GRAPH THEORY – LECTURE 5: SPANNING TREES

Abstract. Several different problem-solving algorithms involve growing a spanning tree, one edge and one vertex at a time. All these techniques are refinements and extensions of the same basic tree-growing scheme given in §4.1. §4.2 presents depth-first and breadth-first search. §4.3 introduces two algorithmic computations, finding a minimum-weight spanning tree and finding a shortest path.

Outline

1. Tree-Growing

Def 1.1. For a given tree T in a graph G, the edges and vertices of T are called ***tree edges*** and ***tree vertices***, and the edges and vertices of G that are not in T are called ***non-tree edges*** and ***non-tree vertices***.

Def 1.2. A ***frontier edge*** for a given tree T in a graph is a non-tree edge with one endpoint in T, called its ***tree endpoint***, and one endpoint not in T, its ***non-tree endpoint***.

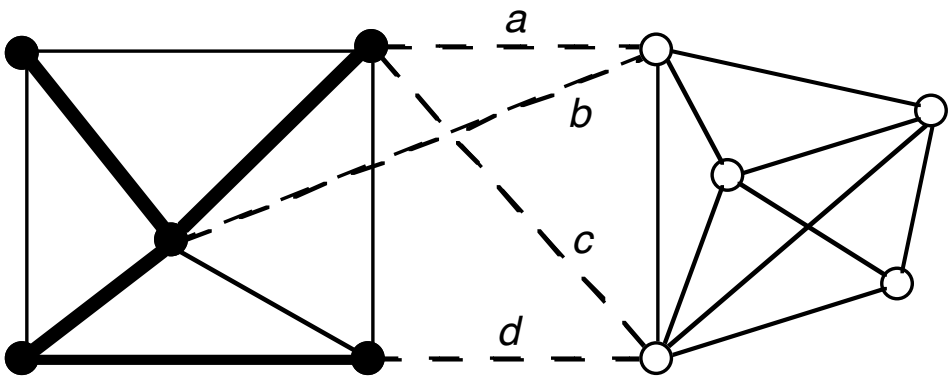


Figure 1.1. A tree with frontier edges a, b, c, and d.

Proposition 1.1. *Let T be a tree in a graph G, and let e be a frontier edge for T. Then the subgraph of G formed by adding edge e to tree T is a tree.* □

Choosing a Frontier Edge

Def 1.3. Let T be a tree subgraph of a graph G, and let S be the set of frontier edges for T. The function ***nextEdge(G,S)*** (usually deterministic) chooses and returns as its value the frontier edge in S that is to be added to tree T.

Def 1.4. After a frontier edge is added to the current tree, the procedure ***updateFrontier(G,S)*** removes from S those edges that are no longer frontier edges and adds to S those that have become frontier edges. (See Fig 1.2 below.)

Table 1.1. The generic tree-growing algorithm.

Algorithm: Tree-Growing(G, v)
Input: a connected graph G, a starting vertex $v \in V_G$, and a selection-function $nextEdge$.
Output: an ordered spanning tree T of G with root v.

Initialize tree T as vertex v.
Initialize S as the set of proper edges incident on v.
While $S \neq \emptyset$
    Let $e = nextEdge(G, S)$.
    Let w be the non-tree endpoint of edge e.
    Add edge e and vertex w to tree T.
    $updateFrontier(G, S)$.
Return tree T.

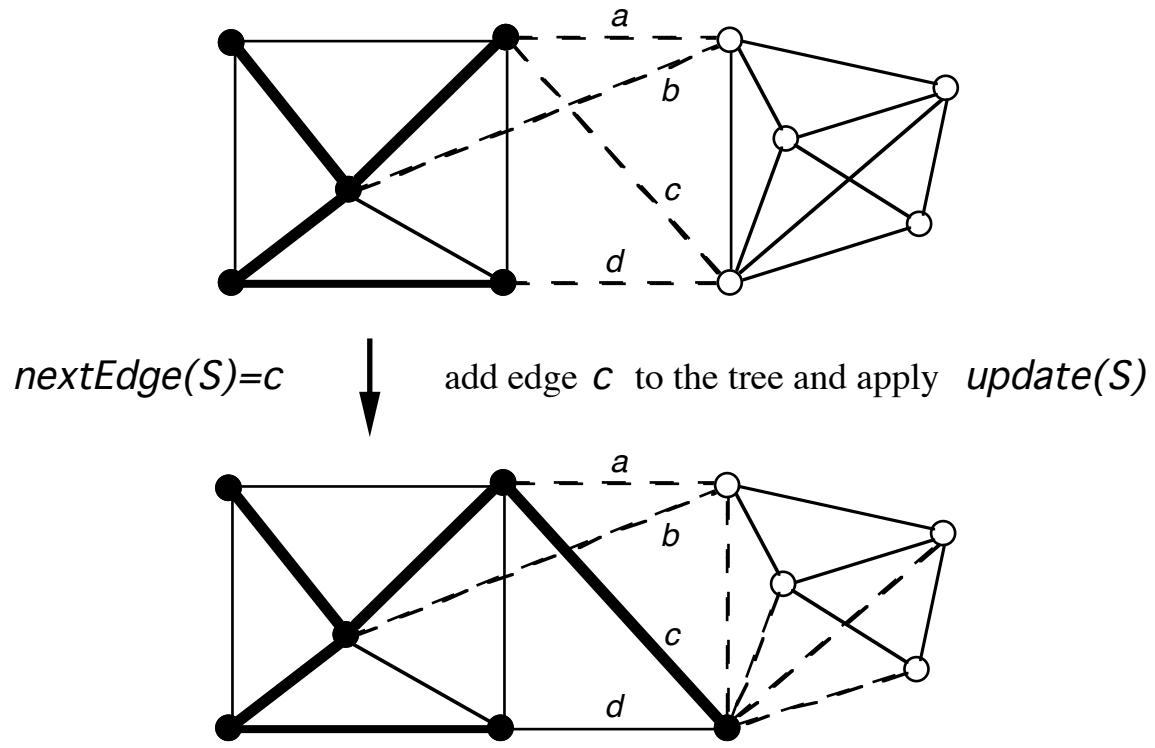


FIGURE 1.2. Result after adding edge c to the tree; note deletion of edge d from the frontier.

