3 Applications of Trees

Growing a tree in a graph is the most efficient way of searching through the vertices of the graph. Most graph algorithms involve some kind of tree-growing procedure. We will consider several such algorithms in this chapter, and also in subsequent chapters. We first describe a way of measuring how efficient an algorithm is.

3.1 Complexity of graph algorithms

Suppose we have an algorithm which will solve a particular problem for any given graph G, for example determining whether G is connected or not. We would expect the time that the algorithm takes to solve the problem to be a function of the size of G, that is to say a function of the numbers of vertices and edges in G. We say *the algorithm has complexity of order* $f(|V(G)|, |E(G)|)$ for some function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ if the time taken by the algorithm to solve the problem for any graph G is at most a constant times $f(|V(G)|, |E(G)|)$. Equivalently we may use 'big O' notation and say that the algorithm has complexity $O(f(|V(G)|, |E(G)|))$. The fastest algorithms have complexity bounded by a linear function of the numbers of vertices and edges in G i.e. have complexity $O(|V(G)| + |E(G)|)$. We say that any algorithm whose complexity is bounded by a polynomial function of $|V(G)|$ and $|E(G)|$ to be a *polynomial* or (*efficient*) algorithm. Algorithms which can run for a time which is an exponential function of $|V(G)|$ or $|E(G)|$ are not efficient.

3.2 Finding the connected component of a graph containing a given vertex

Suppose we are given a graph G and a vertex v of G. We find the connected component containing v by growing a maximal tree T starting at v. We refer to v as the *root vertex* of the tree. We use the following iterative procedure. In the i'th step of the iteration we construct a tree T_i with vertices labelled $x_1, x_2, \ldots, x_i$.

Basic Tree Growing Algorithm

Initial Step Put $x_1 := v_1$ and let T_1 be the tree with $V(T_1) = \{x_1\}$ and $E(T_1) = \emptyset$.

Iterative Step Suppose we have constructed a tree T_i with $V(T_i) = \{x_1, x_2, \ldots, x_i\}$ for some $i \geq 1$.

- If some edge e of G is incident with a vertex x_j of T_i and a vertex y of $G - T_i$ then put $x_{i+1} = y$ and $T_{i+1} := T_i + x_{i+1} + e$.
- If no edge G is incident with both a vertex of T_i and a vertex of $G - T_i$ then STOP. Put $T = T_i$ and output T.

The final tree T will contain all vertices which belong to the same connected component H as v. To construct H we add to T all edges of G whose end vertices both belong to T.

The iterative step in the basic tree growing algorithm can be made more precise by choosing the vertex x_j of T_i such that:

- j is as small as possible - this is called *breadth first search*.
- j is as large as possible - this is called *depth first search*.

It is easy to see that the above algorithm is efficient. In each iteration we check at most all the edges of G incident with the vertices of T_i to see if there is an edge from T_i to $G - T_i$, and hence we check at most $|E(G)|$ edges. Since each iteration increases the number of vertices of T_i, the number of iterations is at most $|V(G)|$. Thus the complexity of the algorithm is $O(|V(G)| \times |E(G)|)$. In fact, it is straightforward to implement the algorithm in such a way that it has complexity $O(|V(G)| + |E(G)|)$ since there is no need to check any edge more than once.

3.3 Finding the connected components of a graph

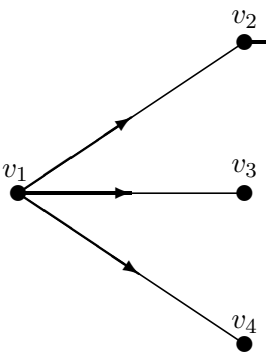
We know by Lemma 1.10 that the connected components of a graph G have no vertices in common. So we can find the connected components of G by first choosing a vertex $v_1 \in V(G)$ and using the algorithm of Section 3.2 to find the connected component H_1 of G which contains v_1. If $H_1 = G$ then G is connected. If $H_1 \neq G$ then we choose $v_2 \in V(G) - V(H_1)$ and use the algorithm of Section 3.2 to find the connected component H_2 of G which contains v_2. We continue until all vertices of G belong to some connected component.

