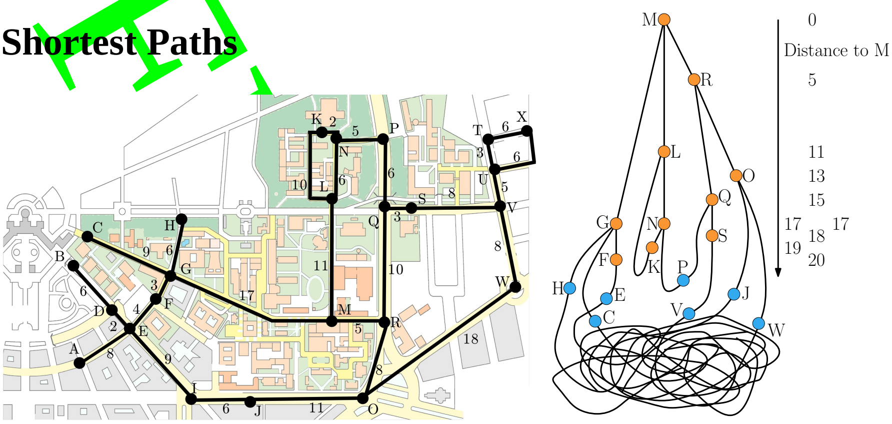
10 Shortest Paths



The problem of the shortest, quickest or cheapest path is ubiquitous. You solve it daily. When you are in a location s and want to move to a location t, you ask for the quickest path from s to t. The fire department may want to compute the quickest routes from a fire station s to all locations in town – the single-source problem. Sometimes we may even want a complete distance table from everywhere to everywhere – the all-pairs problem. In a road atlas, you will usually find an all-pairs distance table for the most important cities.

Here is a route-planning algorithm that requires a city map and a lot of dexterity but no computer. Lay thin threads along the roads on the city map. Make a knot wherever roads meet, and at your starting position. Now lift the starting knot until the entire net dangles below it. If you have successfully avoided any tangles and the threads and your knots are thin enough so that only tight threads hinder a knot from moving down, the tight threads define the shortest paths. The introductory figure of this chapter shows the campus map of the University of Karlsruhe[1] *and illustrates the route-planning algorithm for the source node M.*

Route planning in road networks is one of the many applications of shortest-path computations. When an appropriate graph model is defined, many problems turn out to profit from shortest-path computations. For example, Ahuja et al. [8] mentioned such diverse applications as planning flows in networks, urban housing, inventory planning, DNA sequencing, the knapsack problem (see also Chap. 12), production planning, telephone operator scheduling, vehicle fleet planning, approximating piecewise linear functions, and allocating inspection effort on a production line.

The most general formulation of the shortest-path problem looks at a directed graph $G = (V, E)$ and a cost function c that maps edges to arbitrary real-number

[1] (c) Universität Karlsruhe (TH), Institut für Photogrammetrie und Fernerkundung.

costs. It turns out that the most general problem is fairly expensive to solve. So we are also interested in various restrictions that allow simpler and more efficient algorithms: nonnegative edge costs, integer edge costs, and acyclic graphs. Note that we have already solved the very special case of unit edge costs in Sect. 9.1 – the breadth-first search (BFS) tree rooted at node s is a concise representation of all shortest paths from s. We begin in Sect. 10.1 with some basic concepts that lead to a generic approach to shortest-path algorithms. A systematic approach will help us to keep track of the zoo of shortest-path algorithms. As our first example of a restricted but fast and simple algorithm, we look at acyclic graphs in Sect. 10.2. In Sect. 10.3, we come to the most widely used algorithm for shortest paths: Dijkstra's algorithm for general graphs with nonnegative edge costs. The efficiency of Dijkstra's algorithm relies heavily on efficient priority queues. In an introductory course or at first reading, Dijkstra's algorithm might be a good place to stop. But there are many more interesting things about shortest paths in the remainder of the chapter. We begin with an average-case analysis of Dijkstra's algorithm in Sect. 10.4 which indicates that priority queue operations might dominate the execution time less than one might think. In Sect. 10.5, we discuss *monotone priority queues for integer keys* that take additional advantage of the properties of Dijkstra's algorithm. Combining this with average-case analysis leads even to a linear expected execution time. Section 10.6 deals with arbitrary edge costs, and Sect. 10.7 treats the all-pairs problem. We show that the all-pairs problem for general edge costs reduces to one general single-source problem plus n single-source problems with nonnegative edge costs. This reduction introduces the generally useful concept of node potentials. We close with a discussion of shortest path queries in Sect. 10.8.

10.1 From Basic Concepts to a Generic Algorithm

We extend the cost function to paths in the natural way. The cost of a path is the sum of the costs of its constituent edges, i.e., if $p = \langle e_1, e_2, \ldots, e_k \rangle$, then $c(p) = \sum_{1 \leq i \leq k} c(e_i)$. The empty path has cost zero.

For a pair s and v of nodes, we are interested in a shortest path from s to v. We avoid the use of the definite article "the" here, since there may be more than one shortest path. Does a shortest path always exist? Observe that the number of paths from s to v may be infinite. For example, if $r = pCq$ is a path from s to v containing a cycle C, then we may go around the cycle an arbitrary number of times and still have a path from s to v; see Fig. 10.1. More precisely, p is a path leading from s to u, C is a path leading from u to u, and q is a path from u to v. Consider the path $r^{(i)} = pC^iq$ which first uses p to go from s to u, then goes around the cycle i times, and finally follows q from u to v. The cost of $r^{(i)}$ is $c(p) + i \cdot c(C) + c(q)$. If C is a *negative cycle*, i.e., $c(C) < 0$, then $c(r^{(i+1)}) < c(r^{(i)})$. In this situation, there is no shortest path from s to v. Assume otherwise: say, P is a shortest path from s to v. Then $c(r^{(i)}) < c(P)$ for i large enough[2], and so P is not a shortest path from s to v. We shall show next that shortest paths exist if there are no negative cycles.

[2] $i > (c(p) + c(q) - c(P))/|c(C)|$ will do.