Discovery Order of the Vertices

Def 1.5. The ***discovery order*** is a listing of the vertices of graph G in the order in which they are added (discovered) as spanning tree T is grown.

Proposition 1.2. *The output tree T produced by a Tree-Growing is an ordered tree w.r.t. the discovery order of its vertices.* □

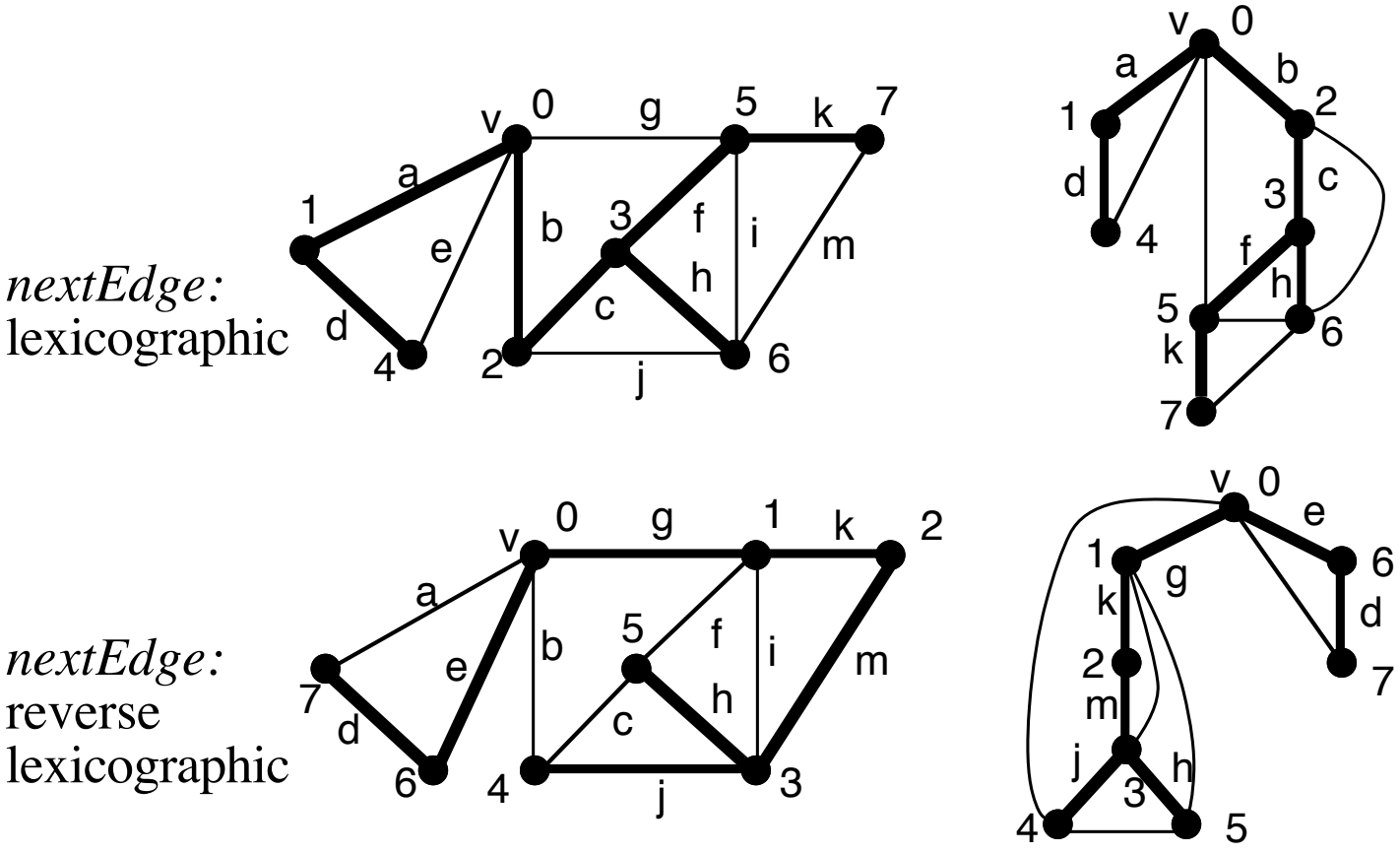


Figure 1.3. Output from two instances of Tree-Growing.

Example 1.1. Both output trees of Fig 1.3 start at v. The right-figs are the left-figs redrawn to display the spanning trees as ordered trees. Notice that the left-to-right order of the children of each vertex is consistent with the discovery order, as asserted by Prop 1.2.

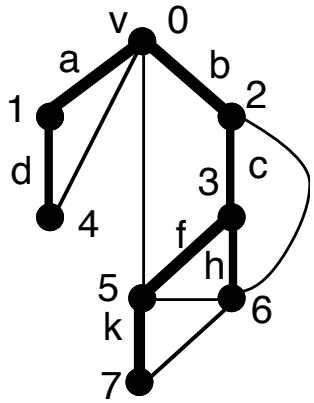
Two Kinds of Non-Tree Edges

The non-tree edges that appear in both output trees in Figure 1.3 above, or any other output tree, fall into two categories.

