7.2. Spanning Trees

7.2.1. Spanning Trees. A tree T is a *spanning tree* of a graph G if T is a subgraph of G that contains all the vertices of G. For instance the graph of figure 7.5 has a spanning tree represented by the thicker edges.

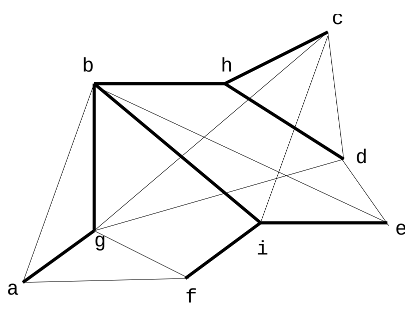

FIGURE 7.5. Spanning tree.

Every connected graph has a spanning tree which can be obtained by removing edges until the resulting graph becomes acyclic. In practice, however, removing edges is not efficient because finding cycles is time consuming.

Next, we give two algorithms to find the spanning tree T of a loop-free connected undirected graph $G = (V, E)$. We assume that the vertices of G are given in a certain order $v_1, v_2, \dots, v_n$. The resulting spanning tree will be $T = (V', E')$.

7.2.2. Breadth-First Search Algorithm. The idea is to start with vertex v_1 as root, add the vertices that are adjacent to v_1, then the ones that are adjacent to the latter and have not been visited yet, and so on. This algorithm uses a queue (initially empty) to store vertices of the graph. In consists of the following:

1. Add v_1 to T, insert it in the queue and mark it as “visited”.
2. If the queue is empty, then we are done. Otherwise let v be the vertex in the front of the queue.
3. For each vertex v' of G that has not been visited yet and is adjacent to v (there might be none) taken in order of increasing subscripts, add vertex v' and edge (v, v') to T, insert v' in the queue and mark it as “visited”.
4. Delete v from the queue.

5. Go to step 2.

A pseudocode version of the algorithm is as follows:

```
procedure bfs(V,E)
  S := (v1) // ordered list of vertices of a fix level
  V' := {v1} // v1 is the root of the spanning tree
  E' := {} // no edges in the spanning tree yet
  while true do
    begin
      for each x in S, in order, do
        for each y in V - V' do
          if (x,y) is an edge then
            add edge (x,y) to E' and vertex y to V'
        if no edges were added then
          return(T)
      S := children of S
    end
end bfs
```

Figure 7.6 shows the spanning tree obtained using the breadth-first search algorithm on the graph with its vertices ordered lexicographically: $a, b, c, d, e, f, g, h, i$.

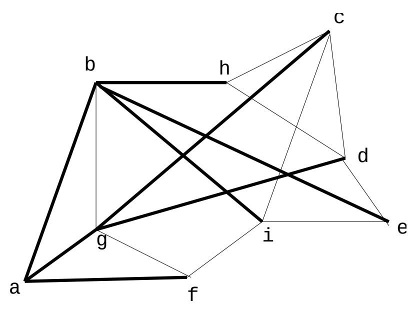

FIGURE 7.6. Breadth-First Search.

7.2.3. Depth-First Search Algorithm. The idea of this algorithm is to make a path as long as possible, and then go back (backtrack) to add branches also as long as possible.

This algorithm uses a stack (initially empty) to store vertices of the graph. In consists of the following:

1. Add v_1 to T, insert it in the stack and mark it as "visited".

2. If the stack is empty, then we are done. Otherwise let v be the vertex on the top of the stack.
3. If there is no vertex v' that is adjacent to v and has not been visited yet, then delete v and go to step 2 (*backtrack*). Otherwise, let v' be the first non-visited vertex that is adjacent to v.
4. Add vertex v' and edge (v, v') to T, insert v' in the stack and mark it as "visited".
5. Go to step 2.

An alternative recursive definition is as follows. We define recursively a process P applied to a given vertex v in the following way:

1. Add vertex v to T and mark it as "visited".
2. If there is no vertex v' that is adjacent to v and has not been visited yet, then return. Otherwise, let v' be the first non-visited vertex that is adjacent to v.
3. Add the edge (v, v') to T.
4. Apply P to v'.
5. Go to step 2 (*backtrack*).

The *Depth-First Search Algorithm* consists of applying the process just defined to v_1.

