MA859: SELECTED TOPICS IN GRAPH THEORY

LECTURE - 19

TREE ALGORITHMS

A graph may contain many different trees. Here we study several algorithms used to construct trees (spanning trees in particular) with certain optimal properties. Trees have applications in many areas, such as facilities design and electrical networks and within algorithms for other problems, such as the travelling salesman problem.

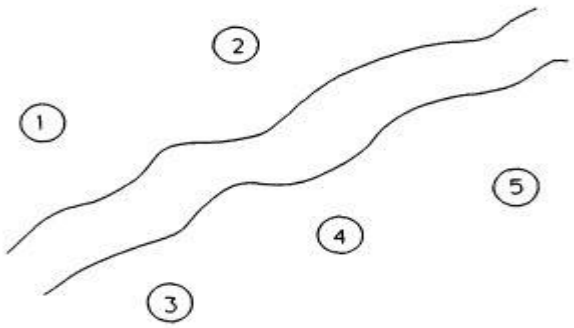
Rumor Monger

Consider a small village in which some of the villagers have a daily chat with one another. Is it possible for a rumor to pass throughout the entire village? To answer this question, represent each villager by a vertex. Join two vertices by an edge if the corresponding two villagers have a daily chat with one another. If the resulting graph is connected, then it is possible for a rumor to pass through the entire village. Since a tree is necessarily connected, a graph that possesses no spanning tree cannot be connected. Therefore, we need only check if the graph is connected.

Let G = (X,E) be an undirected graph in which each edge (x,y) is assigned a weight a(x ,y). We define the weight of the tree as the sum of the weights of the edges in the tree.

Highway Construction

The Department of Highways wishes to build enough new roads so that the five towns in a certain county will all be connected to one another cither directly or via another town. The cost of constructing a highway between each pair of towns is known (see Fig. below).



From/to	1	2	3	4	5
1	—	5	50	80	90
2		—	70	60	50
3			—	8	20
4				—	10
5					—

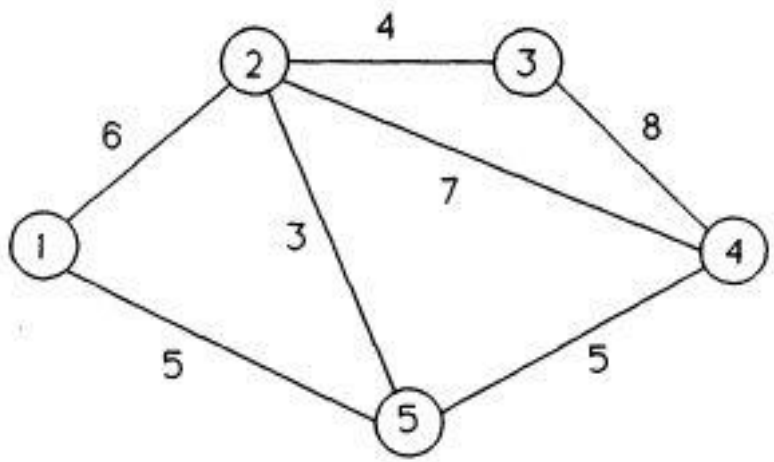
We may represent this problem as a graph by letting each town correspond to a vertex, and each possible highway to be constructed correspond to an edge joining two vertices (towns). Associate a weight to each edge that is equal to the cost of constructing the corresponding highway.

Deciding which highways to build can be viewed as the problem of constructing a minimum-cost spanning tree for the corresponding graph. This follows since the edges of any spanning tree will connect each vertex (town) with every other vertex (town). Moreover, the minimum-weight spanning tree will represent the set of new highways of minimum total cost.

Note that if the highways are allowed to have junctions at places other than the five towns, the resulting problem is more complicated than a minimum spanning tree problem. This problem is called a Steiner tree problem and will be discussed later.