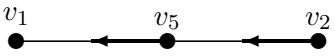
3.4 Finding the strongly connected components of a digraph

Let D be a digraph. Choose $v_1 \in V(D)$. We find all vertices which can be reached along directed paths starting at v_1 by grow a maximal tree T^+

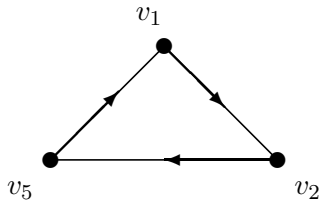
rooted at v_1 such that all arcs of T_1^+ are directed away from v_1. To do this we modify the recursive step in the basic tree growing algorithm by choosing a vertex x_j of T such that there is an arc in D from x_j to some vertex v in $V(D) - V(T)$. We then find all vertices which can reach v_1 along directed paths by growing a maximal tree T^- rooted at v_1 such that all arcs of T_1^- are directed towards v_1. To do this we modify the recursive step in the basic tree growing algorithm by choosing a vertex x_j of T such that there is an arc in D from some vertex v in $V(D) - V(T)$ to x_j. Then the vertex set of the strongly connected component H_1 of D containing v_1 is given by $V_1 := V(T_1^+) \cap V(T_1^-)$. To construct H_1 we add to V_1 all arcs of D which join vertices of V_1. We can then find the next strongly connected component H_2 by repeating the above procedure starting at a vertex $v_2 \in V(D) - V(H_1)$, and so on until all the vertices of D are contained in some strongly connected component.

Example Applying the above algorithm to the digraph of Example 1.12, we grow the following two trees from v_1.

$T_1^+ =$ 

$T_1^- =$ 

This gives $V(D_1) = V(T_1^+) \cap V(T_1^-) = \{v_1, v_2, v_5\}$ and $A(D_1) = \{v_1v_2, v_2v_5, v_5v_1\}$. Thus

$D_1 =$ 

Iterating we obtain the two other strongly connected components of D.

3.5 Finding a minimum weight spanning tree in a connected network

Recall that a *network* is a graph or digraph in which, to each edge e we associate a real number $w(e)$ called the *weight* of e. Given a network N and a subnetwork H of N, we define the weight of H, $w(H)$ to be the sum of the weights of the edges of H.

The following two algorithms find a minimum weight spanning tree in a connected network N. The first is a refinement of the basic tree growing algorithm given in Section 3.2, it grows the tree in a connected way starting from a root vertex. The second chooses edges of minimum weight in the network greedily, but may not become connected until the final edge is chosen.

Prim's Algorithm

We are given a connected network N. We choose a vertex v_1 of N. We grow a tree rooted at v_1 using the basic tree growing algorithm with the iterative step modified so that it chooses an edge e from T_i to $G - T_i$ with $w(e)$ is as small as possible.

Initial Step Put $x_1 := v_1$ and let T_1 be the tree with $V(T_1) = \{x_1\}$ and $E(T_1) = \emptyset$.

Iterative Step Suppose we have constructed a tree T_i with $V(T_i) = \{x_1, x_2, \ldots, x_i\}$ for some $i \geq 1$.

- If $V(T_i) \neq V(N)$ then choose an edge e of N which is incident with a vertex x_j of T_i and a vertex y of $G - T_i$ and is such $w(e)$ is as small as possible. Put $x_{i+1} = y$ and $T_{i+1} := T_i + x_{i+1} + e$.
- If $V(T_i) = V(N)$ then STOP. Put $T = T_i$ and output T.

It can be shown that the above algorithm has complexity $O(|V(N)| \times |E(N)|)$ in exactly the same way as for the basic tree growing algorithm, and hence is an efficient algorithm. The following implementation of the algorithm, suggested by Prim in 1957, uses a labeling procedure so that each edge of N need only be considered once. This reduces the complexity to $O(|V(N)|^2 + |E(N)|)$.

It is easiest to describe the algorithm for *simple* networks i.e. networks without loops or multiple edges. If N is not simple then we delete all loops

in N and replace each multiple edge by a single edge with weight equal to the smallest weight in the multiple edge. This preliminary step will not change the weight of a minimum weight spanning tree of N and can be performed in time $O(|E(N)|)$. Henceforth we assume N is simple. This means we can uniquely represent each edge by its pair of end vertices. We write uv to mean the edge of N with end vertices u and v and use $w(uv)$ to denote the weight of this edge.

In the i'th step of the iteration, we have a tree T_i with $V(T_i) = \{x_1, x_2, \ldots, x_i\}$ and each vertex $y \in V(N) - V(T_i)$ is labeled with an ordered pair $label_i(y) = [x, k_i(y)]$ where $x \in V(T_i)$, yx is an edge of minimum weight from y to T_i, and $k_i(y) = w(yx)$. (If there is no edge from y to T_i then we have $label_i(y) = [x_1, \infty]$.)