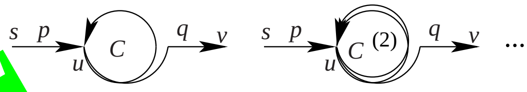


Fig. 10.1. A nonsimple path pCq from s to v

Lemma 10.1. *If G contains no negative cycles and v is reachable from s, then a shortest path P from s to v exists. Moreover P can be chosen to be simple.*

Proof. Let x be a shortest *simple* path from s to v. If x is not a shortest path from s to v, there is a shorter nonsimple path r from s to v. Since r is nonsimple we can, as in Fig. 10.1, write r as pCq, where C is a cycle and pq is a simple path. Then $c(x) \leq c(pq)$, and hence $c(pq) + c(C) = c(r) < c(x) \leq c(pq)$. So $c(C) < 0$ and we have shown the existence of a negative cycle. $\square$

Exercise 10.1. Strengthen the lemma above and show that if v is reachable from s, then a shortest path from s to v exists iff there is no negative cycle that is reachable from s and from which one can reach v.

For two nodes s and v, we define the shortest-path distance $\mu(s,v)$ from s to v as

$$\mu(s,v) := \begin{cases} +\infty & \text{if there is no path from } s \text{ to } v, \\ -\infty & \text{if there is no shortest path from } s \text{ to } v, \\ c\,(\text{a shortest path from } s \text{ to } v) & \text{otherwise.} \end{cases}$$

Since we use s to denote the source vertex most of the time, we also use the shorthand $\mu(v) := \mu(s,v)$. Observe that if v is reachable from s but there is no shortest path from s to v, then there are paths of arbitrarily large negative cost. Thus it makes sense to define $\mu(v) = -\infty$ in this case. Shortest paths have further nice properties, which we state as exercises.

Exercise 10.2 (subpaths of shortest paths). Show that subpaths of shortest paths are themselves shortest paths, i.e., if a path of the form pqr is a shortest path, then q is also a shortest path.

Exercise 10.3 (shortest-path trees). Assume that all nodes are reachable from s and that there are no negative cycles. Show that there is an n-node tree T rooted at s such that all tree paths are shortest paths. Hint: assume first that shortest paths are unique, and consider the subgraph T consisting of all shortest paths starting at s. Use the preceding exercise to prove that T is a tree. Extend this result to the case where shortest paths are not unique.

Our strategy for finding shortest paths from a source node s is a generalization of the BFS algorithm shown in Fig. 9.3. We maintain two *NodeArray*s d and *parent*. Here, $d[v]$ contains our current knowledge about the distance from s to v, and *parent*$[v]$ stores the predecessor of v on the current shortest path to v. We usually

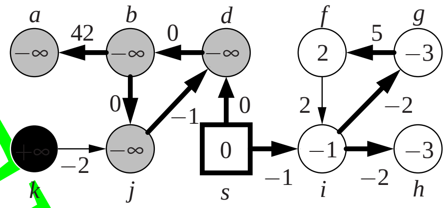


Fig. 10.2. A graph with shortest-path distances $\mu(v)$. Edge costs are shown as edge labels, and the distances are shown inside the nodes. The thick edges indicate shortest paths

refer to $d[v]$ as the *tentative distance* of v. Initially, $d[s] = 0$ and $parent[s] = s$. All other nodes have infinite distance and no parent.

The natural way to improve distance values is to propagate distance information across edges. If there is a path from s to u of cost $d[u]$, and $e = (u,v)$ is an edge out of u, then there is a path from s to v of cost $d[u] + c(e)$. If this cost is smaller than the best previously known distance $d[v]$, we update d and *parent* accordingly. This process is called *edge relaxation*:

Procedure $relax(e = (u,v) : Edge)$
  if $d[u] + c(e) < d[v]$ **then** $d[v] := d[u] + c(e)$; $parent[v] := u$

Lemma 10.2. *After any sequence of edge relaxations, if $d[v] < \infty$, then there is a path of length $d[v]$ from s to v.*

Proof. We use induction on the number of edge relaxations. The claim is certainly true before the first relaxation. The empty path is a path of length zero from s to s, and all other nodes have infinite distance. Consider next a relaxation of an edge $e = (u,v)$. By the induction hypothesis, there is a path p of length $d[u]$ from s to u and a path of length $d[v]$ from s to v. If $d[u] + c(e) \geq d[v]$, there is nothing to show. Otherwise, pe is a path of length $d[u] + c(e)$ from s to v. □

The common strategy of the algorithms in this chapter is to relax edges until either all shortest paths have been found or a negative cycle is discovered. For example, the (reversed) thick edges in Fig. 10.2 give us the *parent* information obtained after a sufficient number of edge relaxations: nodes f, g, i, and h are reachable from s using these edges and have reached their respective $\mu(\cdot)$ values 2, −3, −1, and −3. Nodes b, j, and d form a negative-cost cycle so that their shortest-path cost is $-\infty$. Node a is attached to this cycle, and thus $\mu(a) = -\infty$.

What is a good sequence of edge relaxations? Let $p = \langle e_1, \ldots, e_k \rangle$ be a path from s to v. If we relax the edges in the order e_1 to e_k, we have $d[v] \leq c(p)$ after the sequence of relaxations. If p is a shortest path from s to v, then $d[v]$ cannot drop below $c(p)$, by the preceding lemma, and hence $d[v] = c(p)$ after the sequence of relaxations.