Maximum-Capacity Route

Suppose that G = (X, E) is an undirected network in which each edge (x, y) has a capacity c(x, y) that represents the maximum amount of flow that can pass through the edge (x, y). The maximum-capacity route problem (Fig. below) is to find a route between two specified vertices, say 1 and 5, such that the smallest edge capacity on the route is the maximum among all possible routes. It is easy to demonstrate that the maximum-weight spanning tree provides a solution to this problem and, in fact, solves the problem for every pair of vertices in the graph.



SPANNING TREE ALGORITHMS

The spanning tree algorithm, given in the following discussion, is one of the most elegant algorithms that we shall encounter. The algorithm examines the edges in any arbitrary sequence and decides whether each edge will be included in the spanning tree. Thus, each edge will be labeled as either assigned to the spanning tree or excluded from the spanning tree.

When an edge is examined, the algorithm simply checks if the edge under consideration forms a cycle with the other edges already assigned to the tree. If so, then the edge under consideration is excluded from the tree; otherwise, it is assigned to the tree.

Clearly, if one were performing this algorithm by hand, it would be easy to determine visually when a cycle is formed. However, this may not be as straightforward for a computer implementation. There is a very simple way to accomplish this task. As the edges are assigned to the tree they form one or more connected components. The vertices belonging to a single connected component are collected together into what the algorithm terms a “bucket”. An edge forms a cycle with the edges already assigned to the tree if both its endpoints are in the same connected component (bucket). A unique number can be assigned to each bucket and hence to all the vertices in the bucket. To check whether both endpoints are in the same bucket, we need only compare the bucket number assigned to those vertices.

The algorithm terminates with a spanning tree when the number of edges assigned to the tree equals the number of vertices less one, or, equivalently, when all vertices are in one bucket, since all spanning trees have the same number of edges. If the graph contains no spanning tree, which is equivalent to not being connected, then the algorithm terminates after examining all edges without assigning enough edges to the tree, and the vertices will be contained in more than one bucket.

Spanning Tree Algorithm

Initially all edges are unexamined and all buckets are empty.

Step 1. Select any edge that is not a loop. Assign this edge to the tree and place both its endpoints in an empty bucket.

Step 2. Select any unexamined edge that is not a loop. (If no such edge exists, then stop the algorithm as no spanning tree exists.)

One of four different situations must occur:

(a) Both endpoints of this edge are in the same bucket.

(b) One endpoint of this edge is in a bucket, and the other endpoint is not in any bucket.

(c) Neither endpoint is in any bucket.

(d) Each endpoint is in a different bucket.

If condition (a) occurs, then exclude the edge from the tree and return to step 2. If (b) occurs, assign the edge to the tree and place the unbucketed endpoint in the same bucket as the other endpoint. If (c) occurs, then assign the edge to the tree and place both endpoints in an empty bucket. If (d) occurs, then assign the edge to the tree and combine the contents of both buckets in one bucket, leaving the other bucket empty. Go to step 3.

Step 3. If all the vertices of the graph are in one bucket or if the number of assigned edges equals the number of vertices less one, stop the algorithm since the assigned edges fonn a spanning tree. Otherwise, return to step 2.

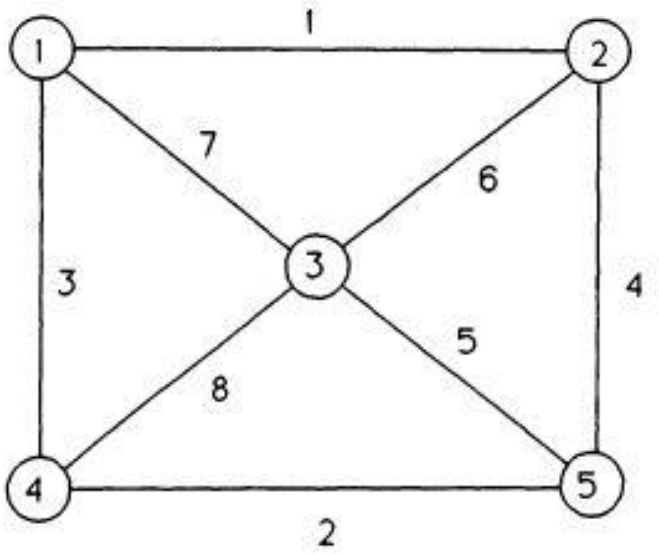
Note that each time a step of the algorithm is performed, one edge is examined. If there is only a finite number of edges in the graph, the algorithm must stop after a finite number of steps. Thus, the time complexity of this algorithm is $O(n)$, where n is the number of edges in the graph.