Def 1.6. Two vertices in a rooted tree are ***related*** if one is a descendant of the other.

Def 1.7. For a given output tree grown by Tree-Growing, a ***skip-edge*** is a non-tree edge whose endpoints are related; a ***cross-edge*** is a non-tree edge whose endpoints are not related.

Example 1.1, continued: For the output tree at the right, which is reproduced from the top half of Figure 1.3, there are three skip-edges (04, 05, 26) and two cross-edges (56, 67). For the other output tree, there are four skip-edges and one cross-edge.



DFS and BFS as Tree-Growing

Preview of §4.2: The *depth-first* and *breadth-first* searches use opposite versions of $nextEdge$.

- ***depth-first***: $nextEdge$ selects a frontier edge whose tree endpoint was most recently discovered.
- ***breadth-first***: $nextEdge$ selects a frontier edge whose tree endpoint was discovered earliest (least recently).

Resolving Ties for Next-Frontier-Edge

Typically, ties are resolved by some *default priority* that is likely to be part of the implementation of the data structures involved. We will often rely on the somewhat artificial lexicographic (alphabetical) order of the edge names and/or vertex names to resolve ties in choosing the next frontier edge.

Tree-Growing in a Non-Connected Graph

Review from §2.3 The ***component of a vertex*** v in a graph G, denoted $C_G(v)$, is the subgraph of G induced on the set of vertices that are reachable from v.

Proposition 1.3. *Let tree T be the output of Tree-Growing (Algorithm 1.1) on a graph G (not necessarily connected), starting at a vertex $v \in V_G$. Then tree T spans the component $C_G(v)$.*

Proof. The result follows by induction, using Prop 1.1. □

big

Corollary 1.4. *A graph G is connected iff the output tree produced from Tree-Growing in G is a spanning tree of G.* □

Tree-Growing in a Digraph

Def 1.8. A ***frontier arc*** for a rooted tree T in a digraph is an arc whose tail is in T and whose head is not in T.

In contrast with undirected graphs, the number of vertices in the output tree for a digraph depends on the choice of a starting vertex, as the next example shows.

Example 1.4. In Fig 1.4, the number of vertices in the output tree ranges between 1 and 5, depending on the starting vertex.

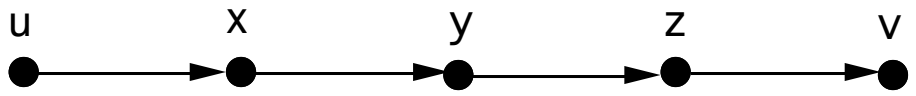


Figure 1.4. Output tree depends on the starting vertex.

Def 1.9. For tree-growing in digraphs, the non-tree edges (arcs) fall into three categories:

- A ***back-arc*** is directed from a vertex to one of its ancestors.
- A ***forward-arc*** is directed from a vertex to one of its descendants.
- A ***cross-arc*** is directed from a vertex to another vertex that is unrelated.

Def 1.10. There are two kinds of cross-arcs: a ***left-to-right cross-arc*** is directed from smaller discovery number to larger one; a ***right-to-left cross-arc*** is the opposite.

Forest-Growing

Def 1.11. A ***full spanning forest*** of a graph G is a spanning forest consisting of a collection of trees, such that each tree is a spanning tree of a different component of G.

Table 1.2. Forest-growing and component-counting.

Algorithm: Forest-Growing
Input: a graph G
Output: a full spanning forest F for G and the number $c(G)$.
Initialize forest F as the empty graph.
Initialize component counter $t := 1$
While forest F does not yet span graph G
Let $v = nextVertex(V_G - V_F)$.
By Tree-Growing, obtain spanning tree T_t of $C_G(v)$.
Add tree T_t to forest F.
$t := t + 1$
Return forest F and component count $c(G) = t$.

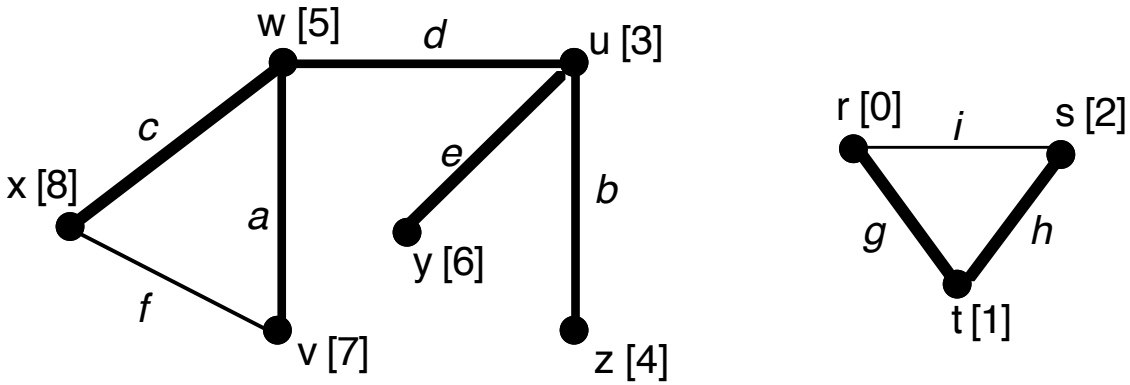


Figure 1.5. Growing a 2-component spanning forest.