Lemma 10.3 (correctness criterion). *After performing a sequence R of edge relaxations, we have $d[v] = \mu(v)$ if, for some shortest path $p = \langle e_1, e_2, \ldots, e_k \rangle$ from* *s to v, p is a subsequence of R, i.e., there are indices $t_1 < t_2 < \cdots < t_k$ such that $R[t_1] = e_1, R[t_2] = e_2, \ldots, R[t_k] = e_k$. Moreover, the parent information defines a path of length $\mu(v)$ from s to v.*

Proof. The following is a schematic view of R and p: the first row indicates the time. At time t_1, the edge e_1 is relaxed, at time t_2, the edge e_2 is relaxed, and so on:

$$\begin{array}{lllllll} & 1,2,\ldots, & t_1, & \ldots, & t_2, & \ldots\ldots & ,t_k, \quad \ldots \\ R := \langle & \ldots & ,e_1, & \ldots, & e_2, & \ldots\ldots & ,e_k, \quad \ldots\rangle \\ p := & & \langle e_1, & & e_2, & \ldots & ,e_k\rangle \end{array}$$

We have $\mu(v) = \sum_{1 \le j \le k} c(e_j)$. For $i \in 1..k$, let v_i be the target node of e_i, and we define $t_0 = 0$ and $v_0 = s$. Then $d[v_i] \le \sum_{1 \le j \le i} c(e_j)$ after time t_i, as a simple induction shows. This is clear for $i = 0$, since $d[s]$ is initialized to zero and d-values are only decreased. After the relaxation of $e_i = R[t_i]$ for $i > 0$, we have $d[v_i] \le d[v_{i-1}] + c(e_i) \le \sum_{1 \le j \le i} c(e_j)$. Thus, after time t_k, we have $d[v] \le \mu(v)$. Since $d[v]$ cannot go below $\mu(v)$, by Lemma 10.2, we have $d[v] = \mu(v)$ after time t_k and hence after performing all relaxations in R.

Let us prove next that the *parent* information traces out shortest paths. We shall do so under the additional assumption that shortest paths are unique, and leave the general case to the reader. After the relaxations in R, we have $d[v_i] = \mu(v_i)$ for $1 \le i \le k$. When $d[v_i]$ was set to $\mu(v_i)$ by an operation $relax(u, v_i)$, the existence of a path of length $\mu(v_i)$ from s to v_i was established. Since, by assumption, the shortest path from s to v_i is unique, we must have $u = v_{i-1}$, and hence $parent[v_i] = v_{i-1}$. □

Exercise 10.4. Redo the second paragraph in the proof above, but without the assumption that shortest paths are unique.

Exercise 10.5. Let S be the edges of G in some arbitrary order and let $S^{(n-1)}$ be $n-1$ copies of S. Show that $\mu(v) = d[v]$ for all nodes v with $\mu(v) \ne -\infty$ after the relaxations $S^{(n-1)}$ have been performed.

In the following sections, we shall exhibit more efficient sequences of relaxations for acyclic graphs and for graphs with nonnegative edge weights. We come back to general graphs in Sect. 10.6.

10.2 Directed Acyclic Graphs

In a directed acyclic graph (DAG), there are no directed cycles and hence no negative cycles. Moreover, we have learned in Sect. 9.2.1 that the nodes of a DAG can be topologically sorted into a sequence $\langle v_1, v_2, \ldots, v_n\rangle$ such that $(v_i, v_j) \in E$ implies $i < j$. A topological order can be computed in linear time $O(m+n)$ using either depth-first search or breadth-first search. The nodes on any path in a DAG increase in topological order. Thus, by Lemma 10.3, we can compute correct shortest-path distances if we first relax the edges out of v_1, then the edges out of v_2, etc.; see Fig. 10.3 for an example. In this way, each edge is relaxed only once. Since every edge relaxation takes constant time, we obtain a total execution time of $O(m+n)$.

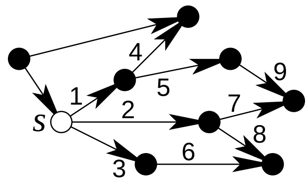


Fig. 10.3. Order of edge relaxations for the computation of the shortest paths from node s in a DAG. The topological order of the nodes is given by their x-coordinates

Theorem 10.4. *Shortest paths in acyclic graphs can be computed in time* $\mathrm{O}(m+n)$.

Exercise 10.6 (route planning for public transportation). Finding the quickest routes in public transportation systems can be modeled as a shortest-path problem for an acyclic graph. Consider a bus or train leaving a place p at time t and reaching its next stop p' at time t'. This connection is viewed as an edge connecting nodes (p,t) and (p',t'). Also, for each stop p and subsequent events (arrival and/or departure) at p, say at times t and t' with $t < t'$, we have the *waiting link* from (p,t) to (p,t'). (a) Show that the graph obtained in this way is a DAG. (b) You need an additional node that models your starting point in space and time. There should also be one edge connecting it to the transportation network. What should this edge be? (c) Suppose you have computed the shortest-path tree from your starting node to all nodes in the public transportation graph reachable from it. How do you actually find the route you are interested in?

10.3 Nonnegative Edge Costs (Dijkstra's Algorithm)

We now assume that all edge costs are nonnegative. Thus there are no negative cycles, and shortest paths exist for all nodes reachable from s. We shall show that if the edges are relaxed in a judicious order, every edge needs to be relaxed only once.

What is the right order? Along any shortest path, the shortest-path distances increase (more precisely, do not decrease). This suggests that we should scan nodes (to scan a node means to relax all edges out of the node) in order of increasing shortest-path distance. Lemma 10.3 tells us that this relaxation order ensures the computation of shortest paths. Of course, in the algorithm, we do not know the shortest-path distances; we only know the *tentative distances* $d[v]$. Fortunately, for an unscanned node with minimal tentative distance, the true and tentative distances agree. We shall prove this in Theorem 10.5. We obtain the algorithm shown in Fig. 10.4. This algorithm is known as Dijkstra's shortest-path algorithm. Figure 10.5 shows an example run.

Note that Dijkstra's algorithm is basically the thread-and-knot algorithm we saw in the introduction to this chapter. Suppose we put all threads and knots on a table and then lift the starting node. The other knots will leave the surface of the table in the order of their shortest-path distances.

