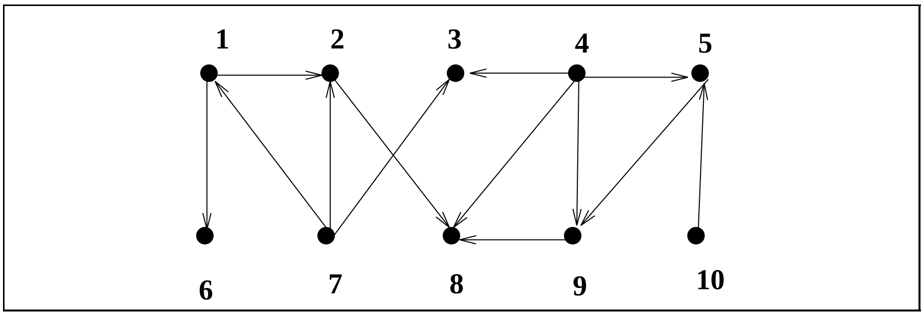


Figure 19.6: Example for TopSort Algorithm

```
indegree array at start:
vertex:    1 2 3 4 5 6 7 8 9 10
indegree   1 2 2 0 2 1 0 3 2  0

step     queue      v     counter
 0        --        -        1    // starting condition
 1      4,7,10      -        1    // vertices with zero indegree
 2      7,10        4        2    // dequeue 4
--------------------------------------------------------------
vertex:    1 2 3 4 5 6 7 8 9 10   // adjust for adjacent to 4
indegree   1 2 1 0 1 1 0 2 1  0
--------------------------------------------------------------
 3      10          7        3    // dequeue 7
--------------------------------------------------------------
vertex:    1 2 3 4 5 6 7 8 9 10   // adjust for adjacent to 7
indegree   0 1 0 0 1 1 0 2 1  0   // 1 and 3 go to zero
--------------------------------------------------------------
 4      1,3        10        4     // dequeue 10, enqueue 1 and 3
--------------------------------------------------------------
vertex:    1 2 3 4 5 6 7 8 9 10   // adjust for adjacent to 10
indegree   0 1 0 0 0 1 0 2 1  0   // 5 goes to zero
--------------------------------------------------------------
 5      3,5        1         5    // dequeue 1, enqueue 5
--------------------------------------------------------------
vertex:    1 2 3 4 5 6 7 8 9 10   // adjust for adjacent to 1
indegree   0 0 0 0 0 0 0 2 1  0   // 6 and 2  go to zero
--------------------------------------------------------------
 6      5,6,2       3         6    // dequeue 3, enqueue 6 and 2
--------------------------------------------------------------
vertex:    1 2 3 4 5 6 7 8 9 10   // adjust for adjacent to 3
indegree   0 0 0 0 0 0 0 2 1  0   // no change
--------------------------------------------------------------
```

```
--------------------------------------------------------------
7      6,2         5         7    // dequeue 5
--------------------------------------------------------------
vertex:     1 2 3 4 5 6 7 8 9 10   // adjust for adjacent to 5
indegree    0 0 0 0 0 0 0 2 0  0   // 9 goes to zero
--------------------------------------------------------------
 8     2,9          6         8    // dequeue 6, enqueue 9
--------------------------------------------------------------
vertex:     1 2 3 4 5 6 7 8 9 10   // adjust for adjacent to 6
indegree    0 0 0 0 0 0 0 2 0  0   // no change
--------------------------------------------------------------
 9     9           2         9    // dequeue 2
--------------------------------------------------------------
vertex:     1 2 3 4 5 6 7 8 9 10   // adjust for adjacent to 2
indegree    0 0 0 0 0 0 0 1 0  0   // no change
--------------------------------------------------------------
 10                9        10    // dequeue 9
--------------------------------------------------------------
vertex:     1 2 3 4 5 6 7 8 9 10   // adjust for adjacent to 9
indegree    0 0 0 0 0 0 0 0 0  0   // 8 goes to zero
--------------------------------------------------------------
 11    8           -        11    // enqueue 8
 12    -           8        12    // dequeue 8
                                  // no adjustment of array
```

19.8 Spanning Trees

◇ **Definition**: A *spanning tree* of a connected graph $G = (V, E)$ is a minimal (lowest number of edges) subgraph G' such that G' is connected, $V(G') = V(G)$, and $E(G') \subset E(G)$.