If the algorithm does not terminate with a spanning tree, then no spanning tree exists in the graph for the following reason. The algorithm will terminate with two nonempty buckets of vertices that have no edge joining a member of one set to a member of the other set. Otherwise, such an edge would have been assigned to the tree [condition (d) would have occurred] and the two buckets would have been combined by the algorithm. Hence, the algorithm does what it is supposed to do, that is, construct a spanning tree.

This algorithm has the property that each edge is examined at most once. Algorithms, like the spanning tree algorithm, which examine each entity at most once and decide its fate once and for all during that examination are called greedy algorithms. The advantage of performing a greedy algorithm is that you do not have to spend time reexamining entities. Clearly, such algorithms are extremely computationally efficient.

EXAMPLE 1:

Let us construct a spanning tree for the graph in Fig. below.



Examine the edges in the following arbitrary order: (1, 2), (4, 5), (1, 4), (2, 5), (5, 3), (2, 3), (1, 3), and (3, 4).

The results of each step of the algorithm are shown below.

Edge	Assigned?	Bucket No. 1	Bucket No. 2
(1, 2)	yes	1, 2	empty
(4, 5)	yes	1, 2	4, 5
(1, 4)	yes	1, 2, 4, 5	empty
(2, 5)	no	1, 2, 4, 5	empty
(5, 3)	yes	1, 2, 4, 5, 3	empty

Since all vertices are in one bucket (and four edges have been assigned from the five-vertex graph), a spanning tree has been found.

Obviously, the spanning tree constructed by the algorithm depends on the order in which the edges are examined. If the edges in the example had been examined in the reverse order, the algorithm would have generated the spanning tree consisting of edges (3, 4), (1, 3), (3, 2), and (5, 3).

Now consider the problem of finding a spanning tree with the smallest possible weight or the largest possible weight, respectively called a *minimum spanning tree* and a *maximum spanning tree*. Obviously, if a graph possesses a spanning tree, it must have a minimum spanning tree and also a maximum spanning tree. These spanning trees can be readily constructed by performing the spanning tree algorithm with an appropriate ordering of the edges.

Minimum Spanning Tree Algorithm

Perform the spanning tree algorithm examining the edges in order of non-decreasing weight (smallest first, largest last). If two or more edges have the same weight, order them arbitrarily.

Maximum Spanning Tree Algorithm

Perform the spanning tree algorithm examining the edges in order of non-increasing weight (largest first, smallest last). If two or more edges have the same weight, order them arbitrarily.

Proof of the Minimum Spanning Tree Algorithm. We shall prove by contradiction that the minimum spanning tree algorithm constructs a minimum spanning tree. Suppose that the algorithm constructs a tree T and some other tree S is in fact a minimum spanning tree. Since S and T are not identical trees, they differ by at least one edge. Denote by e1 = (x, y) the first examined edge that is in T but not in S. Since S is a spanning tree, there exists in S, a unique path from vertex x to vertex y. Call this path C(x, y). If edge e1 = (x, y) is added to tree S, then a cycle is formed by e1 and C(x, y). Since tree T contains no cycles, this cycle must contain at least one edge e2 not contained in tree T.