Initial Step Choose $v \in V(N)$. Put $x_1 = v$ and let T_1 be the tree with $V(T_1) = \{x_1\}$ and $E(T_1) = \emptyset$. For $y \in V(N) - V(T_1)$ put

$$label_1(y) = \begin{cases} [x_1, w(x_1y)] & \text{if } y \text{ is adjacent to } x_1, \\ [x_1, \infty] & \text{otherwise.} \end{cases}$$

Iterative Step Suppose we have constructed a tree T_i with $V(T_i) = \{x_1, x_2, \ldots, x_i\}$ for some $i \geq 1$ and have labeled all vertices $y \in V(N) - V(T_i)$ with a label $label_i(y) = [x, k_i(y)]$.

- If $V(T_i) \neq V(N)$ then choose a vertex $\hat{y} \in V(N) - V(T_i)$ such that $label_i(\hat{y}) = [\hat{x}, k_i(\hat{y})]$ and $k_i(\hat{y})$ is as small as possible. Put $x_{i+1} = \hat{y}$ and $T_{i+1} := T_i + x_{i+1} + \hat{x}x_{i+1}$. For each vertex $y \in V(N) - V(T_{i+1})$ put

$$label_{i+1}(y) = \begin{cases} [x_{i+1}, w(x_{i+1}y)] & \text{if } y \text{ is adjacent to } x_{i+1} \text{ and } w(yx_{i+1}) < k_i(y), \\ label_i(y) & \text{otherwise.} \end{cases}$$

- If $V(T_i) = V(N)$ then STOP. Put $T = T_i$ and output T.

3.5.1 Lemma

The time taken for Prim's algorithm to construct a spanning tree of a network N is $O(|V(N)|^2 + |E(N)|)$.

Proof The time taken to remove any loops from N and replace multiple edges by single edges is $O(|E(N)|)$. We then run Prim's algorithm on the resulting simple network. In the i'th iteration we first read all the vertex

labels, choose $\hat{y} \in V(N) - V(T_i)$ with $k_i(\hat{y})$ as small as possible, and then add $\hat{y}$ to T_i to form T_{i+1}. This requires $O(|V(N)|)$ time since there are $O(|V(N)|)$ vertex labels. We next update the vertex labels for all vertices in $N - T_{i+1}$ which are adjacent to $\hat{y}$. This again takes $O(|V(N)|)$ time. Hence each iteration requires $O(|V(N)|)$ time. Since the algorithm runs for exactly $|V(N)|$ iterations, the whole iterative procedure takes $O(|V(N)|^2)$ time. Thus the total running time is $O(|V(N)|^2 + |E(N)|)$.

Note The above proof assumes that we can perform elementary arithmetic operations with numbers in constant time, *no matter how large the numbers are*. (We assume this for example when we update the vertex labels or when we choose a vertex with the smallest label.) We will make this assumption for all network algorithms considered in this course. We delay a more general discussion of the complexity of network algorithms until the end of the Chapter 4.

We next prove that Prim's algorithm does indeed produce a minimum weight spanning tree for N. We use the following lemma.

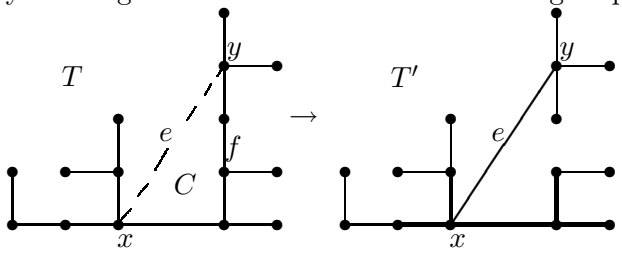
3.5.2 Lemma

Let N be a network and T_i be a tree produced in the i'th iteration of Prim's algorithm applied to N. Then T_i is contained in a minimum weight spanning tree of N.

Proof We use induction on i.
Base Case $i = 1$. Since T_1 is a tree with one vertex and no edges, it is contained in all (minimum weight) spanning trees of N.
Induction Hypothesis Suppose $i \geq 2$ and that T_{i-1} is contained in a minimum weight spanning tree of N.
Inductive Step Let T be a minimum weight spanning tree of N containing T_{i-1}. We have $T_i = T_{i-1}+y+e$ for some edge $e = xy$ of N where $x \in V(T_{i-1})$ and $y \in V(N) - V(T_{i-1})$. If $e \in E(T)$ then T_i is contained in T and we are done. Thus we may suppose that $e \notin E(T)$. Let P be a path in T from x to y and put $H = T + e$. Then $C = P + e$ is a cycle in H. Let f be the first edge of P which does not belong to T_{i-1} (as we walk along P from x to y). Note that f must exist since otherwise we would have $P \subseteq T_{i-1}$ and hence $C = P + e \subseteq T_i$, which is impossible since T_i is a tree. Let $T' = H - f$. (So $T' = T + e - f$). Since f is contained in the cycle C of H, f is not a bridge of H. Thus T' is connected. Since

$$|E(T')| = |E(T)| = |V(T)| - 1 = |V(N)| - 1 = |V(T')| - 1,$$

T' is also a spanning tree of N. Furthermore $T_i \subseteq T'$. We complete the proof by showing that T' is another minimum weight spanning tree of N.



