11 Shortest Paths and Minimum Spanning Trees

Consider a network (V, E) with associated costs c_{ij} for each edge $(i, j) \in E$, corresponding for example to the physical distance between vertices i and j or the cost of establishing a link between them. The *single-pair shortest path problem* then asks for a (directed) path from a given source $s \in V$ to a given destination $t \in V$ that has minimum cost, where the cost of a path is the sum of costs of its edges. The shortest path problem has numerous applications in transportation and communications, and also occurs frequently as a subproblem of more complex problems. It is a special case of the minimum cost flow problem, but can be solved more efficiently using specialized algorithms.

11.1 The Bellman Equations

It will be instructive to consider a destination $t \in V$ and simultaneously look for shortest paths from any vertex $i \in V \setminus \{t\}$ to t. This problems is sometimes called the *single-destination* shortest path problem, and is equivalent to the minimum cost flow problem on the same network where one unit of flow is to be routed from each vertex $i \in V \setminus \{t\}$ to t, i.e., the one with supply $b_i = 1$ at every vertex $i \in V \setminus \{t\}$ and demand $b_t = -(|V| - 1)$ at vertex t.

Let λ_i for $i \in V$ be the dual solution corresponding to an optimal spanning tree solution of this flow problem, and recall that for every edge $(i, j) \in E$ with $x_{ij} > 0$,

$$\lambda_i = c_{ij} + \lambda_j.$$

By setting $\lambda_t = 0$ and adding these equalities along a path from i to t, we see that λ_i is equal to the length of a shortest path from i to t. Moreover, since $b_i = 1$ for all $i \in V \setminus \{t\}$, and given $\lambda_t = 0$, the dual problem is to

$$\text{maximize} \sum_{i \in V \setminus \{t\}} \lambda_i \quad \text{subject to } \lambda_i \leqslant c_{ij} + \lambda_j \text{ for all } (i, j) \in E.$$

In an optimal solution, λ_i will thus be as large as possible subject to the constraints, i.e., it will satisfy the so-called *Bellman equations*

$$\lambda_i = \min_{j:(i,j) \in E} (c_{ij} + \lambda_j) \quad \text{for all } i \in V \setminus \{t\},$$

with $\lambda_t = 0$. The intuition behind these equalities is that in order to find a shortest path from i to t, one should choose the first edge (i, j) on the path in order to minimize the sum of the length of this edge and that of a shortest path from j to t. This situation is illustrated in Figure 11.1.

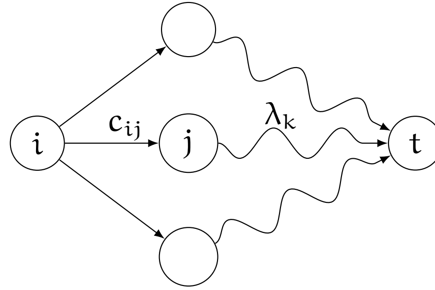

Figure 11.1: Illustration of the Bellman equations for the shortest path problem

11.2 The Bellman-Ford Algorithm

Let $\lambda_i(k)$ be the length of a shortest path from i to t that uses at most k edges. Then, $\lambda_t(k) = 0$ for all $k \geqslant 0$, and

$$\lambda_i(0) = \infty \quad \text{and}$$
$$\lambda_i(k) = \min_{j:(i,j)\in E} (c_{ij} + \lambda_j(k-1))$$

for all $i \in V \setminus \{t\}$ and $k \geqslant 1$.

The algorithm that successively computes $\lambda_i(k)$ for all i and larger and larger values of k is known as the *Bellman-Ford algorithm*. It is an example of a method called *dynamic programming*, which can be applied to problems that are decomposable into *overlapping subproblems* and have what is called *optimal substructure*, such that an overall solution can be constructed efficiently from solutions to the subproblems.

Note that $\lambda_i(|V|) < \lambda_i(|V|-1)$ for some $i \in V$ if and only if there exists a cycle of negative length, and that otherwise $\lambda_i = \lambda_i(|V|-1)$. In any case, $O(|V|)$ iterations of the Bellman-Ford algorithm suffice to determine λ_i. Each iteration requires $O(|E|)$ steps, for an overall running time of $O(|E| \cdot |V|)$. Given the values λ_i for all $i \in V$, a shortest path from i to t then leads along an edge $(i,j) \in E$ such that $\lambda_i = c_{ij} + \lambda_j$. Alternatively, one could store such a successor vertex for every vertex i while running the algorithm, and update it whenever $\lambda_i(k) < \lambda_i(k-1)$.