• Less formally, a *spanning tree* of a connected graph G is a tree consisting solely of edges from G and including all vertices of G.

• If the graph is not connected, then we get a set of spanning trees. This is called a *spanning forest*.

• In general, a graph has more than one spanning tree.

EXAMPLE: Figure 19.7 shows a graph and some of its spanning trees.

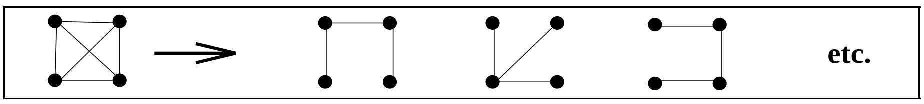


Figure 19.7: A Graph and some of its Spanning Trees

• A spanning tree can be found by doing a breadth-first or depth-first traversal of the graph. The edges traversed form a spanning tree.

• A typical application of spanning trees is finding ways to connect processors with a minimum number of network connections (assigning weight of 1 to each edge). The spanning trees of G represent all feasible choices for a communication network with minimal number of edges.

• A spanning tree is a *free tree* (no vertex distinguished as root).

• Any connected acyclic graph is a free tree.

Theorem 19.4 Every free tree with $n \geq 1$ vertices contains exactly $n-1$ edges.

$\triangleright$ **Proof**: By induction on number of vertices.
Base: True for $n = 1$ (no edges).
IH: Assume true for all free trees with $n < N$ vertices.
Proof: Add a vertex to a free tree of $N-1$ vertices, connecting it to the tree by an edge. By IH, the original tree had $N-2$ edges. It now has $N-1$ edges. $\triangleleft$

Theorem 19.5 If we add an edge (but not a vertex) to a free tree, a cycle is produced.

$\triangleright$ **Proof**: By contradiction: suppose that an edge could be added to a free tree of n vertices without producing a cycle. Then this new tree is still a free tree, and it has n vertices and n edges. This violates Theorem 19.4. $\triangleleft$

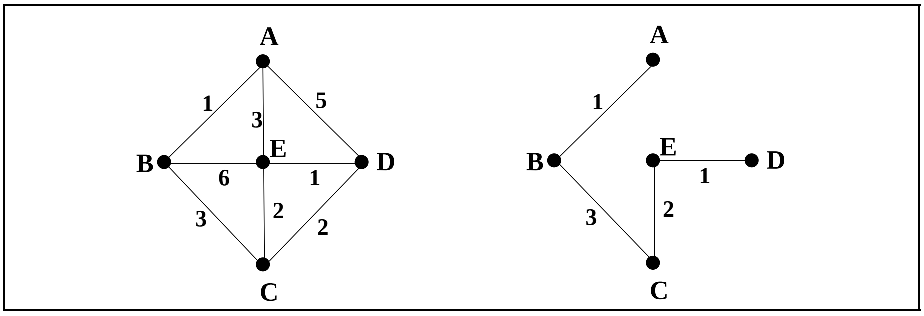
$\diamond$ **Definition**: A *minimum spanning tree* (MST) is a spanning tree of a weighted graph such that the sum of the weights of the edges is minimal.

$\diamond$ **Definition**: Every minimum spanning tree satisfies the *minimum spanning tree property*, namely:
For weighted, undirected graph $G = (V, E)$, let U be some subset of V such that neither U nor $V - U$ are empty. Let (u, v) be a lowest cost edge such that $u \in U$ and $v \in V - U$. Then, there is a minimum spanning tree that includes (u, v) as an edge.

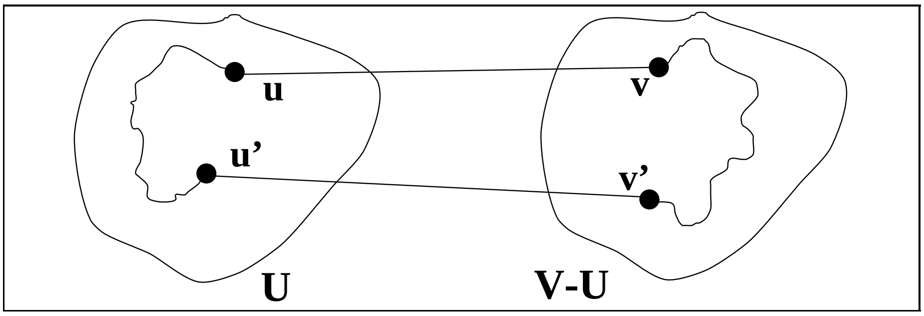
Note: The MST property does not say that *every* MST contains the edge (u, v). It does say that *some* MST contains it.

