The length $l(p)$ of a path p is the number of edges in the path. The *distance* between two vertices v_1 and v_2 written $d(v_1, v_2)$ is the length of the shortest path connecting the vertices, $d(v_1, v_2) = min\{l(p) \,|\, p$ is a path connecting v_1 and $v_2\}$. We often represent the distances of all pairs of vertices in a graph with a matrix.

Example The array below shows the distance matrix for the graph G_1.

$$\mathbf{D} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 2 \\ 1 & 0 & 2 & 1 & 2 & 2 \\ 1 & 2 & 0 & 1 & 2 & 2 \\ 1 & 1 & 1 & 0 & 2 & 1 \\ 1 & 2 & 2 & 2 & 0 & 3 \\ 2 & 2 & 2 & 1 & 3 & 0 \end{pmatrix}$$

The *diameter* of a graph G is the maximum distance between any two vertices in G, $diam(G) = max\{d(v_1, v_2) \,|\, v_1, v_2$ are vertices of $G\}$.

Example The diameter of G_1 is 3. The diameter of the first subgraph in Figure 4 is 2 and the diameter of the second subgraph is 4.

Another matrix often associated to a graph G is the *adjacency matrix* which has entry $ij = 1$ if $(v_i, v_j) \in E(G)$ and equal to 0 otherwise.

Example Below is the adjacency matrix for G_1.

$$\mathbf{D} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

A *rooted tree* is a graph with a distinguished vertex called the *root* of the tree. Often we draw a rooted tree with the root on the top and the other vertices in layers below corresponding to their distance from the root. The collection of vertices at distance i from the root is called the ith *level* of the tree. The *depth* of a rooted tree is the number of levels of the tree.
Example Figure 15 is the tree of Figure 11 rooted at vertex 1.

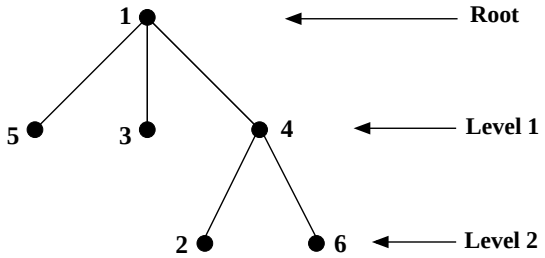

Figure 15: A rooted tree with depth 2.

For a vertex of a rooted tree at level i, its neighbor on level $i-1$ is called its *parent* and its neighbors on level $i+1$ is called its *children*. A rooted *binary tree* is a rooted tree such that each vertex has at most 2 children.
Example In Figure 15, vertex 1 has children $\{3, 4, 5\}$ and vertex 4 has children $\{2, 6\}$. This tree is not a binary tree because vertex 1 has three children. If we removed vertex 5, the resulting tree would be binary.

A *spanning tree* $T = (V', E')$ of a graph $G = (V, E)$ is a subgraph such that T is a tree and $V' = V$.
Example In Figure 16 we show three different spanning trees of G_1.

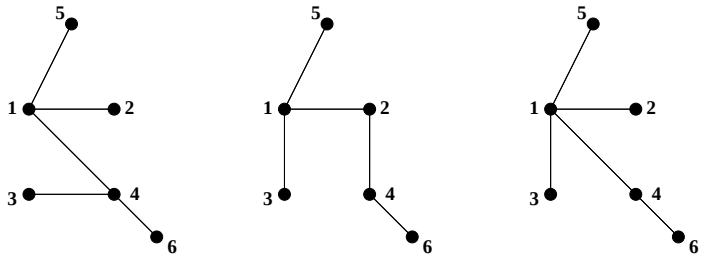

Figure 16: Spanning trees.

Next we consider ways of finding spanning trees of arbitrary simple graphs. Here we consider two algorithms, breadth first search and depth first search. For either of these algorithms, we must have an ordering on the vertices of the graph. In most of our examples, the vertices have been labeled by $\{1, 2, \ldots, n\}$ where n is the number of vertices. With this kind of labeling, we have a natural ordering where the vertex labeled 1 comes first then 2 and so on. These algorithms will output a rooted spanning tree.

Depth First Search Algorithm

Step 1: Let v be a variable. Let v_1 be the vertex of smallest label in G. Initialize v to v_1. Define T to be the tree consisting of just the vertex v.

Step 2: Find the vertex w of smallest label such that (v, w) is an edge in G and we have not yet considered w.

- If such a w exists add the edge (v, w) to T. Reassign vertex w to the variable v and repeat step 2.
- If no such w exists go to step 3.

Step 3: If $v = v_1$ then stop, T is a spanning tree.

Step 4: If $v \neq v_1$ then assign its parent in T to v and repeat step 2.

Example Let us run depth first search on the graph G_1. Vertex 1 is the vertex of smallest label. This will be the root of our tree. So we assign $v = 1$ and T is the tree of a single vertex 1. Next we look for the neighbor of 1 with smallest label, this is vertex 2. Therefore we add the edge $(1, 2)$ to T and set $v = 2$. Now we look for the neighbor of 2 with smallest label. This is vertex 1 but we have already considered vertex 1 so we look for the next smallest which is vertex 4. Again we add edge $(2, 4)$ to T and set $v = 4$. From 4, vertex 3 is the smallest labeled neighbor that we have not yet considered. Hence we add the edge $(3, 4)$ to T and set $v = 3$. Now we see that all neighbors of 3 have already been considered and we must go to step 3. At this point, $v \neq v_1$ so we move to step 4. The parent of 3 in T is vertex 4 so we set $v = 4$ and move back to step 2. There is a neighbor of 4 that has not yet been considered, vertex 6. Thus we add the edge $(4, 6)$ to T and set

$v = 6$. Vertex 6 has no other neighbors, so we will have to move back to its parent, vertex 4. All neighbors of 4 have been considered, so we must move to its parent, vertex 2. Similarly, we must move back to the parent of vertex 2 which is vertex 1. From here, there is a neighbor which has not been considered, vertex 5. We add the edge $(1, 5)$ to T and set $v = 5$. From here we will backtrack to 1 and in step 3 v will equal v_1 and we are done!