Remove edge e2 from S and add edge ej to S. Call the resulting set of edges S'. Clearly, S' is also a spanning tree. Since S is by definition a minimum spanning tree, the weight of S' must be greater than or equal to the weight of S, and hence $a(e1) \geq a(e2)$.

Suppose that edge e2 was examined before edge el in the minimum spanning tree algorithm that generated tree T. Since e2 was not included in tree T, edge e2 must form a cycle with the edges of tree T that were examined before edge e2' But since el is defined as

the first edge in T that is not in S, all other edges in this cycle must be in tree S. This is a contradiction since edge e2 forms a cycle with these edges. Therefore, we must conclude that edge e, was examined before edge e2, and $a(e2) \geq a(e1)$. Therefore, trees S and S' have the same total cost, and tree S' has one more edge in common with tree T than does spanning tree S.

The proof can now be repeated using S' as the minimum-cost spanning tree instead of S. This generates another minimum spanning tree S" that has one more edge in common with tree T than did tree S'. Ultimately, a minimum spanning tree will be generated that is identical to tree T. Thus, tree T is a minimum spanning tree. □

The proof for the maximum spanning tree algorithm is identical to the preceding proof except that minimum should everywhere be replaced by maximum.

EXAMPLE 2

We will construct a minimum-cost spanning tree for the highway problem given in Fig. (in slide 2). The edges in nondecreasing order of cost are (1, 2), (3, 4), (4, 5), (3, 5), (1, 3), (2, 5), (2, 4), (2, 3), (1, 4) and (1, 5).

The results of the minimum spanning tree algorithm are:

Edge	Assigned?	Bucket No. 1	Bucket No. 2
(1, 2)	yes	1, 2	empty
(3, 4)	yes	1, 2	3, 4
(4, 5)	yes	1, 2	3, 4, 5
(3, 5)	no	1, 2	3, 4, 5
(1, 3)	yes	1, 2, 3, 4, 5	empty

Stop, since all vertices are in the same bucket and four edges have been assigned to the tree. A minimum-cost spanning tree consists of the edges (1, 2), (3,4), (4, 5), and (1, 3).

The tree consisting of the edges (1, 2), (3, 4), (4, 5), and (2, 5) could also be a minimum-cost spanning tree. What is the optimum cost?

The algorithm that we have presented for minimum spanning trees is due to Kruskal (1956). For a graph with *m* vertices and *n* edges, Kruskal's method may take up to *n* iterations. Each iteration requires the determination of the smallest edge available; this can be done in *O*(*log n*) using appropriate data structures. Hence the complexity is of order *O*(*n log n*). Notice that if the graph is dense, then the number of edges approaches *m(m - 1)/2*; hence the time complexity will be of order *O*(*m^2 log n*). On the other hand, if the graph is sparse, the number of edges is of order *m* and the time complexity will be *O*(*m log n*).

An alternative algorithm was developed by Prim (1957). It can be described as follows:
Step 1. Begin with any vertex. Select the edge of least weight that connects this vertex with another.
Step 2. At any intermediate iteration we have a subtree (a tree that is not spanning). Select the edge of least weight that connects some vertex in the subtree to a vertex not in the subtree.
Step 3. If the subtree spans all vertices, stop. Otherwise, return to step 2.

We shall illustrate Prim's algorithm using the highway construction example. We will arbitrarily select vertex 1 to begin. The edge of least weight incident to vertex 1 is (1, 2) having a weight of 5. Thus, the first subtree consists of the edge (1, 2). The remaining steps of the algorithm are summarized below.