Theorem 10.5. *Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.*

Dijkstra's Algorithm
declare all nodes unscanned and initialize d and *parent*
while there is an unscanned node with tentative distance $< +\infty$ **do**

  u:= the unscanned node with minimal tentative distance
  relax all edges (u,v) out of u and declare u scanned

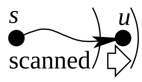


Fig. 10.4. Dijkstra's shortest-path algorithm for nonnegative edge weights

Operation	Queue
$insert(s)$	$\langle(s,0)\rangle$
$deleteMin \rightsquigarrow (s,0)$	$\langle\rangle$
$relax\ s \xrightarrow{2} a$	$\langle(a,2)\rangle$
$relax\ s \xrightarrow{10} d$	$\langle(a,2),(d,10)\rangle$
$deleteMin \rightsquigarrow (a,2)$	$\langle(d,10)\rangle$
$relax\ a \xrightarrow{3} b$	$\langle(b,5),(d,10)\rangle$
$deleteMin \rightsquigarrow (b,5)$	$\langle(d,10)\rangle$
$relax\ b \xrightarrow{2} c$	$\langle(c,7),(d,10)\rangle$
$relax\ b \xrightarrow{1} e$	$\langle(e,6),(c,7),(d,10)\rangle$
$deleteMin \rightsquigarrow (e,6)$	$\langle(c,7),(d,10)\rangle$
$relax\ e \xrightarrow{9} b$	$\langle(c,7),(d,10)\rangle$
$relax\ e \xrightarrow{8} c$	$\langle(c,7),(d,10)\rangle$
$relax\ e \xrightarrow{0} d$	$\langle(d,6),(c,7)\rangle$
$deleteMin \rightsquigarrow (d,6)$	$\langle(c,7)\rangle$
$relax\ d \xrightarrow{4} s$	$\langle(c,7)\rangle$
$relax\ d \xrightarrow{5} b$	$\langle(c,7)\rangle$
$deleteMin \rightsquigarrow (c,7)$	$\langle\rangle$

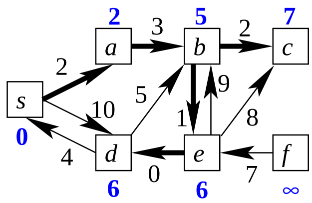


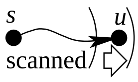
Fig. 10.5. Example run of Dijkstra's algorithm on the graph given on the *right*. The bold edges form the shortest-path tree, and the numbers in bold indicate shortest-path distances. The table on the *left* illustrates the execution. The *queue* contains all pairs $(v, d[v])$ with v reached and unscanned. A node is called *reached* if its tentative distance is less than $+\infty$. Initially, s is reached and unscanned. The actions of the algorithm are given in the first column. The second column shows the state of the queue after the action

Proof. We proceed in two steps. In the first step, we show that all nodes reachable from s are scanned. In the second step, we show that the tentative and true distances agree when a node is scanned. In both steps, we argue by contradiction.

For the first step, assume the existence of a node v that is reachable from s, but never scanned. Consider a shortest path $p = \langle s = v_1, v_2, \ldots, v_k = v\rangle$ from s to v, and let i be minimal such that v_i is not scanned. Then $i > 1$, since s is the first node scanned (in the first iteration, s is the only node whose tentative distance is less than $+\infty$) . By the definition of i, v_{i-1} has been scanned. When v_{i-1} is scanned, $d[v_i]$ is set to $d[v_{i-1}] + c(v_{i-1}, v_i)$, a value less than $+\infty$. So v_i must be scanned at some point during the execution, since the only nodes that stay unscanned are nodes u with $d[u] = +\infty$ at termination.

For the second step, consider the first point in time t, when a node v is scanned with $\mu[v] < d(v)$. As above, consider a shortest path $p = \langle s = v_1, v_2, \ldots, v_k = v\rangle$ from s to v, and let i be minimal such that v_i is not scanned before time t. Then $i > 1$, since s is the first node scanned and $\mu(s) = 0 = d[s]$ when s is scanned. By the definition of i,

```
Function Dijkstra(s : NodeId) : NodeArray×NodeArray                  // returns (d, parent)
    d = ⟨∞,...,∞⟩ : NodeArray of ℝ ∪ {∞}                       // tentative distance from root
    parent = ⟨⊥,...,⊥⟩ : NodeArray of NodeId
    parent[s] := s                                                      // self-loop signals root
    Q : NodePQ                                                      // unscanned reached nodes
    d[s] := 0;    Q.insert(s)
    while Q ≠ ∅ do
        u := Q.deleteMin                                                   // we have d[u] = μ(u)
        foreach edge e = (u,v) ∈ E do
            if d[u] + c(e) < d[v] then                                              // relax
                d[v] := d[u] + c(e)
                parent[v] := u                                                 // update tree
                if v ∈ Q then Q.decreaseKey(v)
                else Q.insert(v)
    return (d, parent)
```



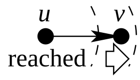


Fig. 10.6. Pseudocode for Dijkstra's algorithm

v_{i-1} was scanned before time t. Hence $d[v_{i-1}] = \mu(v_{i-1})$ when v_{i-1} is scanned. When v_{i-1} is scanned, $d[v_i]$ is set to $d[v_{i-1}] + c(v_{i-1}, v_i) = \mu(v_{i-1}) + c(v_{i-1}, v_i) = \mu(v_i)$. So, at time t, we have $d[v_i] = \mu(v_i) \leq \mu(v_k) < d[v_k]$ and hence v_i is scanned instead of v_k, a contradiction. $\square$

Exercise 10.7. Let $v_1, v_2, \ldots$ be the order in which the nodes are scanned. Show that $\mu(v_1) \leq \mu(v_2) \leq \ldots$, i.e., the nodes are scanned in order of increasing shortest-path distance.

Exercise 10.8 (checking of shortest-path distances). Assume that all edge costs are positive, that all nodes are reachable from s, and that d is a node array of nonnegative reals satisfying $d[s] = 0$ and $d[v] = \min_{(u,v)\in E} d[u] + c(u,v)$ for $v \neq s$. Show that $d[v] = \mu(v)$ for all v. Does the claim still hold in the presence of edges of cost zero?