T_{i-1} in thick lines $\qquad$ T_i in thick lines

In the i'th step of Prim's algorithm, we chose the edge e as the edge of smallest weight from T_{i-1} to $N - T_{i-1}$. Since f is also an edge from T_{i-1} to $N - T_{i-1}$, we must have $w(f) \geq w(e)$. Thus $w(T') = w(T) + w(e) - w(f) \leq w(T)$. Since T is a minimum weight spanning tree of N, we must have $w(T') = w(T)$ and T' is another minimum weight spanning tree of N.

3.5.3 Corollary

Let N be a network and T^* be a spanning tree for N produced by applying Prim's algorithm to N. Then T^* is a minimum weight spanning tree of N.

Proof It follows from Lemma 3.5.2 that T^* is contained in a minimum weight spanning tree T for N. Since both T^* and T are spanning trees of N we must have $T^* = T$.

Kruskal's Algorithm

Kruskal (1956) gave a different algorithm for constructing a minimum weight spanning tree in a network. Instead of growing a tree rooted at a vertex it grows a *forest* i.e. a graph in which each component is a tree.

Initial Step Let F_1 be the spanning forest of N with $V(F_1) = V(N)$ and $E(F_1) = \emptyset$.

Recursive Step Suppose we have constructed a spanning forest F_i of N for some $i \geq 1$.

- If $i < |V(N)|$ then choose an edge $e \in E(N) - E(F_i)$ such that $F_i + e$ contains no cycles and, subject to this condition, $w(e)$ is as small as possible. Put $F_{i+1} = F_i + e$.

- If $i = |V(N)|$ then STOP. Put $T = F_i$ and output T.

We may show that Kruskal's algorithm is an efficient algorithm for constructing a minimum weight spanning tree of N in a similar way as we did for Prim's algorithm.

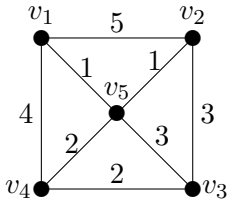
No efficient algorithm for finding a minimum weight strongly connected spanning subnetwork of a directed network is known. All known algorithms for doing this can take an exponential amount of time.

3.6 Finding shortest paths from a given vertex to every vertex in a network

In this section we will consider an undirected network N in which all edges have non-negative weights. We will describe an algorithm for finding shortest paths in N from a given vertex to every vertex of N.

3.6.1 Example

The following network N represents an existing rail network between cities. The weight of an edge represents the cost of a ticket to travel between the two cities represented by the end-vertices of the edge. A salesman located at x_1 wants to find the cheapest routes from x_1 to each city of N.



3.6.2 Definitions

Let P be path in a network N. The *length* of P, $w(P)$, is the sum of the weights of the edges of P. For $v \in V(N)$ and $S \subseteq V(N)$ the *distance* from v to S, $dist_N(v, S)$, is the minimum length of a path from v to a vertex of S.

Dijkstra's Algorithm

Let N be a network with positive edge weights and v be a vertex of N. Dijkstra (1959) gave an algorithm for constructing shortest paths from v to every vertex of N. His algorithm grows a tree rooted at v in a similar way to Prim's algorithm. As for Prim's algorithm, it is easiest to describe the algorithm for simple networks. If N is not simple then we delete all loops in N and replace each multiple edge by a single edge with weight equal to the smallest weight in the multiple edge. This preliminary step will not change the length of any shortest path in N and can be performed in time $O(|E(N)|)$. Henceforth we assume N is simple. This means we can uniquely represent each edge by its pair of end vertices. We write uv to mean the edge of N with end vertices u and v and use $w(uv)$ to denote the weight of this edge.