2. DF and BF Search

Def 2.1. The output trees produced by the depth-first and breadth-first searches of a graph are called the ***depth-first tree*** (or ***dfs-tree***) and the ***breadth-first tree*** (or ***bfs-tree***).

As previewed in §4.1, depth-first search and breadth-first search use two opposite priority rules for the function $nextEdge$.

Depth-First Search

Def 2.2. Let S be the current set of frontier edges. The function ***dfs-nextEdge*** is defined as follows: $dfs\text{-}nextEdge(G, S)$ selects and returns as its value **the frontier edge whose tree-endpoint has the largest discovery number**.

Table 2.1. Depth-first search.

Algorithm: Depth-First Search
Input: a connected graph G, a starting vertex $v \in V_G$.
Output: an ordered spanning tree T of G with root v.
Initialize tree T as vertex v.
Initialize S as the set of proper edges incident on v.
While $S \neq \emptyset$
Let $e = dfs\text{-}nextEdge(G, S)$.
Let w be the non-tree endpoint of edge e.
Add edge e and vertex w to tree T.
$updateFrontier(G, S)$.
Return tree T.

Def 2.3. The discovery number of each vertex w for a dfs is called the ***dfnumber*** of w and is denoted $dfnumber(w)$.

Example 2.1. Fig 2.1 shows $dfnumbers$ in parentheses, and the tree edges are drawn in bold. Notice that there are no cross-edges. More specifically, for each non-tree edge, the endpoint with the smaller $dfnumber$ is an ancestor of the other endpoint. The next proposition shows that this is always the case for a depth-first search of an *undirected* graph.

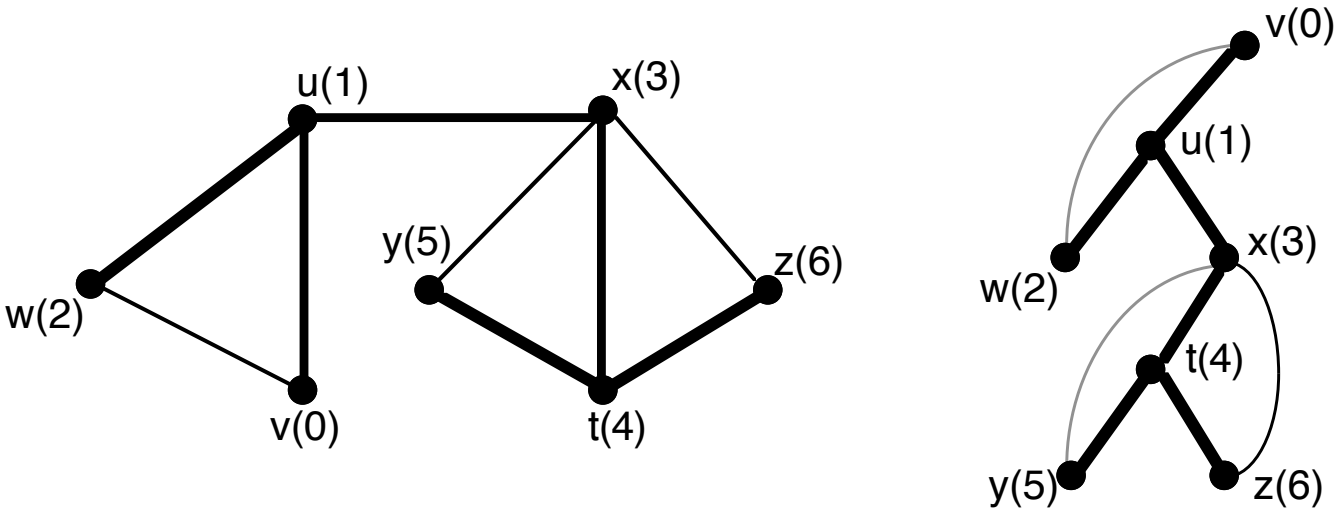


FIGURE 2.1. Dfs using lexicographic order for ties.

Proposition 2.1. *Depth-first search trees have no cross-edges.*

Proof. Let T be the output tree produced by a dfs, and let e be a non-tree edge with endpoints x and y, such that

$$dfnumber(x) < dfnumber(y)$$

At the point when the search discovers the vertex x, the edge e becomes a frontier edge.

Since edge e never becomes a tree edge, the df-search must discover vertex y before it backtracks to vertex x for the last time. Thus, y is in the subtree (of the dfs-tree T) that is rooted at x. Hence, vertex y is a descendant of vertex x. □

Depth-First Search in a Digraph

The depth-first search in a digraph is Algorithm 2.1 with the function *dfs-nextArc* replacing *dfs-nextEdge*. Also, we do not assume that the input digraph is strongly connected, so the dfs-tree produced will not necessarily be a spanning tree.

Def 2.4. The function ***dfs-nextArc*** selects and returns as its value the frontier arc whose tree-endpoint has the largest dfnumber.

Proposition 2.2. *When a depth-first search is executed on a digraph, the only kind of non-tree arc that cannot occur is a left-to-right cross arc.*

Proof. See Exercises. □

Breadth-First Search

In the breadth-first search, the search "fans out" from the starting vertex and grows the tree by selecting frontier edges incident on vertices as close to the starting vertex as possible.

Algorithm 2.2 below uses the following version of the function $nextEdge$.

Def 2.5. Let S be the current set of frontier edges. The function ***bfs-nextEdge*** is defined as follows: $bfs\text{-}nextEdge(G, S)$ selects and returns as its value the frontier edge whose tree-endpoint has the smallest discovery number.