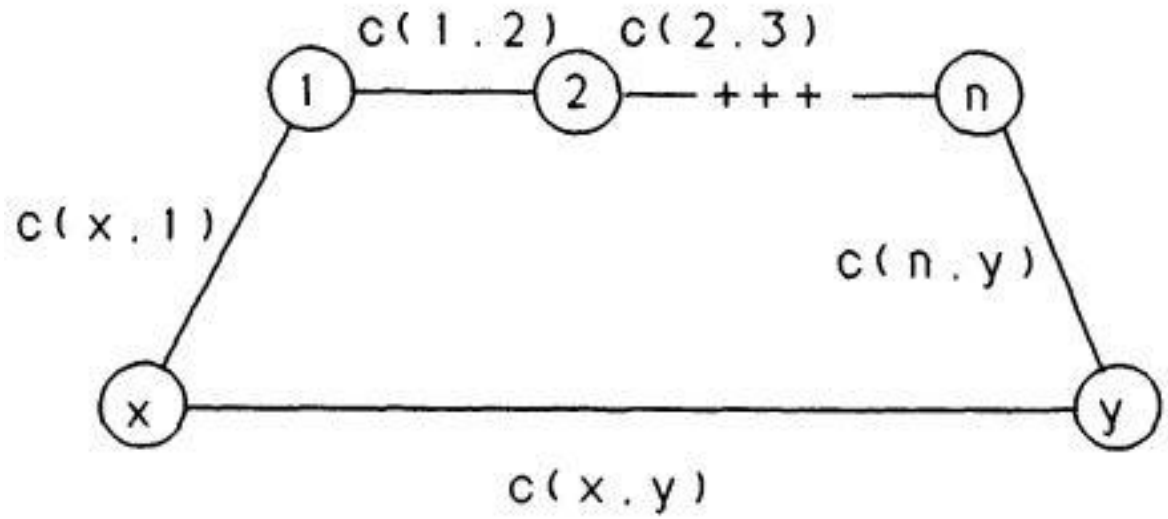
Iteration	Vertices in Subtree	Edge Selected	Weight
1	1	(1, 2)	5
2	1, 2	(1, 3)	50
3	1, 2, 3	(3, 4)	8
4	1, 2, 3, 4	(4, 5)	10

Prim's algorithm takes *m - 1* iterations to complete because one edge is added at each iteration. Finding the next edge to add can be done in *O*(*m*) time. Thus, the overall complexity of this algorithm is *O*(*m^2*).

EXAMPLE 4

To solve the maximum-capacity route problem, we find a maximum spanning tree. This can be done easily by selecting edges largest first. Why does the maximum spanning tree solve this problem? Any edge (*x, y*) not in the tree must satisfy

$$c(x, y) \leq \min\{c(x, 1), c(1, 2), \ldots, c(n, y)\}$$ (see fig. below)



Otherwise we could replace some edge by (x, y) and have a larger minimum capacity from vertex x to vertex y. Thus, in the graph in Fig. (in slide no. 3), the maximum spanning tree is given by the edges (3, 4), (2, 4), (1, 2) and (4, 5). The maximum capacity on the route from vertex 1 to vertex 5 is 5; from vertex 2 to vertex 3, it is 7 and so on.

VARIATIONS OF THE MINIMUM SPANNING TREE PROBLEM

Many variations of the minimum spanning tree problem have been proposed and investigated. These typically involve adding restrictions on the structure of feasible solutions. In this section, we briefly review some of these variations.

The Capacitated Minimum Spanning Tree (Chandy and Lo, 1973)

The capacitated minimum spanning tree problem involves finding a spanning tree on a flow network that has applications in the design of teleprocessing networks. We are given an undirected graph $G = (X, E)$ with cost $d(x, y)$ and capacity $c(x, y)$ associated with edge (x, y). One of the vertices is a sink; the rest are sources. Each source i has a supply $a(i)$, and the sink has a demand equal to the sum of all supplies. The problem is to find a spanning tree in which all flow goes from the sources to the sink only along the edges of the tree at minimum cost and subject to the capacity constraints and, of course, conservation of flow. Fig. 1 below shows an example for which the minimum spanning tree results in an infeasible solution. Figure 2 below gives the optimal solution. No efficient algorithm is known for the general problem; thus heuristics must be used.