A pseudocode version of the algorithm is as follows:

```
procedure dfs(V,E)
  V' := {v1} // v1 is the root of the spanning tree
  E' := {} // no edges in the spanning tree yet
  w := v1
  while true do
    begin
      while there is an edge (w,v) that when added
             to T does not create a cycle in T do
        begin
          Choose first v such that (w,v)
          does not create a cycle in T
          add (w,v) to E'
          add v to V'
          w := v
        end
      if w = v1 then
```

```
17:         return(T)
18:       w := parent of w in T // backtrack
19:     end
20: end
```

Figure 7.7 shows the spanning tree obtained using the breadth-first search algorithm on the graph with its vertices ordered lexicographically: $a, b, c, d, e, f, g, h, i$.

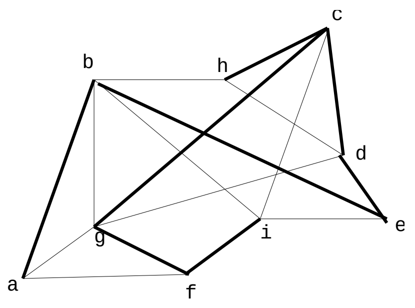

Figure 7.7. Depth-First Search.

7.2.4. Minimal Spanning Trees. Given a connected weighted tree G, its *minimal spanning tree* is a spanning tree of G such that the sum of the weights of its edges is minimum. For instance for the graph of figure 7.8, the spanning tree shown is the one of minimum weight.

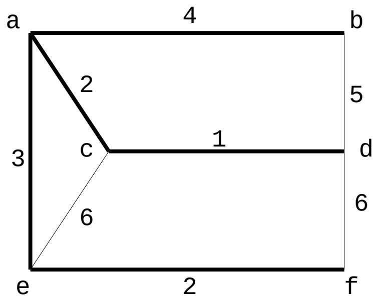

Figure 7.8. Minimum Spanning Tree.

Prim's Algorithm. An algorithm to find a minimal spanning tree is *Prim's Algorithm.* It starts with a single vertex and at each iteration adds to the current tree a minimum weight edge that does not complete a cycle.

The following is a pseudocode version of Prim's algorithm. If (x, y) is an edge in $G = (V, E)$ then $w(x, y)$ is its weight, otherwise $w(x, y) = \infty$. The starting vertex is s.

```
procedure prim(V,w,s)
  V' := {s} // vertex set starts with s
  E' = {} // edge set initially empty
  for i := 1 to n-1 do // put n edges in spanning tree
    begin
      find x in V' and y in V - V' with minimum w(x,y)
      add y to V'
      add (x,y) to E'
    end
  return(E')
end prim
```

Prim's algorithm is an example of a *greedy algorithm.* A greedy algorithm is an algorithm that optimized the choice at each iteration without regard to previous choices ("doing the best locally"). Prim's algorithm makes a minimum spanning tree, but in general a greedy algorithm does not always finds an optimal solution to a given problem. For instance in figure 7.9 a greedy algorithm to find the shortest path from a to z, working by adding the shortest available edge to the most recently added vertex, would return acz, which is not the shortest path.

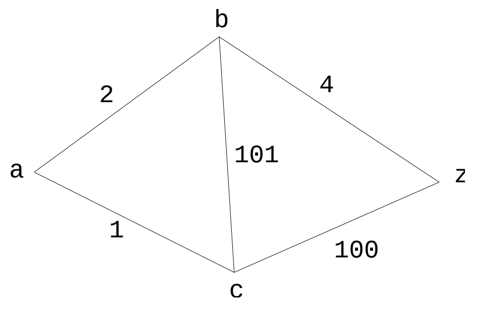

FIGURE 7.9

Kruskal's Algorithm. Another algorithm to find a minimal spanning tree in a connected weighted tree $G = (V, E)$ is *Kruskal's Algorithm.* It starts with all n vertices of G and no edges. At each iteration we add an edge having minimum weight that does not complete a cycle. We stop after adding $n - 1$ edges.

```
procedure kruskal(E,w,n)
  V' := V
  E' := {}
  while |E'| < n-1 do
    begin
      among all edges not completing a cycle in T
      choose e of minimum weight and add it to E
    end
  T' = (V',E')
  return(T')
end kruskal
```