In the i'th step of the iteration, we have a tree T_i with $V(T_i) = \{x_1, x_2, \ldots, x_i\}$ with $x_1 = v$. We grow the tree by choosing an edge $xy \in E(N)$ from T_i to $N - T_i$ such that $dist_{T_i}(v, x) + w(xy)$ is as small as possible, and then putting $T_{i+1} = T_i + y + xy$. As for Prim's algorithm, we use a labelling procedure to ensure that no edge of N is considered more than once. Each vertex $y \in V(N) - V(T_i)$ is labelled with an ordered pair $label_i(y) = [x, h_i(y)]$ where $h_i(y)$ is the length of a shortest path from v to y *which uses only vertices of* $V(T_i) \cup \{y\}$, and $x \in V(T_i)$ is the vertex which precedes y on such a path. (If there is no path from v to y which uses only the vertices of $V(T_i) \cup \{y\}$ then we have $label_i(y) = [x_1, \infty]$.)

Initial Step Put $x_1 = v$ and let T_1 be the tree with $V(T_1) = \{x_1\}$ and $E(T_1) = \emptyset$. For $y \in V(N) - V(T_1)$ put

$$label_1(y) = \begin{cases} [x_1, w(x_1y)] & \text{if } y \text{ is adjacent to } x_1, \\ [x_1, \infty] & \text{otherwise.} \end{cases}$$

Iterative Step Suppose we have constructed a tree T_i with $V(T_i) = \{x_1, x_2, \ldots, x_i\}$ for some $i \geq 1$ and have labelled all vertices $y \in V(N) - V(T_i)$ with a label $label_i(y) = [x, h_i(y)]$.

- If $V(T_i) \neq V(N)$ then choose a vertex $\hat{y} \in V(N) - V(T_i)$ such that $label_i(\hat{y}) = [\hat{x}, h_i(\hat{y})]$ and $h_i(\hat{y})$ is as small as possible. Put $x_{i+1} = \hat{y}$ and $T_{i+1} := T_i + x_{i+1} + \hat{x}x_{i+1}$. For each vertex $y \in$

$V(N) - V(T_{i+1})$ put

$$label_{i+1}(y) = \begin{cases} [x_{i+1}, h_i(x_{i+1}) + w(x_{i+1}y)] & \text{if } y \text{ is adjacent to } x_{i+1} \text{ and} \\ & h_i(x_{i+1}) + w(x_{i+1}y) < h_i(y), \\ label_i(y) & \text{otherwise.} \end{cases}$$

- If $V(T_i) = V(N)$ then STOP. Put $T = T_i$ and output T.

Our next lemma shows that the tree constructed by Dijkstra's algorithm contains shortest paths from v to every vertex of N. Recall that the length of a shortest path in N from v to a set S of vertices of N is denoted by $dist_N(v, S)$.

3.6.3 Lemma

Let N be a network with positive weights on its edges and v be a vertex of N. Let T_i be a tree rooted at v produced in the i'th iteration of Dijkstra's algorithm applied to N. Then $dist_{T_i}(v, x) = dist_N(v, x)$ for each vertex x of T_i, and hence the unique path in T_i from v to x in T_i is a shortest path in N from v to x.

Proof We use induction on i.
Base Case $i = 1$. Since T_1 is a tree with one vertex $x_1 = v$ and no edges, we have $dist_{T_1}(v, x_1) = 0 = dist_N(v, x_1)$.
Induction Hypothesis Suppose $i \geq 2$ and that $dist_{T_{i-1}}(v, x) = dist_N(v, x)$ for each vertex x of T_{i-1}.
Inductive Step Let $S = V(N) - V(T_{i-1})$. We have $T_i = T_{i-1} + \hat{y} + \hat{x}\hat{y}$ where $\hat{x} \in V(T_{i-1}$, $\hat{y} \in S$, $\hat{x}\hat{y} \in E(N)$, and $\hat{x}, \hat{y}$ are chosen such that $dist_{T_{i-1}}(v, \hat{x}) + w(\hat{x}\hat{y})$ is as small as possible. Since $dist_{T_i}(v, x) = dist_{T_{i-1}}(v, x) = dist_N(v, x)$ for all $x \in V(T_{i-1})$, it suffices to show that $dist_{T_i}(v, \hat{y}) = dist_N(v, \hat{y})$. We have

$$dist_N(v, S) \leq dist_N(v, \hat{y}) \leq dist_{T_{i-1}}(v, \hat{x}) + w(\hat{x}\hat{y}), \tag{1}$$

since there is a path in N from v to $\hat{y}$ of length $dist_{T_{i-1}}(v, \hat{x}) + w(\hat{x}\hat{y})$. On the other hand, if $P = vu_1u_2 \ldots u_r y$ is a shortest path in N from v to S, then, since all edge weights are positive, we must have $u_r \in V(T_{i-1})$ and $P'vu_1u_2 \ldots u_r$ is a shortest path in N from v to u_r. Thus