11.3 Dijkstra's Algorithm

The Bellman-Ford algorithm does not make any assumptions about edge lengths, and works in particular if some or all of them are negative. In the special case where all edges are known to have non-negative lengths, the running time can sometimes be decreased. The idea is to collect vertices in the order of increasing shortest path length to t. We assume from now on that $E = V \times V$, and set $c_{ij} = \infty$ if necessary. The following lemma will be useful.

LEMMA 11.1. *Consider a graph with vertices* V *and edge lengths* $c_{ij} \geqslant 0$ *for all* $i,j \in V$. *Fix* $t \in V$ *and let* λ_i *denote the length of a shortest path from* $i \in V$ *to* t. *Let* $j \in V \setminus \{t\}$ *such that* $c_{jt} = \min_{i\in V\setminus\{t\}} c_{it}$. *Then,* $\lambda_j = c_{jt}$ *and* $\lambda_j = \min_{i\in V\setminus\{t\}} \lambda_i$.

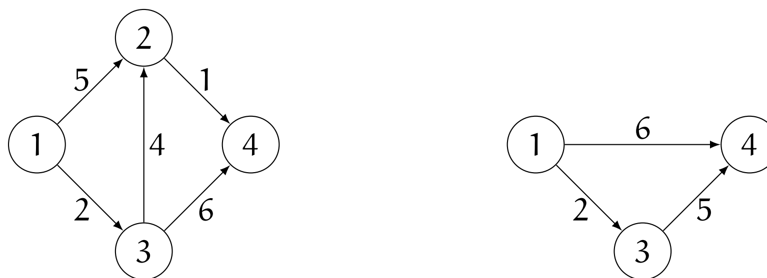

Figure 11.2: An iteration of Dijkstra's algorithm with $t = 4$. In the graph on the left, $c_{2t} = \min_{i \in V \setminus \{t\}} c_{it}$ and therefore, by Lemma 11.1, $\lambda_2 = c_{2t} = 1$. The graph on the right is then obtained by removing vertex 2 and updating c_{14} to $\min\{c_{14}, c_{12} + c_{24}\} = \min\{\infty, 5 + 1\} = 6$ and c_{34} to $\min\{c_{34}, c_{32} + c_{24}\} = \min\{6, 4 + 1\} = 5$.

Proof. Let $i \in V \setminus \{t\}$, consider a shortest path from i to t, and let (ℓ, t) be the last edge on this path. Then, $\lambda_i \geqslant \lambda_\ell \geqslant c_{\ell t} \geqslant c_{jt}$. This holds in particular for $i = j$, and on the other hand $\lambda_j \leqslant c_{jt}$. Thus $\lambda_j = c_{jt} \leqslant \lambda_i$. □

Dijkstra's algorithm uses this lemma to determine λ_j for a particular vertex j, removes j from the graph, and repeats the process for the new graph:

1. Find a vertex $j \in V \setminus \{t\}$ with $c_{jt} = \min_{i \in V \setminus \{t\}} c_{it}$. Set $\lambda_j = c_{jt}$.
2. For every vertex $i \in V \setminus \{j\}$, set $c_{it} = \min\{c_{it}, c_{ij} + c_{jt}\}$.
3. Remove vertex j from V. If $|V| > 1$, return to Step 1.

An example is shown in Figure 11.2.

The algorithm performs $|V| - 1$ iterations, each of which determines the new length of one edge for each of the remaining $O(|V|)$ vertices. The overall running time is thus $O(|V|^2)$. This improves on the Bellman-Ford algorithm in graphs with many edges, and is optimal in the sense that any algorithm for the single-destination shortest path problem has to inspect all of the edges, of which there are $\Omega(|V|^2)$ in the worst case.