Table 2.2. Breadth-first search.

Algorithm: Breadth-First Search
Input: a connected graph G, a starting vertex $v \in V_G$.
Output: an ordered spanning tree T of G with root v.
Initialize tree T as vertex v.
Initialize S as the set of proper edges incident on v.
While $S \neq \emptyset$
Let $e = bfs\text{-}nextEdge(G, S)$.
Let w be the non-tree (undiscovered) endpt of e.
Add edge e and vertex w to tree T.
$updateFrontier(G, S)$.
Return tree T.

Example 2.2. Fig 2.2 below compares the results of dfs and bfs by displaying typical possibilities for their partial output trees after 11 iterations, starting at vertex v.

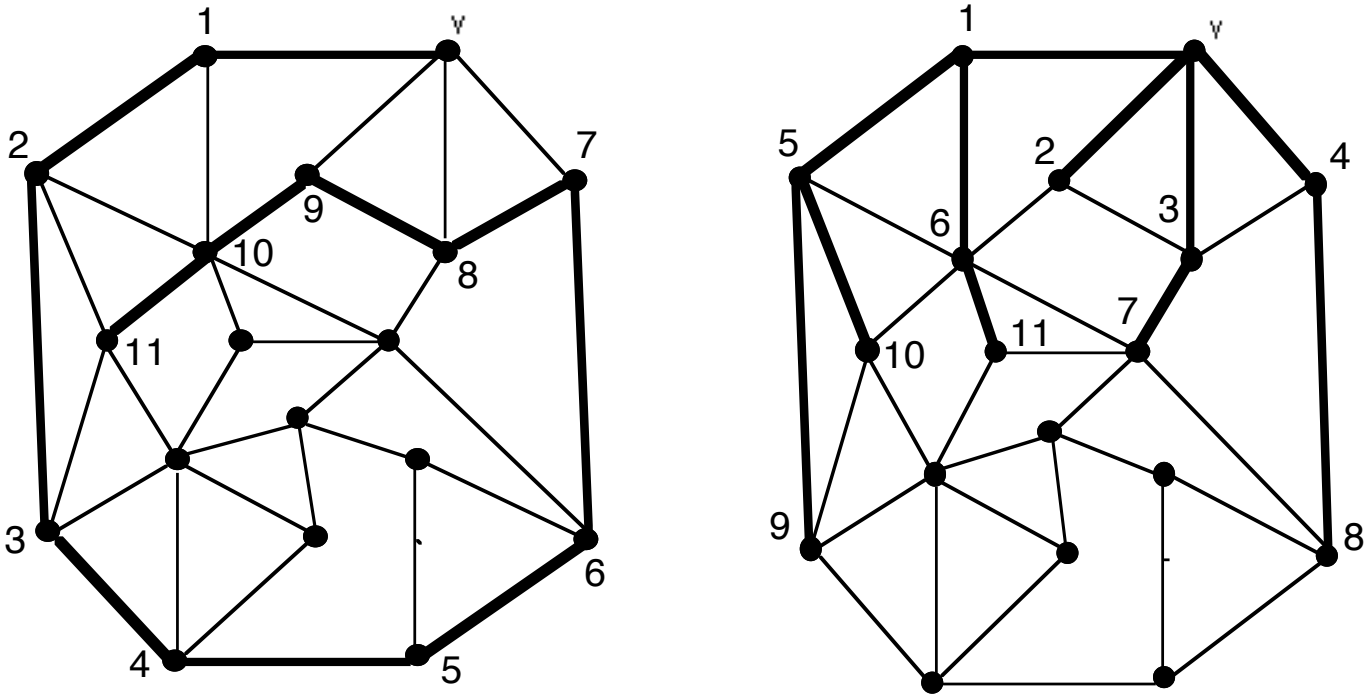


FIGURE 2.2. Dfs and bfs after 11 iterations.

Example 2.3. Fig 2.3 shows the result of a bfs applied to the graph from Example 2.1, again starting at vertex v. Observe that the non-tree edges are all cross-edges, which is opposite from a dfs tree. Also notice that the bfs tree is a shortest-path tree (§3.2). Both these properties hold in general, as asserted by the next two propositions.

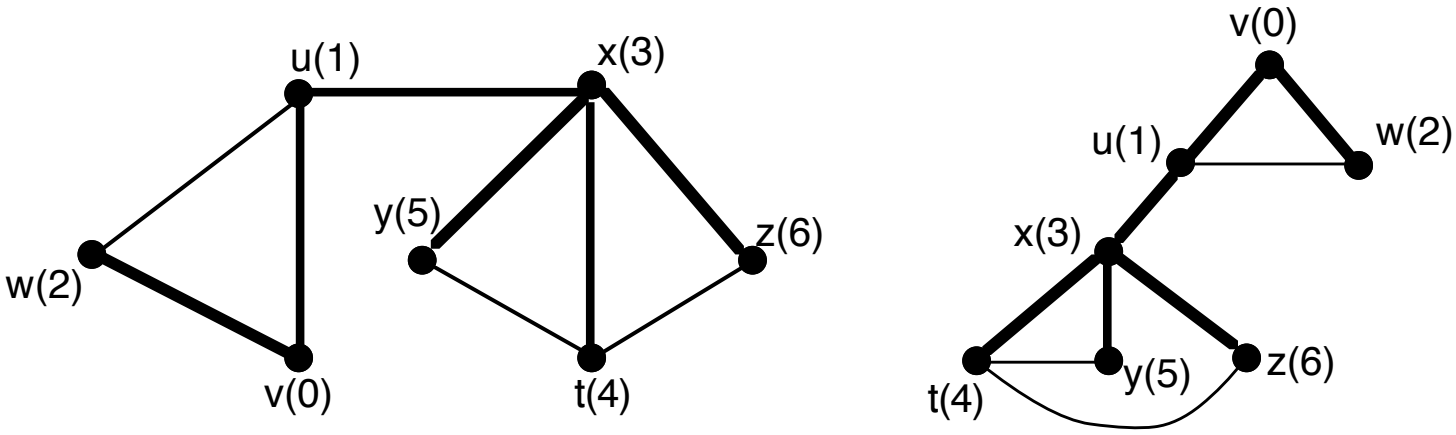


FIGURE 2.3. Bfs using lexicographic order for ties.