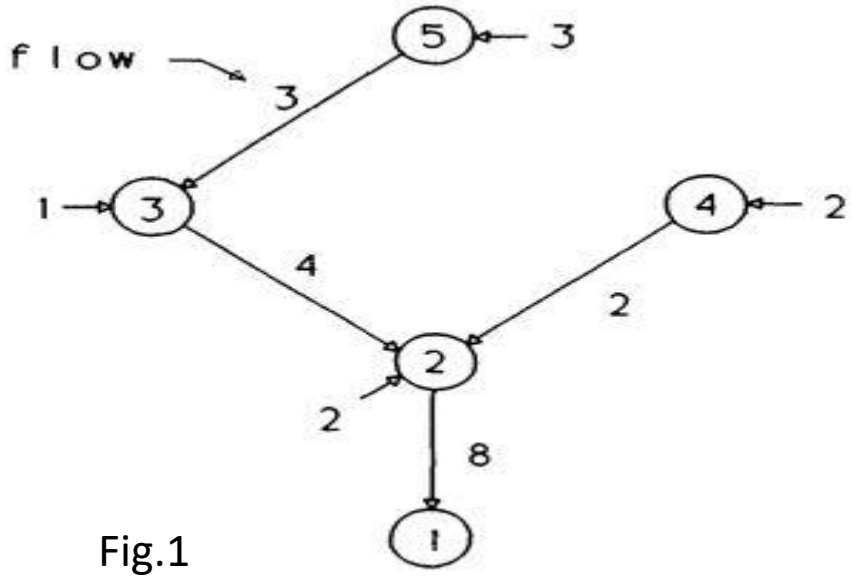


Fig.1

Cost Matrix

	1	2	3	4	5	Supply
1	–	1	3	3	4	Sink
2	1	–	1	1	3	2
3	3	1	–	2	1	1
4	3	1	2	–	3	2
5	4	3	1	3	–	3

All edge capacities = 5

Degree-Constrained Spanning Trees (Gabow, 1978; Glover and Klingman, 1974)

Many applications of spanning trees occur in computer and communication networks. Many of these problems have constraints restricting the number of edges that are allowed to be incident to a vertex. For example, one vertex might represent a central computing site. The other vertices might represent terminals which must be linked to the central site by cable paths. If we restrict the number of edges incident to the central site vertex to *b*, this guarantees that the computer's load is spread over *b* ports. Finding the minimum spanning tree guarantees that as little cable as possible is used. If only one vertex is constrained, as in this example, poJynomial algorithms have been developed.

Optimum Communication Spanning Trees (Hu, 1974)

We are given a network with *n* vertices and a set of requirements *r*(*x, y*) between the vertices. These might represent telephone calls, for example. We wish to design a spanning tree so that the cost of communication is minimum among all spanning trees. We compute the cost of communication as the sum over all pairs of vertices of *r*(*x, y*) times the distance *d*(*x, y*) between vertices *x* and *y*. Recall that in a tree there is a unique path between any two vertices; thus these distances are unique. Figure 2 shows an example problem. A feasible solution is shown in Fig. 3.