$$\begin{aligned} dist_N(v, S) = w(P) = w(P') + w(u_r y) &= dist_N(v, u_r) + w(u_r y) \\ &= dist_{T_{i-1}}(v, u_r) + w(u_r y), \end{aligned} \tag{2}$$

again by the induction hypothesis. In the i'th step in Dijkstra's algorithm we chose the vertices $\hat{x}, \hat{y}$ so that $dist_{T_{i-1}}(v, \hat{x}) + w(\hat{x}\hat{y})$ was as small as possible. Thus

$$dist_{T_{i-1}}(v, \hat{x}) + w(\hat{x}\hat{y}) \leq dist_{T_{i-1}}(v, u_r) + w(u_r y). \tag{3}$$

Combining (1), (2) and (3), we obtain

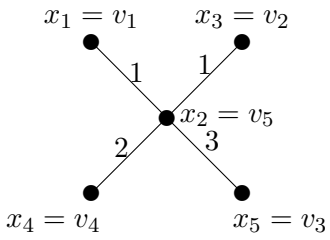
$$\begin{aligned} dist_N(v, S) \leq dist_N(v, \hat{y}) \leq dist_{T_{i-1}}(v, \hat{x}) + w(\hat{x}\hat{y}) &\leq dist_{T_{i-1}}(v, u_r) + w(u_r y) \\ &= dist_N(v, S). \end{aligned}$$

Hence equality must hold throughout, and in particular

$$dist_N(v, \hat{y}) = dist_{T_{i-1}}(v, \hat{x}) + w(\hat{x}\hat{y}) = dist_{T_i}(v, \hat{y}).$$

We say that a spanning tree in a network N which contains shortest paths from a vertex v to all vertices in the network is a *shortest path spanning tree of N rooted at v*. We can show in a similar way as for Prim's algorithm that the time Dijkstra's algorithm takes to construct a shortest path spanning tree of N rooted at v is $O(|V(N)|^2 + |E(N)|)$.

Applying the algorithm to the network of Example 3.6.1, we obtain the following shortest path spanning tree T_5 rooted at v_1.



The corresponding table of vertex labels is shown below.

	v_1	v_2	v_3	v_4	v_5
1'st Iteration	x_1	$[x_1, 5]$	$[x_1, \infty]$	$[x_1, 4]$	$[x_1, 1]$
2'nd Iteration	x_1	$[x_2, 2]$	$[x_2, 4]$	$[x_2, 3]$	x_2
3'rd Iteration	x_1	x_3	$[x_2, 4]$	$[x_2, 3]$	x_2
4'th Iteration	x_1	x_3	$[x_2, 4]$	x_4	x_2
5'th Iteration	x_1	x_3	x_5	x_4	x_2

Thus a shortest path in N from v_1 to v_2 is $v_1v_5v_2$ and a shortest in N from v_1 to v_4 is $v_1v_5v_4$.

Dijkstra's algorithm can easily be modified to find shortest directed paths from a given vertex to every vertex in a directed network which has positive weights on its arcs.

3.7 Finding longest paths (and shortest paths) from a given vertex to every vertex in an acyclic directed network

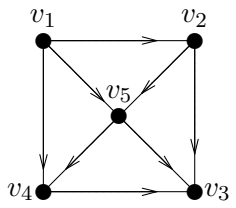
No efficient algorithm for finding longest paths in networks is known. All known algorithms for doing this can take an exponential amount of time. We can, however, find longest paths efficiently in directed networks which have no directed cycles.

3.7.1 Definitions

A digraph D is *acyclic* if it has no directed cycles. An *acyclic vertex labelling* of D is a labelling of its vertices as $x_1, x_2, \ldots, x_n$ such that for all arcs $e = x_ix_j$ of N, we have $i < j$.

3.7.2 Example

The following digraph D is acyclic and $x_1 = v_1$, $x_2 = v_2$, $x_3 = v_5$, $x_4 = v_4$, $x_5 = v_3$ is an acyclic vertex labelling of its vertices.



We shall show that a digraph is acyclic if and only if it has an acyclic labelling. We need the following lemma.

3.7.3 Lemma

Let D be an acyclic digraph. Then D contains a vertex v with $d_D^-(v) = 0$.

Proof

Let $P = v_0e_1v_1 \dots e_mv_m$ be a directed path of maximum length in D. There are no arcs in D from any vertex of $V(D) - V(P)$ to v_0 since otherwise we could extend P. There are no arcs in D from any vertex of $V(P)$ to v_0 since otherwise we would obtain a directed cycle in D. Thus $d^-_D(v_0) = 0$.