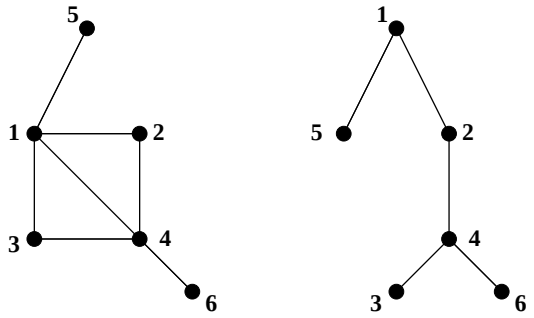

Figure 17: A spanning tree of G_1 found using DFS.

Breadth First Search Algorithm

Step 1 Let L be an ordered set initialized to the set containing 1. Let $i = 1$.

Step 2 Find all neighbors of i not already in T. Add them in order to the end of L. Add all edges $\{(i, x) \mid x \in N(i) \text{ and } x \notin T\}$ to T.

- If $|T| = n - 1$ stop.

Step 3 Remove the first element of L, let i equal the new first element of L. Go to step 2.

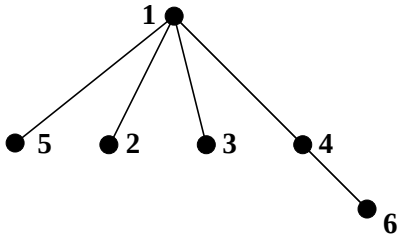

Figure 18: A spanning tree of G_1 found using BFS.

A *weighted graph* is a graph such that to each edge e there is an associated real number $w(e)$ called the *weight* of the edge. Given a graph with weights on each of its edges, we want to determine a spanning tree with the smallest total weight, this is called a *minimal spanning tree*. The total weight of a tree (or any graph) is the sum of the weights of its edges. Here we consider a greedy algorithm, Kruskal's Algorithm.

Kruskal's Algorithm

Step 1 Take an edge $e \in G$ such that $w(e)$ is minimal.

- If $T \cup e$ is a tree, add e to T.

Step 2 Remove e from G.

Step 3 If $|T| = n - 1$ stop.

- Otherwise, go to step 1.

Example: For the graph in Figure 19, the edges of the spanning tree were added in the following order: $(1,3)$, $(2,4)$, $(4,6)$, $(1,4)$, $(1,5)$.

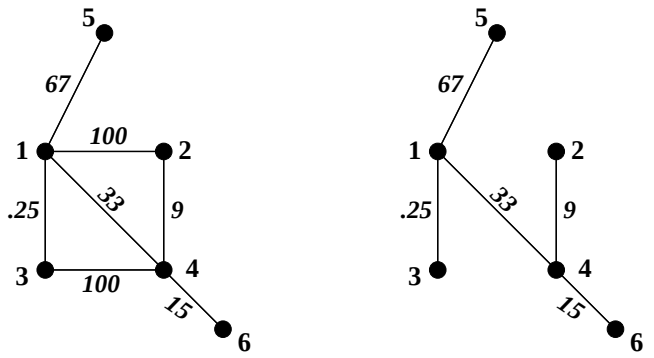

Figure 19: Minimal spanning tree.

Suppose we have a directed graph where the vertices represent tasks and the edges represent dependence. Namely, the edge (i, j) means that task j cannot be accomplished until task i is complete. Given such a graph, we want to be able to determine an order to complete all tasks. We will call such an order a *total order* for a directed graph. First however, we must ask if such an order is possible. If for example the graph contained a directed cycle we would not be able to find such an order. A directed graph with no directed cycle is called an *acyclic* graph.

Example: The first graph of Figure 20 is not acyclic, $(4, 1)$, $(1, 3)$, $(3, 4)$ is a directed cycle. In the second graph, the edge $(1, 3)$ has been replaced with the edge $(3, 1)$ and the graph is now acyclic.

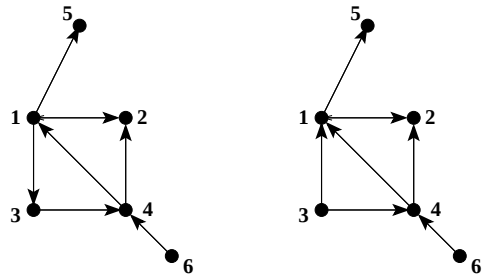

Figure 20: One non-acyclic and one acyclic graph.

Proposition. *A directed graph has a total order if and only if it is acyclic.*

Now suppose we have an acyclic graph, next we give an algorithm for finding a total order called topological sort.

Topological Sort

Step 1: Let $i = 1$ and G be an acyclic graph on n vertices.

Step 2: Find a vertex v_i such that $outdeg(v_i) = 0$.

- If $i = n$ then stop. $v_n < v_{n-1} < \ldots v_2 < v_1$ is a total order.
- Otherwise, remove v_i from G. Let $i = i + 1$. Repeat step 2.

Example In the second graph of Figure 20, one total ordering found by topological search is: $3 < 6 < 4 < 1 < 2 < 5$.