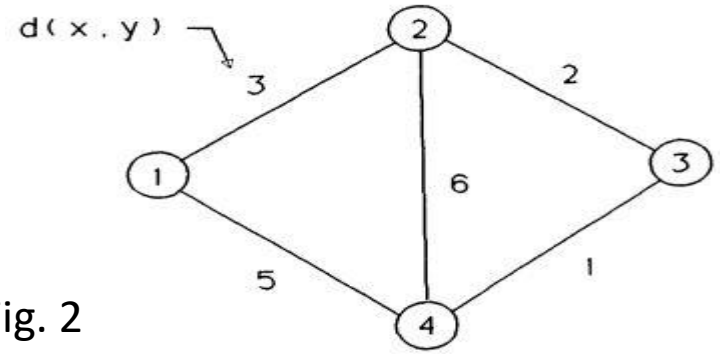


Fig. 2

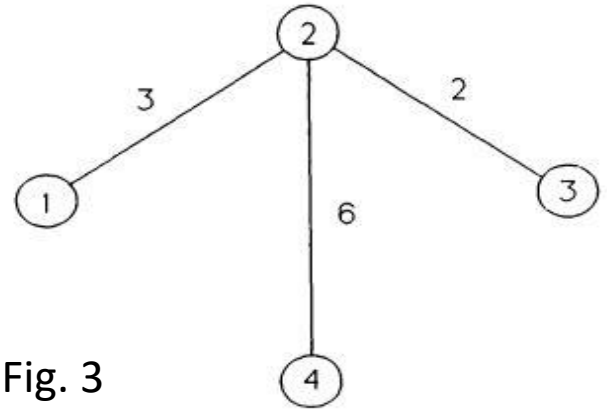


Fig. 3

$r(x, y)$

	1	2	3	4
1	—	2	5	3
2		—	3	3
3			—	6
4				—

The cost of this solution is given by

$$r(1,2)d(1,2) + r(1,3)d(1,3) + r(1,4)d(1,4) + r(2,3)d(2,3) + r(2,4)d(2,4) + r(3,4)d(3,4)$$
$$= 2(3) + 5(5) + 3(9) + 3(2) + 3(6) + 6(8)$$
$$= 130$$

Except for the special cases in which all requirements are 1 or all distances are 1, the problem is not efficiently solved.

Steiner Trees

The Steiner problem on a graph is described as follows. Given a graph $G = (X, E)$ and a subset of vertices X', find a minimal-weight tree that includes all vertices of X' (and possibly others). The minimum spanning tree may not provide the optimal solution. For example, consider the graph in Fig. 4 and the subset of vertices {*3, 4, 6*}.

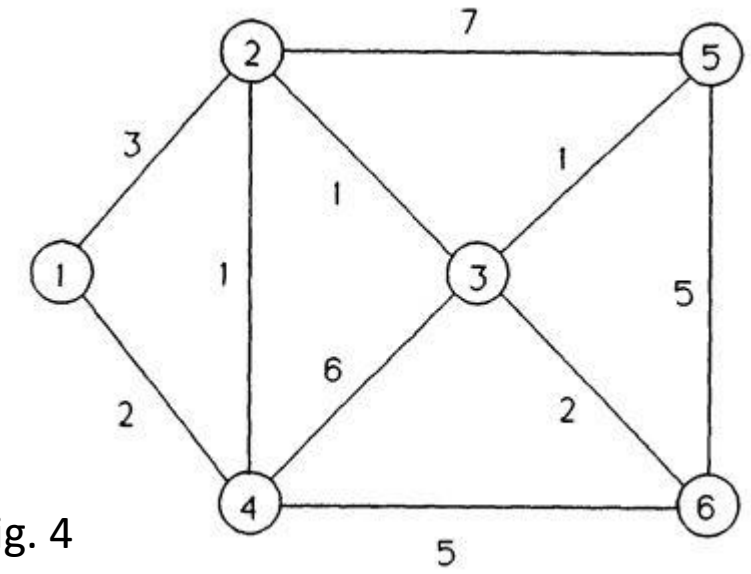
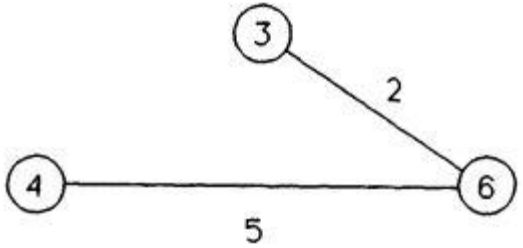


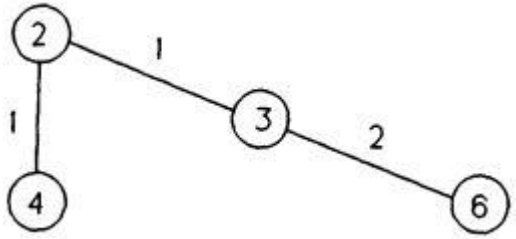
Fig. 4

If we find the minimum spanning tree only on the subgraph generated by vertices *3, 4* and *6*, we get



having a total weight of 7.