3.7.4 Lemma

Let D be a digraph. Then D is acyclic if and only if D has an acyclic vertex labelling.

Proof
Sufficiency Suppose D has an acyclic vertex labelling $x_1, x_2, \dots, x_n$. Draw D with all its vertices on a horizontal line with x_1 first, then x_2, and so on. Then every arc of D goes from left to right. Thus D can have no directed cycles since every directed walk with at least one arc must move continuously towards the right so cannot return to its first vertex.
Necessity Suppose D is acyclic. By Lemma 3.7.2, D has a vertex v with $d^-_D(v) = 0$. Let $x_1 = v$ and let $D_1 = D - x_1$ be the digraph obtained by deleting x_1 and all arcs incident to x_1 from D. Then D_1 is acyclic so again by Lemma 3.7.2, D_1 has a vertex w with $d^-_{D_1}(w) = 0$. Let $x_2 = w$ and let $D_2 = D_1 - x_2$. Continuing in this way we eventually obtain an acyclic vertex labelling $x_1, x_2, x_3, \dots, x_n$ for D.

Note that the 'Necessity part' of the above proof is constructive and readily gives rise to an efficient algorithm for finding an acyclic vertex labelling of an acyclic digraph. We can use acyclic vertex labellings to find longest paths in acyclic directed networks.

Morávek's Algorithm

Let N be an acyclic directed network and $x_1, x_2, \dots, x_n$ be an acyclic vertex labelling of N. Suppose that every vertex of N can be reached by a directed path starting at x_1. Morávek (1970) gave an algorithm for constructing longest paths from x_1 to every vertex of N. His algorithm grows an out arborescence rooted at x_1. It is easiest to describe the algorithm for acyclic directed networks without multiple arcs. If N contains multiple arcs then we replace each multiple arc by a single arc with weight equal to the largest weight in the multiple arc. This preliminary step will not change the length of any longest path in N and can be performed in time $O(|A(N)|)$. Henceforth we assume N has no multiple arcs. This means we can uniquely represent each arc by its ordered pair of end vertices. We write uv to mean

the arc of N with tail u and head v, and use $w(uv)$ to denote the weight of this arc.

In the i'th step of the iteration, we have an out-arborescence T_i rooted at x_1 with $V(T_i) = \{x_1, x_2, \ldots, x_i\}$. Thus the vertices of T_i are the first i vertices in the acyclic labelling of N. We grow the out-arborescence by choosing an arc $x_j x_{i+1} \in A(N)$ from T_i to x_{i+1} such that $dist_{T_i}(x_1, x_j) + w(x_j x_{i+1})$ is as large as possible, and then putting $T_{i+1} = T_i + x_{i+1} + x_j x_{i+1}$.

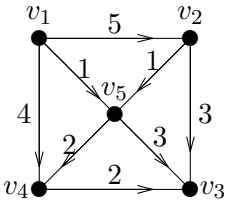
Initial Step Put $x_1 = v$ and let T_1 be the out-arborescence with $V(T_1) = \{x_1\}$ and $E(T_1) = \emptyset$.

Iterative Step Suppose we have constructed an out-arborescence T_i rooted at x_1 with $V(T_i) = \{x_1, x_2, \ldots, x_i\}$ for some $i \geq 1$.

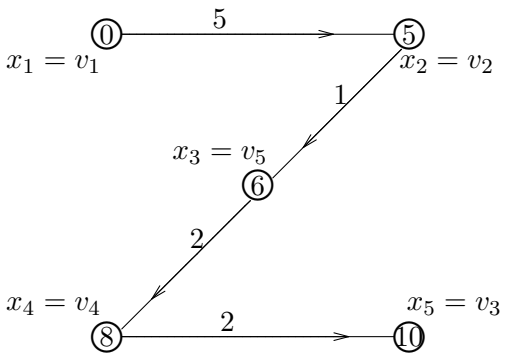
- If $V(T_i) \neq V(N)$ then choose a vertex $x_j \in V(T_i)$ such that $x_j x_{i+1} \in A(N)$ and $dist_{T_i}(x_1, x_j) + w(x_j x_{i+1})$ is as large as possible. Put $T_{i+1} = T_i + x_{i+1} + x_j x_{i+1}$.
- If $V(T_i) = V(N)$ then STOP. Put $T = T_i$ and output T.

3.7.5 Example

Consider the following acyclic directed network.



Applying Moravék's algorithm using the acyclic labelling from Example 3.7.2, we obtain the longest path out-arborescence rooted at v_1 shown below.