We come now to the implementation of Dijkstra's algorithm. We store all unscanned reached nodes in an addressable priority queue (see Sect. 6.2) using their tentative-distance values as keys. Thus, we can extract the next node to be scanned using the queue operation *deleteMin*. We need a variant of a priority queue where the operation *decreaseKey* addresses queue items using nodes rather than handles. Given an ordinary priority queue, such a *NodePQ* can be implemented using an additional *NodeArray* translating nodes into handles. We can also store the priority queue items directly in a *NodeArray*. We obtain the algorithm given in Fig. 10.6. Next, we analyze its running time in terms of the running times for the queue operations. Initializing the arrays d and *parent* and setting up a priority queue $Q = \{s\}$ takes time $O(n)$. Checking for $Q = \emptyset$ and loop control takes constant time per iteration of the while loop, i.e., $O(n)$ time in total. Every node reachable from s is removed from the queue exactly once. Every reachable node is also *insert*ed exactly once. Thus we have at most n *deleteMin* and *insert* operations. Since each node is scanned at most once, each edge is relaxed at most once, and hence there can be at most m *decreaseKey* operations. We obtain a total execution time of

$$T_{\text{Dijkstra}} = \mathrm{O}\big(m \cdot T_{decreaseKey}(n) + n \cdot (T_{deleteMin}(n) + T_{insert}(n))\big)\,,$$

where $T_{deleteMin}$, T_{insert}, and $T_{decreaseKey}$ denote the execution times for *deleteMin*, *insert*, and *decreaseKey*, respectively. Note that these execution times are a function of the queue size $|Q| = \mathrm{O}(n)$.

Exercise 10.9. Design a graph and a nonnegative cost function such that the relaxation of $m-(n-1)$ edges causes a *decreaseKey* operation.

In his original 1959 paper, Dijkstra proposed the following implementation of the priority queue: maintain the number of reached unscanned nodes, and two arrays indexed by nodes – an array d storing the tentative distances and an array storing, for each node, whether it is unscanned or reached. Then *insert* and *decreaseKey* take time $\mathrm{O}(1)$. A *deleteMin* takes time $\mathrm{O}(n)$, since it has to scan the arrays in order to find the minimum tentative distance of any reached unscanned node. Thus the total running time becomes

$$T_{Dijkstra59} = \mathrm{O}\big(m + n^2\big)\ .$$

Much better priority queue implementations have been invented since Dijkstra's original paper. Using the binary heap and Fibonacci heap priority queues described in Sect. 6.2, we obtain

$$T_{DijkstraBHeap} = \mathrm{O}((m+n)\log n)$$

and

$$T_{DijkstraFibonacci} = \mathrm{O}(m + n\log n)\,,$$

respectively. Asymptotically, the Fibonacci heap implementation is superior except for sparse graphs with $m = \mathrm{O}(n)$. In practice, Fibonacci heaps are usually not the fastest implementation, because they involve larger constant factors and the actual number of *decreaseKey* operations tends to be much smaller than what the worst case predicts. This experimental observation will be supported by theoretical analysis in the next section.

10.4 *Average-Case Analysis of Dijkstra's Algorithm

We shall show that the expected number of *decreaseKey* operations is $\mathrm{O}(n\log(m/n))$.

Our model of randomness is as follows. The graph G and the source node s are arbitrary. Also, for each node v, we have an arbitrary set $C(v)$ of *indegree*(v) nonnegative real numbers. So far, everything is arbitrary. The randomness comes now: we assume that, for each v, the costs in $C(v)$ are assigned randomly to the edges *into* v, i.e., our probability space consists of $\prod_{v\in V}$ *indegree*(v)! assignments of edge costs to edges. We want to stress that this model is quite general. In particular, it covers the situation where edge costs are drawn independently from a common distribution.

Theorem 10.6. *Under the assumptions above, the expected number of decreaseKey operations is* $\mathrm{O}(n\log(m/n))$.

Proof. We present a proof due to Noshita [151]. Consider a particular node v. In any run of Dijkstra's algorithm, the edges whose relaxation can cause *decreaseKey* operations for v have the form $e_i := (u_i, v)$, where $\mu(u_i) \le \mu(v)$. Say there are k such edges $e_1, \ldots, e_k$. We number them in the order in which their source nodes u_i are scanned. We then have $\mu(u_1) \le \mu(u_2) \le \ldots \le \mu(u_k) \le \mu(v)$. These edges are relaxed in the order $e_1, \ldots, e_k$, no matter how the costs in $C(v)$ are assigned to them. If e_i causes a *decreaseKey* operation, then

$$\mu(u_i) + c(e_i) < \min_{j<i} \mu(u_j) + c(e_j)\ .$$

Since $\mu(u_j) \le \mu(u_i)$, this implies

$$c(e_i) < \min_{j<i} c(e_j),$$

i.e., only left-to-right minima of the sequence $c(e_1), \ldots, c(e_k)$ can cause *decreaseKey* operations. We conclude that the number of *decreaseKey* operations on v is bounded by the number of left-to-right minima in the sequence $c(e_1), \ldots, c(e_k)$ minus one; the "-1" accounts for the fact that the first element in the sequence counts as a left-to-right minimum but causes an *insert* and no *decreaseKey*. In Sect. 2.8, we have shown that the expected number of left-to-right maxima in a permutation of size k is bounded by H_k. The same bound holds for minima. Thus the expected number of *decreaseKey* operations is bounded by $H_k - 1$, which in turn is bounded by $\ln k$. Also, $k \le \mathit{indegree}(v)$. Summing over all nodes, we obtain the following bound for the expected number of *decreaseKey* operations:

$$\sum_{v\in V} \ln \mathit{indegree}(v) \le n \ln \frac{m}{n}\ ,$$

where the last inequality follows from the concavity of the ln function (see (A.15)). □

We conclude that the expected running time is $\mathrm{O}(m + n\log(m/n)\log n)$ with the binary heap implementation of priority queues. For sufficiently dense graphs ($m > n\log n \log\log n$), we obtain an execution time linear in the size of the input.

Exercise 10.10. Show that $\mathrm{E}[T_{DijkstraBHeap}] = \mathrm{O}(m)$ if $m = \Omega(n\log n\log\log n)$.