EXAMPLE: For the following graph, suppose $U = \{B, C\}$. Then, there is a MST that includes edge (A, B).

Theorem 19.6 (Proof of MST property) For weighted, undirected, connected graph $G = (V, E)$, let U be some subset of V such that neither U nor $V - U$ are empty. Let (u, v) be a lowest cost edge such that $u \in U$ and $v \in V - U$. Then, there is a minimum spanning tree that includes (u, v) as an edge.



▷ **Proof**: By contradiction: Suppose there is **NO** MST which includes (u, v). Let T be some MST of G. Add (u, v) to it, thus introducing a cycle (which involves (u, v)). There must also be an edge (u', v') between U and $V - U$, or else there would be no cycle.



So, remove (u', v') to get a spanning tree again. Since the weight of (u, v) is no greater than the weight of (u', v') ((u, v) is a minimum cost edge between U and $V - U$), the spanning tree cost is no larger than T, which was a MST. Therefore, there is a MST which contains (u, v), contradicting the assumption. ◁

• Two famous algorithms for finding MST use the MST property. These algorithms are *Prim's algorithm* and *Kruskal's algorithm*.

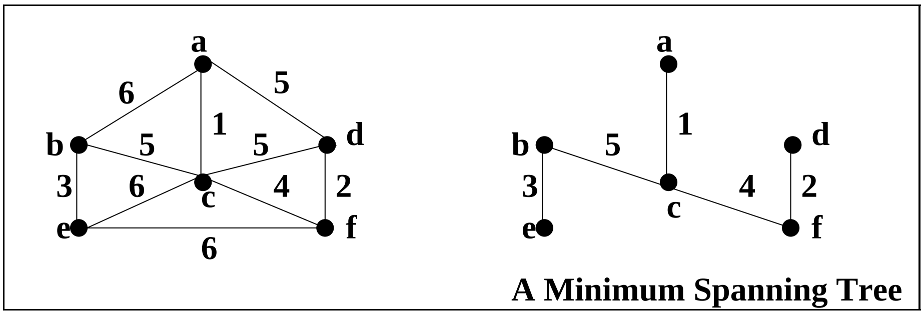
19.8.1 Prim's Algorithm

Start with $U = \phi$. Vertices are then added to U by finding the lowest cost edge from U to V-U at any time.

```
Put any vertex in U;  // U initially empty
while (V-U is not empty)
  {
    Choose a lowest cost edge (u,v) where
        u is in U  and v is in V-U;
    Add v to U (thus removing it from V-U);
  }
```

Prim is $O(|V|^2)$ since every vertex is eventually added to U; and each time a vertex is added to U, we must scan the remaining vertices in $V - U$.

EXAMPLE: In the graph shown below, start with $U = \{a\}$. Sequentially, U becomes $\{a, c\}$, $\{a, c, f\}$, $\{a, c, f, d\}$, $\{a, c, f, d, b\}$, $\{a, c, f, d, b, e\}$; producing the MST shown. The cost of the MST is 15.



19.8.2 Kruskal's Algorithm

Unlike Prim, Kruskal's algorithm does not require scanning over all the vertices to find minimum weight edges. Instead, it chooses a greedy approach, selecting the appropriate minimum weight edge at any time. Here's the algorithm:

Start with the vertices of G, but no edges. This gives us a collection of $|V|$ connected (i.e., single vertex) components. We build up the size of these components one edge at a time. The number of edges needed to form the spanning tree will be $|V| - 1$. We know that $|E|$ is at least this large since the graph is connected.

```
Produce a list of edges, L, in non-decreasing order by weight
while ( number of edges selected < |V|-1 )
 {
  w = L.Remove(); // remove from head of list (lowest cost edge)
  if (w connects two unconnected components)
      select w and add it to the MST
  else // w is internal to a component
      do not select w  // it would have caused a cycle in T
 }
```

• Analysis of Kruskal:
$O(|E| \lg |E|)$ to put edges in order.