Our next lemma shows that the tree constructed by Morávek's algorithm contains longest directed paths from v to every vertex of N.

3.7.6 Lemma

Let N be an acyclic directed network and $x_1, x_2, \ldots, x_n$ be an acyclic vertex labelling of N. Suppose that every vertex of N can be reached by a directed path starting at x_1. Let T_i be an out-arborescence rooted at x_1 produced in the i'th iteration of Morávek's algorithm applied to N. Then the unique directed path in T_i from x_1 to x in T_i is a longest directed path in N from x_1 to x, for all vertices x of T_i.

Proof Let $long_N(x_1, x_i)$ denote the length of a longest directed path in N from x_1 to x_i, for all $1 \leq i \leq n$. We need to show that $dist_{T_i}(x_1, x) = long_N(x_1, x)$ for all $x \in V(T_i)$. We use induction on i.

Base Case $i = 1$. Since T_1 is an out arborescence with one vertex $x_1 = v$ and no edges, we have $dist_{T_1}(v, x_1) = 0 = long_N(v, x_1)$.

Induction Hypothesis Suppose $i \geq 2$ and that $dist_{T_{i-1}}(x_1, x) = long_N(x_1, x)$ for each vertex x of T_{i-1}.

Inductive Step We have $T_i = T_{i-1} + x_i + x_j x_i$ where $x_j \in V(T_{i-1})$, $x_j x_i \in A(N)$, and x_j is chosen such that $dist_{T_{i-1}}(v, x_j) + w(x_j x_i)$ is as large as possible. Since $dist_{T_i}(x_1, x) = dist_{T_{i-1}}(x_1, x) = long_N(x_1, x)$ for all $x \in V(T_{i-1})$, it suffices to show that $dist_{T_i}(x_1, x_i) = long_N(x_1, x_i)$. We have

$$long_N(x_1, x_i) \geq dist_{T_{i-1}}(x_1 x_j) + w(x_j x_i), \tag{4}$$

since there is a directed path in N from x_1 to x_i of length $dist_{T_{i-1}}(x_1, x_j) + w(x_j x_i)$. On the other hand, if $P = x_1 u_1 u_2 \ldots u_r x_i$ is a longest directed path in N from x_1 to x_i, then, since $u_r x_i \in A(N)$ and $x_1, x_2, \ldots, x_n$ is an acyclic labelling of N, we must have $u_r = x_k$ for some $1 \leq k \leq i - 1$. Let

$P' = x_1u_1u_2 \dots u_r$. Then

$$\begin{aligned} long_N(x_1, x_i) = w(P) = w(P') + w(x_kx_i) &\leq long_N(x_1, x_k) + w(x_kx_i) \\ &= dist_{T_{i-1}}(x_1, x_k) + w(x_kx_i), \end{aligned} \tag{5}$$

by the induction hypothesis, since $x_k \in V(T_{i-1})$. In the i'th step in Morávek's algorithm we chose the vertex x_j so that $dist_{T_{i-1}}(x_1, x_j) + w(x_jx_i)$ was as large as possible. Thus

$$dist_{T_{i-1}}(x_1, x_j) + w(x_jx_i) \geq dist_{T_{i-1}}(x_1, x_k) + w(x_kx_i). \tag{6}$$

Combining (4), (5) and (6), we obtain

$$\begin{aligned} long_N(x_1, x_i) \geq dist_{T_{i-1}}(x_1x_j) + w(x_jx_i) &\geq dist_{T_{i-1}}(x_1, x_k) + w(x_kx_i) \\ &\geq long_N(x_1, x_i). \end{aligned}$$

Hence equality must hold throughout, and in particular

$$long_N(x_1, x_i) = dist_{T_{i-1}}(x_1x_j) + w(x_jx_i) = dist_{T_i}(x_1x_i).$$

In the i'th iteration of Morávek's algorithm, we consider each arc entering x_i and choose the arc x_jx_i entering x_i for which $dist_{T_{i-1}}(x_1x_j) + w(x_jx_i)$ is as large as possible. Thus the time taken by the i'th iteration is $O(d_N^-(x_i))$. Hence the time taken for the algorithm to construct a spanning out arborescence is $O(\sum_{i=1}^n d_N^-(x_i)) = O(|A(N)|)$. We also have to consider the time taken to construct the acyclic labelling of N. It can be shown that this is also $O(|A(N)|)$, and hence the total running time of the algorithm is $O(|A(N)|)$. It is straightforward to adapt the algorithm to find *shortest* directed paths from x_1 to every vertex of N. The total running time will again be $O(|A(N)|)$.