10.5 Monotone Integer Priority Queues

Dijkstra's algorithm is designed to scan nodes in order of nondecreasing distance values. Hence, a monotone priority queue (see Chapter 6) suffices for its implementation. It is not known whether monotonicity can be exploited in the case of general real edge costs. However, for integer edge costs, significant savings are possible. We therefore assume in this section that edge costs are integers in the range $0..C$ for some integer C. C is assumed to be known when the queue is initialized.

Since a shortest path can consist of at most $n-1$ edges, the shortest-path distances are at most $(n-1)C$. The range of values in the queue at any one time is even smaller. Let *min* be the last value deleted from the queue (zero before the first deletion). Dijkstra's algorithm maintains the invariant that all values in the queue are contained in $min..min+C$. The invariant certainly holds after the first insertion. A *deleteMin* may increase *min*. Since all values in the queue are bounded by C plus the old value of *min*, this is certainly true for the new value of *min*. Edge relaxations insert priorities of the form $d[u]+c(e) = min+c(e) \in min..min+C$.

10.5.1 Bucket Queues

A bucket queue is a circular array B of $C+1$ doubly linked lists (see Figs. 10.7 and 3.8). We view the natural numbers as being wrapped around the circular array; all integers of the form $i+(C+1)j$ map to the index i. A node $v \in Q$ with tentative distance $d[v]$ is stored in $B[d[v] \bmod (C+1)]$. Since the priorities in the queue are always in $min..min+C$, all nodes in a bucket have the *same* distance value.

Initialization creates $C+1$ empty lists. An *insert*(v) inserts v into $B[d[v] \bmod C+1]$. A *decreaseKey*(v) removes v from the list containing it and inserts v into $B[d[v] \bmod C+1]$. Thus *insert* and *decreaseKey* take constant time if buckets are implemented as doubly linked lists.

A *deleteMin* first looks at bucket $B[min \bmod C+1]$. If this bucket is empty, it increments *min* and repeats. In this way, the total cost of all *deleteMin* operations is $O(n+nC) = O(nC)$, since *min* is incremented at most nC times and at most n elements are deleted from the queue. Plugging the operation costs for the bucket queue implementation with integer edge costs in $0..C$ into our general bound for the cost of Dijkstra's algorithm, we obtain

$$T_{\mathrm{DijkstraBucket}} = O(m+nC)\,.$$

***Exercise 10.11 (Dinic's refinement of bucket queues [57]).** Assume that the edge costs are positive real numbers in $[c_{\min}, c_{\max}]$. Explain how to find shortest paths in time $O(m+nc_{\max}/c_{\min})$. Hint: use buckets of width $c_{\min}$. Show that all nodes in the smallest nonempty bucket have $d[v] = \mu(v)$.

10.5.2 *Radix Heaps

Radix heaps [9] improve on the bucket queue implementation by using buckets of different widths. Narrow buckets are used for tentative distances close to *min*, and

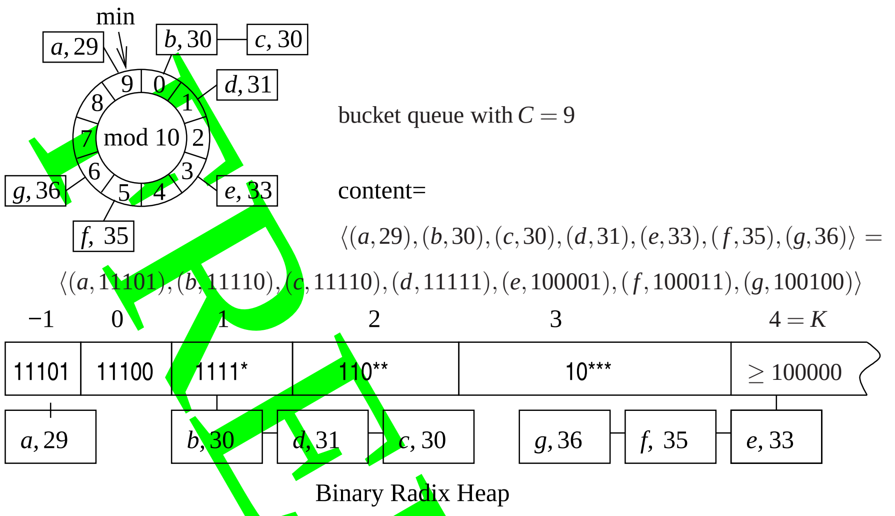


Fig. 10.7. Example of a bucket queue (*upper part*) and a radix heap (*lower part*). Since $C = 9$, we have $K = 1 + \lfloor \log C \rfloor = 4$. The bit patterns in the buckets of the radix heap indicate the set of keys they can accommodate

wide buckets are used for tentative distances far away from *min*. In this subsection, we shall show how this approach leads to a version of Dijkstra's algorithm with running time

$$T_{\text{DijkstraRadix}} := \mathrm{O}(m + n \log C) .$$

Radix heaps exploit the binary representation of tentative distances. We need the concept of the *most significant distinguishing index* of two numbers. This is the largest index where the binary representations differ, i.e., for numbers a and b with binary representations $a = \sum_{i \ge 0} \alpha_i 2^i$ and $b = \sum_{i \ge 0} \beta_i 2^i$, we define the most significant distinguishing index $msd(a,b)$ as the largest i with $\alpha_i \ne \beta_i$, and let it be -1 if $a = b$. If $a < b$, then a has a zero bit in position $i = msd(a,b)$ and b has a one bit.

A radix heap consists of an array of buckets $B[-1]$, $B[0]$, ..., $B[K]$, where $K = 1 + \lfloor \log C \rfloor$. The queue elements are distributed over the buckets according to the following rule:

any queue element v is stored in bucket $B[i]$, where $i = \min(msd(min, d[v]), K)$.

We refer to this rule as the bucket queue invariant. Figure 10.7 gives an example. We remark that if *min* has a one bit in position i for $0 \le i < K$, the corresponding bucket $B[i]$ is empty. This holds since any $d[v]$ with $i = msd(min, d[v])$ would have a zero bit in position i and hence be smaller than *min*. But all keys in the queue are at least as large as *min*.