In each iteration:

$O(\lg |E|)$ to find least cost edge.

$O(\lg |E|)$ to check if the edge is in one partition.

$O(\lg |E|)$ to merge two partitions.

• Checking if an edge is in one partition amounts to checking for membership of its vertices in a set. This hasn't been covered in the course yet, but the operation can be done in $O(\lg |E|)$ time.

• Since there are $O(|E|)$ iterations, Kruskal is $O(|E| \lg |E|)$.

• Kruskal is a "greedy" algorithm in that it always takes the lowest cost edge next.

• Kruskal works even when some edges have negative weight.

19.9 Shortest-Path Algorithms

◇ **Definition**: The *single-source shortest-path* problem is this:
"Given as input a weighted graph $G = (V, E)$, and a distinguished vertex $s \in V$, find the path with lowest weight from s to every other vertex in G."

• For unweighted graphs, the single-source shortest-path problem is the weighted graph problem in which each edge has a weight of 1.

EXAMPLE: In the graph of Figure 19.8, the shortest weighted path from vertex 1 to vertex 6 has a cost of 6. The path is 1, 4, 7, 6.
The shortest unweighted path has a cost of 2.

• Actually, there are two shortest unweighted paths from vertex 1 to vertex 6. They are 1, 3, 6 and 1, 4, 6. The "shortest" weighted or unweighted path is not necessarily unique.

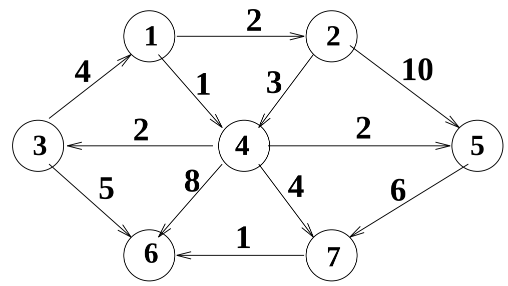


Figure 19.8: Graph For Shortest Path Examples

19.9.1 Unweighted Single-Source Shortest-Paths

• The underlying idea of the algorithm is to do a breadth-first search from the source vertex, tabulating the distances to the other vertices as we go.

Here is the algorithm. `T` is a table of vertices, distances, and path such that `T[i].Dist` is the distance of vertex `i` from the source vertex and `T[i].Path` is the vertex from which vertex `i` was reached on the shortest path from the source. Initially, `T[i].Dist` is `infinity` for all `i`.

```
void
unweightedShortestPath( Table T )
{
  Vertex V, W;
  Queue Q = new Queue();

  Q.Enqueue( S );  // Enqueue the start vertex S, determined elsewhere.

  while( ! Q.isEmpty( ) )
    {
      V = Q.dequeue( );
      for each W Adjacent To V
        if( T[ W ].Dist == Infinity ) // if "unvisited"
         {
            T[ W ].Dist = T[ V ].Dist + 1;
            T[ W ].Path = V;
            Q.enqueue( W );
         }
    }
}
```

• The text has `T[ W ].Dist = T[ V ].Dist;` which is incorrect.

▷ **QUESTION**:

What is the asymptotic performance of `Unweighted Shortest Path`?

▷ **ANSWER**:
The algorithm is clearly just a breadth-first traversal of the graph. We showed, in Section 19.4.1, that breadth-first and depth-first traversal can be done in $O(|V| + |E|)$.

EXAMPLE: Run the algorithm on the graph of Figure 19.8, with edge weights taken as unity, for the single source vertex `3`. The table (call it `Tab3`) is:

Vertex	Dist	Path
1	1	3
2	2	1
3	0	0
4	2	1
5	3	2
6	1	3
7	3	4

Here is the function for printing the path from a vertex `V` to the starting vertex.

```
void
printPath( Vertex V, Table T )
{
  if( T[ V ].Path != Not_A_Vertex )
    {
      printPath( T[ V ].Path, T );
      print " to ";
    }
  print V;
}
```

EXAMPLE: To print the path to vertex 5, using table `Tab3`, call the function as `Print_Path(5, Tab3)`, resulting in the output `3 to 1 to 2 to 5`.