Proposition 2.3. *When breadth-first search is applied to an undirected graph, the endpoints of each non-tree edge are either at the same level or at consecutive levels.*

Proof. The result uses an argument analogous to the one given in the proof of Prop 2.1. □

Proposition 2.4. *Any bfs-tree produced by Algo 2.2 is a shortest-path tree for the input graph.* (See Exercises.) □

Remark 2.1. Whereas the *stack* (LIFO) is the appropriate data structure to store the frontier edges in a dfs, the *queue* (FIFO) is most appropriate for the bfs.

3. MST and Shortest Paths

Finding the MST: Prim's Algorithm

Min-Spanning-Tree Problem: In a conn weighted graph, find a spanning tree whose total edge-weight is a minimum. (By Cayley's formula (§3.7), the number of different spanning trees of an n-vertex graph could be as many as n^{n-2}.)

Def 3.1. Let S be the current set of frontier edges. The function ***Prim-nextEdge*** (G, S) selects and returns as its value the frontier edge with smallest edge-weight.

Table 3.1. Prim's minimum spanning-tree.

Algorithm: Prim Minimum Spanning-Tree
Input: a weighted conn graph G and starting vertex v.
Output: a minimum spanning tree T.
Initialize tree T as vertex v.
Initialize S as the set of proper edges incident on v.
While $S \neq \emptyset$
Let $e = Prim\text{-}nextEdge\,(G, S)$.
Let w be the non-tree endpoint of edge e.
Add edge e and vertex w to tree T.
$updateFrontier(G, S)$.
Return tree T.

Example 3.1. Fig 3.1 shows the MST yielded by Prim's algorithm for a graph, starting at vertex v. The discovery numbers appear in parentheses.

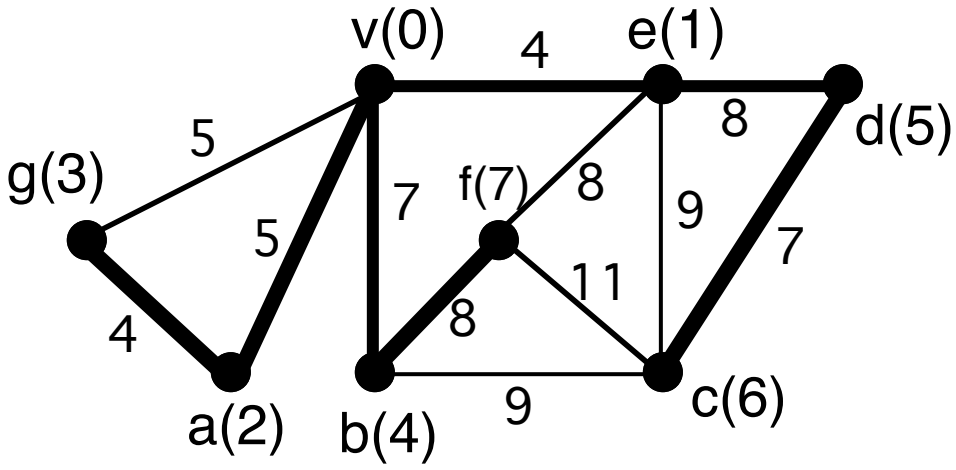


FIGURE 3.1. MST produced by Prim's algorithm.

The correctness of Prim's algorithm is an immediate consequence of the next proposition.

Proposition 3.1. *Let T_k be the Prim tree after k iterations of Prim-next-edge on a conn graph G, for $0 \leq k \leq |V_G| - 1$. Then T_k is a subtree of an MST of G.*

Proof. The assertion is trivially true for $k = 0$.

By way of induction, assume for some k, with $0 \leq k \leq |V_G| - 2$, that T_k is a subtree of an MST T of G. We consider the tree T_{k+1}.

The algorithm grew T_{k+1} during the $(k+1)^{\text{st}}$ iteration by adding to T_k a frontier edge e of smallest weight. Let u and v be the endpoints of edge e, such that endpoint u is in tree T_k and endpoint v is not.

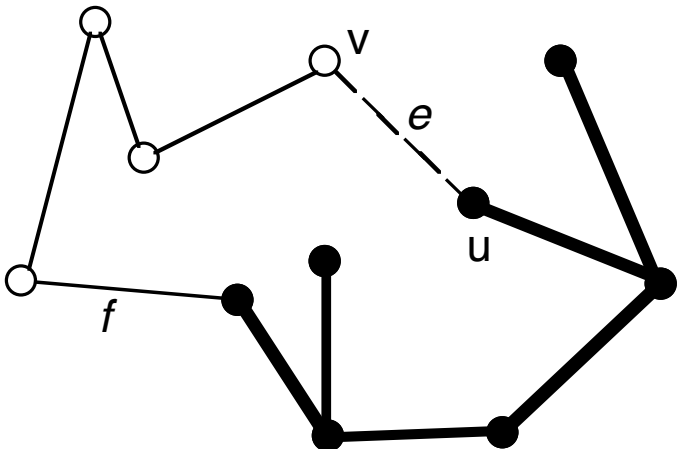


FIGURE 3.2. The tree T_k is in bold.