How can we compute $i := msd(a,b)$? We first observe that for $a \ne b$, the bitwise exclusive OR $a \oplus b$ of a and b has its most significant one in position i and hence represents an integer whose value is at least 2^i and less than 2^{i+1}. Thus $msd(a,b) =$

$\lfloor \log(a \oplus b) \rfloor$, since $\log(a \oplus b)$ is a real number with its integer part equal to i and the floor function extracts the integer part. Many processors support the computation of *msd* by machine instructions.[3] Alternatively, we can use lookup tables or yet other solutions. From now on, we shall assume that *msd* can be evaluated in constant time.

We turn now to the queue operations. Initialization, *insert*, and *decreaseKey* work completely analogously to bucket queues. The only difference is that bucket indices are computed using the bucket queue invariant.

A *deleteMin* first finds the minimum i such that $B[i]$ is nonempty. If $i = -1$, an arbitrary element in $B[-1]$ is removed and returned. If $i \geq 0$, the bucket $B[i]$ is scanned and *min* is set to the smallest tentative distance contained in the bucket. Since *min* has changed, the bucket queue invariant needs to be restored. A crucial observation for the efficiency of radix heaps is that only the nodes in bucket i are affected. We shall discuss below how they are affected. Let us consider first the buckets $B[j]$ with $j \neq i$. The buckets $B[j]$ with $j < i$ are empty. If $i = K$, there are no j's with $j > K$. If $i < K$, any key a in bucket $B[j]$ with $j > i$ will still have $msd(a, min) = j$, because the old and new values of *min* agree at bit positions greater than i.

What happens to the elements in $B[i]$? Its elements are moved to the appropriate new bucket. Thus a *deleteMin* takes constant time if $i = -1$ and takes time $O(i + |B[i]|) = O(K + |B[i]|)$ if $i \geq 0$. Lemma 10.7 below shows that every node in bucket $B[i]$ is moved to a bucket with a smaller index. This observation allows us to account for the cost of a *deleteMin* using amortized analysis. As our unit of cost (one token), we shall use the time required to move one node between buckets.

We charge $K+1$ tokens for operation $insert(v)$ and associate these $K+1$ tokens with v. These tokens pay for the moves of v to lower-numbered buckets in *deleteMin* operations. A node starts in some bucket j with $j \leq K$, ends in bucket -1, and in between never moves back to a higher-numbered bucket. Observe that a $decreaseKey(v)$ operation will also never move a node to a higher-numbered bucket. Hence, the $K+1$ tokens can pay for all the node moves of *deleteMin* operations. The remaining cost of a *deleteMin* is $O(K)$ for finding a nonempty bucket. With amortized costs $K + 1 + O(1) = O(K)$ for an *insert* and $O(1)$ for a *decreaseKey*, we obtain a total execution time of $O(m + n \cdot (K + K)) = O(m + n \log C)$ for Dijkstra's algorithm, as claimed.

It remains to prove that *deleteMin* operations move nodes to lower-numbered buckets.

Lemma 10.7. *Let i be minimal such that $B[i]$ is nonempty and assume $i \geq 0$. Let min be the smallest element in $B[i]$. Then $msd(min, x) < i$ for all $x \in B[i]$.*

[3] $\oplus$ is a direct machine instruction, and $\lfloor \log x \rfloor$ is the exponent in the floating-point representation of x.

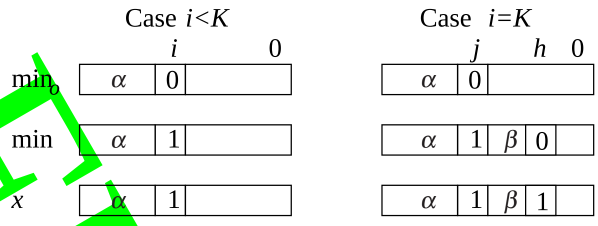


Fig. 10.8. The structure of the keys relevant to the proof of Lemma 10.7. In the proof, it is shown that β starts with $j-K$ zeros

Proof. Observe first that the case $x = min$ is easy, since $msd(x,x) = -1 < i$. For the nontrivial case $x \neq min$, we distinguish the subcases $i < K$ and $i = K$. Let min_o be the old value of min. Figure 10.8 shows the structure of the relevant keys.

Case $i < K$. The most significant distinguishing index of min_o and any $x \in B[i]$ is i, i.e., min_o has a zero in bit position i, and all $x \in B[i]$ have a one in bit position i. They agree in all positions with an index larger than i. Thus the most significant distinguishing index for min and x is smaller than i.

Case $i = K$. Consider any $x \in B[K]$. Let $j = msd(min_o, min)$. Then $j \geq K$, since $min \in B[K]$. Let $h = msd(min, x)$. We want to show that $h < K$. Let α comprise the bits in positions larger than j in min_o, and let A be the number obtained from min_o by setting the bits in positions 0 to j to zero. Then α followed by $j+1$ zeros is the binary representation of A. Since the j-th bit of min_o is zero and that of min is one, we have $min_o < A + 2^j$ and $A + 2^j \leq min$. Also, $x \leq min_o + C < A + 2^j + C \leq A + 2^j + 2^K$. So

$$A + 2^j \leq min \leq x < A + 2^j + 2^K,$$

and hence the binary representations of min and x consist of α followed by a 1, followed by $j-K$ zeros, followed by some bit string of length K. Thus min and x agree in all bits with index K or larger, and hence $h < K$.