19.9.2 Weighted Single-Source Shortest-Paths

• The Dijkstra algorithm is an example of a greedy algorithm.

• Greedy algorithms are famous for finding locally valid solutions. They do not always work in finding globally valid solutions.

• The Dijkstra algorithm solves the weighted single-source shortest-path problem as long as the weights are all positive. The locally valid solution is also the globally valid solution (as will be proven later).

Here is pseudo-code for the Dijkstra algorithm with positive weight edges. `G = (V,E)` is the graph. Let `D(v)` be the distance from the source to vertex `v`. Let `C(u,v)` be the weight of edge `(u,v)`. Let `S` be a set of vertices, initially empty. Let `P(v)` be the vertex from which `v` is reached on the shortest path from source.

```
Add the source vertex to S;
// initialize D.  This is  O(|V|)
for (each vertex v in V-S)
   D(v) = C(source,v);
// determine shortest paths
while (V-S is not empty)  // do this O(|V|) times
  {
    let w be the vertex in V-S which has minimum D
    add w to S   // thereby removing it from V-S
    for (each vertex v in V-S,  v adjacent to w)
       if (D(w) + C(w,v) < D(v))
         {
           D(v) = D(w) + C(w,v);
           P(v) = w;
         }
   }
```

▷ **QUESTION**:

What is the asymptotic performance of Dijkstra?

▷ **ANSWER**:

It depends on the data structure used to represent the graph. Initializing D is $O(|V|)$, no matter what data structure is used. The while loop performance depends on the data structure.

adjacency matrix The 'adjacent to' operation is $O(|V|)$. Since it is done $O(|V|)$ times, the performance of the while loop is $O(|V|^2)$

adjacency list The 'adjacent to' operation for vertex v is $O(\mathit{getDegree}(v))$. Since it is done $O(|V|)$ times, the while loop is $O(|E|)$.

Updating the vertices in $V - S$ can be done in $O(\lg |V - S|) = O(\lg |V|)$ using a priority queue (e.g., a binary heap).
Therefore, Dijkstra is $O(|V|^2)$ with adjacency matrices and $O(|E| \lg |V|)$ with adjacency lists.

Theorem 19.7 Dijkstra's algorithm works for $G = (V, E)$, a weighted graph with no negative weight edges.

▷ **Proof**: By contradiction. (This proof is due to Aho, Hopcroft and Ullman). In each step of the algorithm,we find a shortest path to a vertex w in $V - S$

which stays entirely within S (except for the edge which goes from S to $V-S$). There cannot be a shorter path from source to w. If there were, it would have to leave S at some point. Figure 19.9 shows a hypothetical shortest path which goes through vertex $x \in V - S$, wanders in $V - S$ and S, and finally gets to vertex $w \in V - S$. If this path is shorter than the path found by the algorithm, then the portion of the path to x is shorter than the path to w. But then, $D(x)$ is shorter than $D(w)$ and x would have been chosen before w. $\triangleleft$

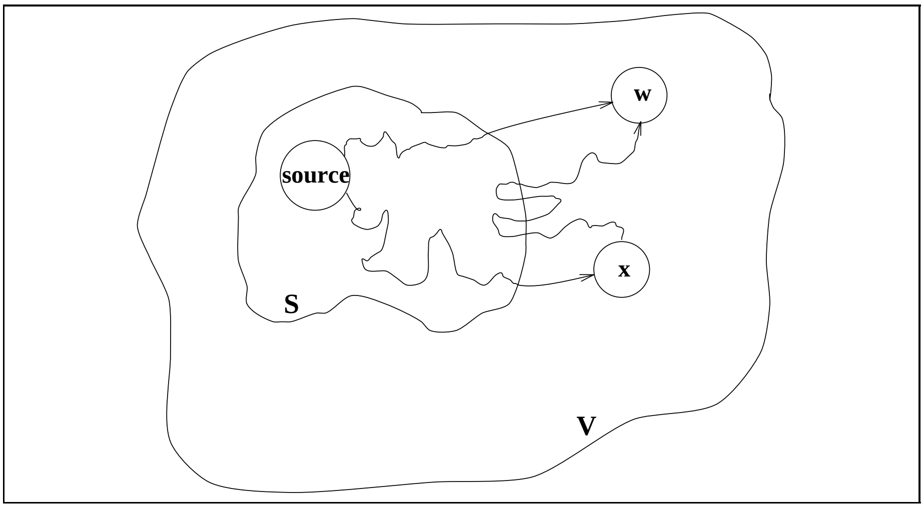


Figure 19.9: Dijkstra: Hypothetical Shorter Path to Vertex `w`

• Notice that Theorem 19.7 is not true if negative edge weights are allowed.