If the MST T contains edge e, then T_{k+1} is a subtree of T.

Alternatively, if e is not an edge in tree T, then e is part of the unique cycle contained in $T + e$. Consider the path in T from u to v that proceeds in the "long way around the cycle".

On this path, let f be the first edge that joins a vertex in T_k to a vertex not in T_k. The situation is illustrated in Fig 3.3; the black vertices and bold edges make up Prim tree T_k, the spanning tree T consists of everything except edge e, and Prim tree $T_{k+1} = (T_k \cup v) + e$.

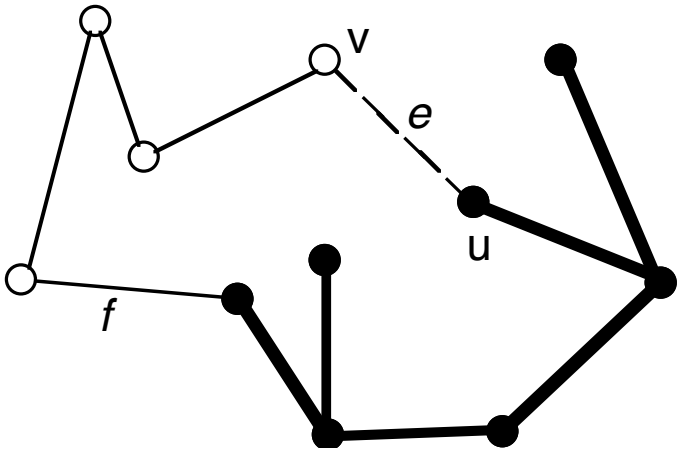


FIGURE 3.3. The tree T_k is in bold.

Since f was a frontier edge at the beginning of the $(k+1)^{\text{st}}$ iteration, we have $w(e) \leq w(f)$ (since the Prim algorithm chose e). The tree $\hat{T} = T + e - f$ clearly spans G, and T_{k+1} is a subtree of $\hat{T}$ (since f was not part of T_k). Finally,

$$w(\hat{T}) \;=\; w(T) + w(e) - w(f) \;\leq\; w(T)$$

which shows that $\hat{T}$ is an MST of G. □

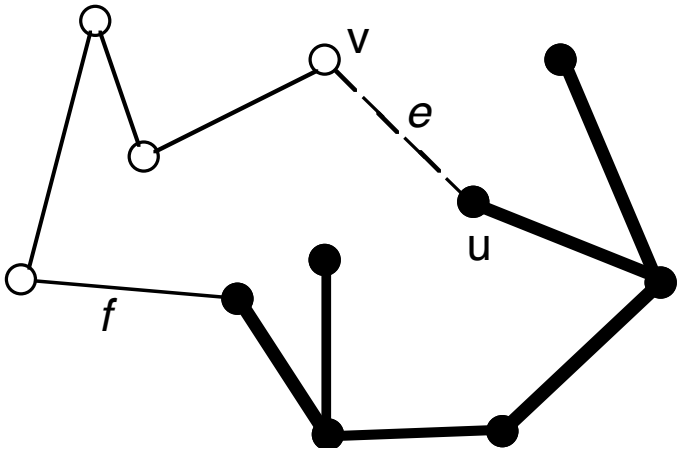


FIGURE 3.4.

Corollary 3.2. *When Prim's algorithm is applied to a connected graph, it produces a minimum spanning tree.*

Proof. Prop 3.1 implies that the Prim tree resulting from the final iteration is a minimum spanning tree. □

The Steiner-Tree Problem

If instead of requiring that the minimum-weight tree *span* the given graph G, i.e., that it contain every vertex of G, we require that it include an *arbitrarily prescribed subset* U of the vertices, we get the ***Steiner-tree problem***.

Observe that if $U = V_G$, then the Steiner-tree problem reduces to the minimum-spanning tree problem.

Example 3.2. Figure 3.5 shows a weighted graph and a Steiner tree (with bold edges) for the vertex subset $U = \{x, y, z\}$.

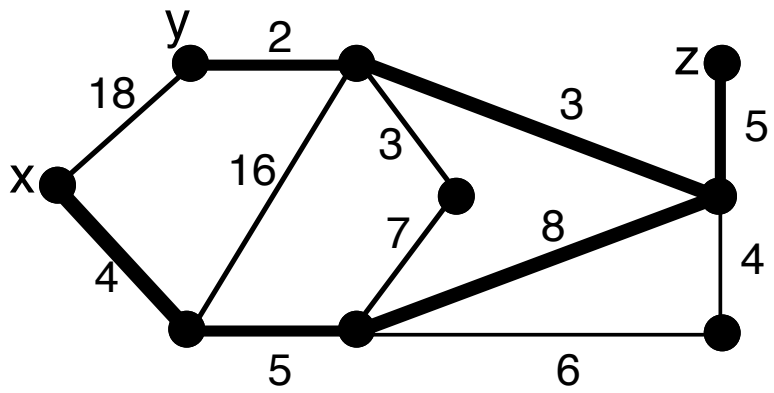


Figure 3.5. Steiner tree for $U = \{x, y, z\}$.