However, by including vertex 2 in the set, we find



which has a weight of 4.

The Steiner graph problem is NP-complete, so no polynomial algorithm is likely to exist. Although various exact algorithms have been developed, large problems require the use of heuristics. A comprehensive survey of this problem is given by Winter (1987). One heuristic with a guaranteed performance measure is given by Kou et al. (1978). The algorithm proceeds as follows for a graph *G*.

Step 1. Construct a complete graph *G'* from *G* and *X'* in the following manner. The set of vertices in *G'* is the set *X'*, and for every edge (*x, y*) in *G'*, the distance of edge (*x, y*) is equal to the distance of the shortest path between vertices *x* and *y* in G. (We shall study shortest-path algorithms in the next discussion. For now, we assume that these distances can be found.)

Step 2. Find a minimum spanning tree *T*(*G'*) in *G'*.

Step 3. Replace each edge (*x, y*) in *T*(*G'*) by a shortest path between vertices *x* and *y* in *G*. The resulting graph *G''* is a subgraph of *G*.

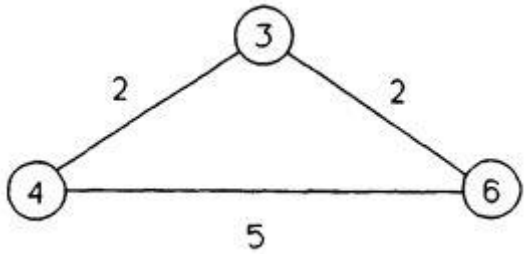
Step 4. Find a minimum spanning tree *T*(*G''*) in *G''*.

Step 5. Delete edges in *T*(*G''*), if necessary, so that all vertices having degree 1 in *T*(*G''*) are members of *X'*. This represents a solution to the Steiner tree problem.

EXAMPLE 5

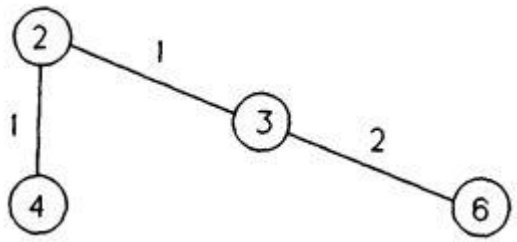
We will apply this algorithm to the graph in Fig.4.

Step 1. The complete graph *G′* is shown below. Note that the shortest distance between vertices 3 and 4 is two.



Step 2. The minimum spanning tree in *G′* consists of edges (3,4) and (3,6).

Step 3. We replace edge (3, 4) by its shortest path in *G*. The resulting graph, *G″*, is shown below.



Step 4. The minimum spanning tree *T*(*G″*) in *G″* consists of the edges (2, 3), (2, 4) and (3, 6).

Step 5. Since no vertices having degree 1 are not members of *X′* , we do not delete any edges.

The solution to the Steiner tree problem determined by this algorithm is (2, 3), (2, 4), and (3, 6).

It can be shown that the time complexity of this algorithm is $O(pm^2)$, where p is the cardinality of the set X' and m is the number of vertices in G. Moreover, it can be shown that the solution will be no more than $2(1 - 1/k)$ times the cost of the optimal solution, where k is the number of vertices of degree 1 in the optimal Steiner tree. Since k generally will not be known, in the worst case $k = m - 1$. Thus, the solution will never be more than $2|1 – 1/(m - 1)|$ times the cost of the optimal tree. The proof is simple but relies on a concept that we will not cover, so we will not prove this result here. This is an example of a heuristic with guaranteed worst-case performance.

// END OF LECTURE //