19.10 All-Pairs Shortest-Path Problem

$\diamond$ **Definition**: Given a weighted directed graph $G = (V, E)$, the *all-pairs shortest-path problem* (APSP) is to produce a table (of size $|V| \times |V|$) in which entry (i, j) gives the shortest path length between vertex i and vertex j.

• One way to solve this problem is to use Dijkstra's algorithm $|V|$ times, once for each vertex as source. On sparse graphs (using adjacency list), Dijkstra runs in $O(|E| \lg |V|)$, so APSP can be done in $O(|V||E| \lg |V|)$. For dense graphs (using adjacency matrix), Dijkstra runs in $O(|V|^2)$, so APSP would run in $O(|V|^3)$.

19.10.1 Floyd's Algorithm for APSP

This section is due to Aho, Hopcroft, and Ullman.

• Floyd's algorithm for solving APSP runs in $O(|V|^3)$ time, but involves only simple matrix operations. Therefore, although it has the same asymptotic performance for dense graphs as Dijkstra done $|V|$ times, the constant of proportionality may be much lower for Floyd than for Dijkstra.

• Dijkstra is better than Floyd for sparse enough graphs.

• Floyd's algorithm requires a $|V| \times |V|$ matrix of weights, in addition to the usual cost matrix kept as part of the graph representation. Let $W(i,j)$ be this weight matrix and $C(i,j)$ be the cost matrix. Initially, $W(i,j) = C(i,j)$.

• Make $|V|$ iterations over the W matrix. After the k-th iteration, the W matrix will contain the smallest path from vertex i to vertex j which does not go through any vertex numbered higher than k.

• The operation performed in the k-th iteration is

$$W[i,j] = \min(W[i,j], W[i,k] + W[k,j])$$

In other words, if there is a shorter path going through vertex k, then use it. Figure 19.10 shows the situation.

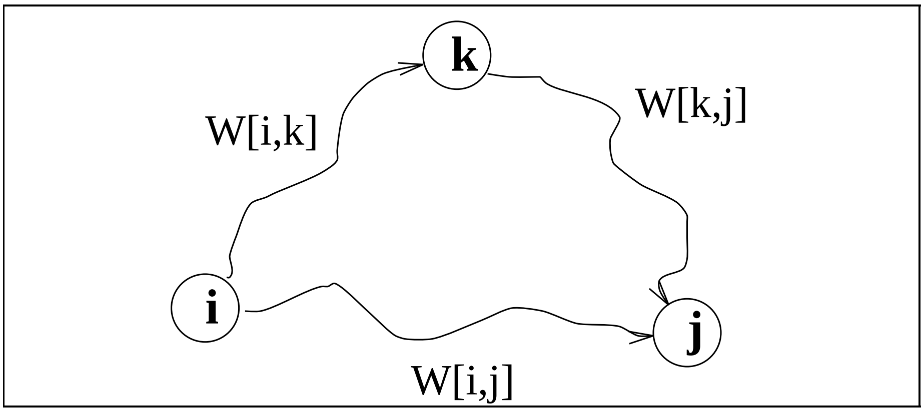


Figure 19.10: Floyd: Paths Involving Vertices i, j, and k

Here is pseudo-code for Floyd's algorith:

```
// initialize the W matrix
for (each vertex i)
  for (each vertex j)
     W[i,j] = C[i,j];
for (each vertex k)
  for (each vertex i)
    for (each vertex j)
      if (W[i,k] + W[k,j] < W[i,j]
        W[i,j] = W[i,k] + W[k,j];
```

• Because of the triple loop over all vertices, Floyd is obviously $O(|V|^3)$.