One might wonder whether there exists a way to transform the edge lengths to make them non-negative without affecting the structure of the shortest paths, so that Dijkstra's algorithm could be used in the presence of negative lengths as well. Let λ_i be the length of a shortest path from vertex i to vertex t, and recall that $\lambda_i \leqslant c_{ij} + \lambda_j$ for all $(i, j) \in E$. Let $\bar{c}_{ij} = c_{ij} + \lambda_j - \lambda_i$. Then, $\bar{c}_{ij} \geqslant 0$ for every edge $(i, j) \in E$, and for an arbitrary path $v_1, v_2, \ldots, v_k$,

$$\sum_{i=1}^{k-1} \bar{c}_{v_i v_{i+1}} = \sum_{i=1}^{k-1} (c_{v_i v_{i+1}} + \lambda_{v_{i+1}} - \lambda_{v_i}) = \lambda_{v_k} - \lambda_{v_1} + \sum_{i=1}^{k-1} c_{v_i v_{i+1}}.$$

So indeed, changing edge lengths from c_{ij} to $\bar{c}_{ij}$ allows Dijkstra's algorithm to work correctly, and it does not affect the structure of the shortest paths.

This observation is not very useful in the context of single-pair or single-destination shortest path problems: we do not know the values λ_i, and computing them is at least as hard as the problem we are trying to solve. For the *all-pairs* shortest path problem,

however, which requires us to find a shortest path between every pair of vertices $i, j \in V$, the situation is different. The straightforward solution to this problem is to run the Bellman-Ford algorithm $|V|$ times, once for every possible destination vertex. In a graph with $\Omega(|V|^2)$ edges, this leads to an overall running time of $|V| \cdot O(|V|^3) = O(|V|^4)$. Using the above observation, we can instead invoke the Bellman-Ford algorithm for one destination vertex t to obtain the shortest path lengths λ_i for all $i \in V$, and compute shortest paths for the remaining destination vertices by running Dijkstra's algorithm on the graph with edge lengths $\bar{c}_{ij}$. This improves the asymptotic running time to $O(|V|^3) + |V - 1| \cdot O(|V|^2) = O(|V|^3)$.

11.4 Minimum Spanning Trees and Prim's Algorithm

The *minimum spanning tree problem* for a network (V, E) with associated costs c_{ij} for each edge $(i, j) \in E$ asks for a spanning tree of minimum cost, where the cost of a tree is the sum of costs of all its edges. This problem arises, for example, if one wishes to design a communication network that connects a given set of locations. The following property of minimum spanning trees will be useful.

THEOREM 11.2. *Let* (V, E) *be a graph with edge costs* c_{ij} *for all* $(i, j) \in E$. *Let* $U \subseteq V$ *and* $(u, v) \in U \times (V \setminus U)$ *such that* $c_{uv} = \min_{(i,j) \in U \times (V \setminus U)} c_{ij}$. *Then there exists a spanning tree of minimum cost that contains* (u, v).

Proof. Let $T \subseteq E$ be a spanning tree of minimum cost. If $(u, v) \in T$ we are done. Otherwise, $T \cup \{(u, v)\}$ contains a cycle, and there must be another edge $(u', v') \in T$ such that $(u', v') \in U \times (V \setminus U)$. Then, $(T \cup \{(u, v)\}) \setminus \{(u', v')\}$ is a spanning tree, and since (u, v) has minimum cost among the edges in $U \times (V \setminus U)$ its cost is no greater than that of T, and therefore minimum. □

Prim's algorithm uses this property to inductively construct a minimum spanning tree. It proceeds as follows:

1. Set $U = \{1\}$ and $T = \emptyset$.
2. If $U = V$, return T. Otherwise find an edge $(u, v) \in U \times (V \setminus U)$ such that $c_{uv} = \min_{(i,j) \in U \times (V \setminus U)} c_{ij}$.
3. Add v to U and (u, v) to T, and return to Step 2.

Prim's algorithm is called a *greedy algorithm*, because it always chooses an edge of minimum cost.

Suppose that after each iteration, we compute and store for every vertex $j \in V \setminus U$ a minimum cost edge to any vertex in U, i.e., an edge $(i, j) \in U \times (V \setminus U)$ such that $c_{ij} = \min_{(i',j) \in U \times (V \setminus U)} c_{i'j}$. This only requires a comparison between the previously stored edge and the edge to the vertex added to U in the current iteration, and can be done in time $O(|V|)$ per iteration. It then suffices in Step 2 to minimize cost among vertices in $V \setminus U$, of which there are $O(|V|)$. Since the algorithm performs $|V| - 1$ iterations, it thus has an overall running time of $O(|V|^2)$.