Remark 3.1. The Steiner-tree problem often arises in network-design and wiring-layout problems. It does not lend itself to the tree-growing strategy that we used for the minimum-spanning tree problem. A version of the Steiner-tree problem is discussed in §8.6 in the context of *graph drawings*.

Shortest Path: Dijkstra's Algorithm

Shortest-Path Problem: In a conn (non-negatively) weighted graph, find a path from s to t whose total edge-weight is minimum, i.e., a shortest s-t path.

Remark 3.2. If all edge-weights are equal, then the problem can be solved by breadth-first search (Algo 2.2).

Def 3.2. Let S be the current set of frontier edges. The process ***Dijkstra-nextEdge***(G, S) selects and returns as its value the frontier edge whose non-tree endpoint is closest to starting vertex s.

Table 3.2. Dijkstra's Shortest Path

Algorithm: Dijkstra Shortest Path
Input: a weighted conn graph G and starting vertex s.
Output: a shortest-path tree T with root s.
Initialize tree T as vertex s.
Initialize S as the set of proper edges incident on s.
While $S \neq \emptyset$
Let $e := Dijkstra\text{-}nextEdge\,(G, S)$.
Let w be the non-tree endpoint of edge e.
Add edge e and vertex w to tree T.
$updateFrontier\,(G, S)$.
Return tree T.

Notation: For each tree vertex x, let $dist[x]$ denote the distance from vertex s to x.

Example 3.3. Fig 3.6 shows the shortest-path tree produced by Dijkstra for a graph, starting at vertex s. In the parentheses at each vertex v, the discovery number appears first and $dist[v]$ appears second.

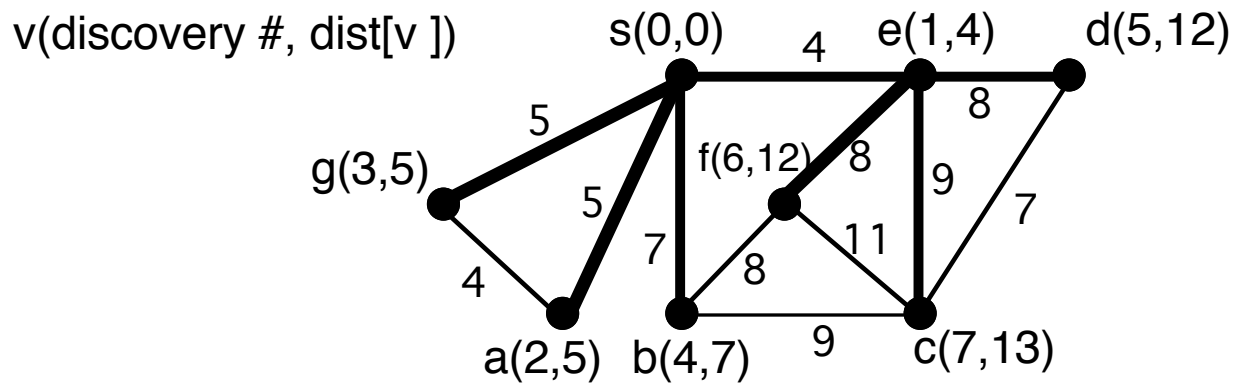


FIGURE 3.6. A Dijkstra shortest-path tree.

Calculating Distances as the Tree Grows

Notation: For each frontier edge e in the weighted graph, $w(e)$ denotes its edge-weight.

Notice that each tree edge q in Figure 3.6 has the following property: if x is the endpoint with the smaller discovery number and y is the other endpoint, then

$$dist[y] \;=\; dist[x] + w(q)$$

Thus, when q was the frontier edge selected by the procedure $Dijkstra\text{-}nextEdge$, the value of $dist[x] + w(q)$ must have been a minimum over all frontier edges in that iteration.

This suggests the following definition, which enables the function $Dijkstra\text{-}nextEdge$ to be efficiently calculated. It is also used in the proof of correctness of Dijkstra's algorithm (Theorem 3.3 below).

Def 3.3. Let e be a frontier edge of the Dijkstra tree grown so far, and let x be the tree endpoint of e. The ***P-value*** of edge e, denoted $P(e)$, is given by

$$P(e) \;=\; dist[x] + w(e)$$

Thus, $Dijkstra\text{-}nextEdge(G, S)$ selects and returns as its value the edge e^* such that $P(e^*) = \min_{e \in S}\{P(e)\}$. (As usual, if there is more than one such edge, then $Dijkstra\text{-}nextEdge(G, S)$ selects the one determined by the default priority.)

Correctness of Dijkstra's Algorithm

The following theorem establishes the correctness of Dijkstra's algorithm. Its proof is similar to the one used to show the correctness of Prim's algorithm.

Theorem 3.3. *On a conected graph G, let tree T_j be the Dijkstra tree after j iterations of Dijkstra's algorithm, where $0 \leq j \leq |V_G| - 1$. Then for each v in T_j, the unique s-v path in T_j is a shortest s-v path in G.*

Proof. Similar to Prim. See text. □

7. Supplementary Exercises

Exercise 2 In how many different spanning trees of the complete graph K_n does a given edge e lie?

Hint (not in text): The sum of # edges over all spanning trees is $(n-1)\cdot n^{n-2}$. Divide by the # edges in K_n. Sol: $2n^{n-3}$.

Exercise 4 Draw an example of a connected, simple graph G with a spanning tree T indicated by thickened edges, such that no matter what root or local ordering is selected, T could not possibly be either the BFS-tree or the DFS-tree. Explain briefly why not.

Hint (not in text): Try the tree of deg seq 11123 in K_5.