In order to aid intuition, we give a second proof for the case $i = K$. We first observe that the binary representation of min starts with α followed by a one. We next observe that x can be obtained from min_o by adding some K-bit number. Since $min \leq x$, the final carry in this addition must run into position j. Otherwise, the j-th bit of x would be zero and hence $x < min$. Since min_o has a zero in position j, the carry stops at position j. We conclude that the binary representation of x is equal to α followed by a 1, followed by $j-K$ zeros, followed by some K-bit string. Since $min \leq x$, the $j-K$ zeros must also be present in the binary representation of min. □

***Exercise 10.12.** Radix heaps can also be based on number representations with base b for any $b \geq 2$. In this situation we have buckets $B[i,j]$ for $i = -1, 0, 1, \ldots, K$ and $0 \leq j \leq b$, where $K = 1 + \lfloor \log C / \log b \rfloor$. An unscanned reached node x is stored in bucket $B[i,j]$ if $msd(min, d[x]) = i$ and the i-th digit of $d[x]$ is equal to j. We also store, for each i, the number of nodes contained in the buckets $\cup_j B[i,j]$. Discuss the implementation of the priority queue operations and show that a shortest-path algorithm with running time $\mathrm{O}(m+n(b+\log C/\log b))$ results. What is the optimal choice of b?

If the edge costs are random integers in the range $0..C$, a small change to Dijkstra's algorithm with radix heaps guarantees linear running time [139, 76]. For every node v, let $c^{\text{in}}_{\min}(v)$ denote the minimum cost of an incoming edge. We divide Q into two parts, a set F which contains unscanned nodes whose tentative-distance label is known to be equal to their exact distance from s, and a part B which contains all other reached unscanned nodes. B is organized as a radix heap. We also maintain a value *min*. We scan nodes as follows.

When F is nonempty, an arbitrary node in F is removed and the outgoing edges are relaxed. When F is empty, the minimum node is selected from B and *min* is set to its distance label. When a node is selected from B, the nodes in the first nonempty bucket $B[i]$ are redistributed if $i \geq 0$. There is a small change in the redistribution process. When a node v is to be moved, and $d[v] \leq min + c^{\text{in}}_{\min}(v)$, we move v to F. Observe that any future relaxation of an edge into v cannot decrease $d[v]$, and hence $d[v]$ is known to be exact at this point.

We call this algorithm ALD (average-case linear Dijkstra). The algorithm ALD is correct, since it is still true that $d[v] = \mu(v)$ when v is scanned. For nodes removed from F, this was argued in the previous paragraph, and for nodes removed from B, this follows from the fact that they have the smallest tentative distance among all unscanned reached nodes.

Theorem 10.8. *Let G be an arbitrary graph and let c be a random function from E to $0..C$. The algorithm ALD then solves the single-source problem in expected time $\mathrm{O}(m+n)$.*

Proof. We still need to argue the bound on the running time. To do this, we modify the amortized analysis of plain radix heaps. As before, nodes start out in $B[K]$. When a node v has been moved to a new bucket but not yet to F, $d[v] > min + c^{\text{in}}_{\min}(v)$, and hence v is moved to a bucket $B[i]$ with $i \geq \log c^{\text{in}}_{\min}(v)$. Hence, it suffices if *insert* pays $K - \log c^{\text{in}}_{\min}(v) + 1$ tokens into the account for node v in order to cover all costs due to *decreaseKey* and *deleteMin* operations operating on v. Summing over all nodes, we obtain a total payment of

$$\sum_v (K - \log c^{\text{in}}_{\min}(v) + 1) = n + \sum_v (K - \log c^{\text{in}}_{\min}(v)) \ .$$

We need to estimate this sum. For each vertex, we have one incoming edge contributing to this sum. We therefore bound the sum from above if we sum over all edges, i.e.,

$$\sum_v (K - \log c^{\text{in}}_{\min}(v)) \leq \sum_e (K - \log c(e)) \ .$$

$K - \log c(e)$ is the number of leading zeros in the binary representation of $c(e)$ when written as a K-bit number. Our edge costs are uniform random numbers in $0..C$, and $K = 1 + \lfloor \log C \rfloor$. Thus $\mathrm{prob}(K - \log c(e) = i) = 2^{-i}$. Using (A.14), we conclude that

$$\mathrm{E}\left[\sum_e (k - \log c(e))\right] = \sum_e \sum_{i \geq 0} i 2^{-i} = \mathrm{O}(m)\,.$$

Thus the total expected cost of the *deleteMin* and *decreaseKey* operations is $\mathrm{O}(m+n)$. The time spent outside these operations is also $\mathrm{O}(m+n)$. □

It is a little odd that the maximum edge cost C appears in the premise but not in the conclusion of Theorem 10.8. Indeed, it can be shown that a similar result holds for random real-valued edge costs.

****Exercise 10.13.** Explain how to adapt the algorithm ALD to the case where c is a random function from E to the real interval $(0,1]$. The expected time should still be $\mathrm{O}(m+n)$. What assumptions do you need about the representation of edge costs and about the machine instructions available? Hint: you may first want to solve Exercise 10.11. The narrowest bucket should have a width of $\min_{e \in E} c(e)$. Subsequent buckets have geometrically growing widths.

10.6 Arbitrary Edge Costs (Bellman–Ford Algorithm)

For acyclic graphs and for nonnegative edge costs, we got away with m edge relaxations. For arbitrary edge costs, no such result is known. However, it is easy to guarantee the correctness criterion of Lemma 10.3 using $\mathrm{O}(n \cdot m)$ edge relaxations: the Bellman–Ford algorithm [18, 63] given in Fig. 10.9 performs $n-1$ rounds. In each round, it relaxes all edges. Since simple paths consist of at most $n-1$ edges, every shortest path is a subsequence of this sequence of relaxations. Thus, after the relaxations are completed, we have $d[v] = \mu(v)$ for all v with $-\infty < d[v] < \infty$, by Lemma 10.3. Moreover, *parent* encodes the shortest paths to these nodes. Nodes v unreachable from s will still have $d[v] = \infty$, as desired.

It is not so obvious how to find the nodes v with $\mu(v) = -\infty$. Consider any edge $e = (u,v)$ with $d[u] + c(e) < d[v]$ after the relaxations are completed. We can set $d[v] := -\infty$ because if there were a shortest path from s to v, we would have found it by now and relaxing e would not lead to shorter distances anymore. We can now also set $d[w] = -\infty$ for all nodes w reachable from v. The pseudocode implements this approach using a recursive function *infect*(v). It sets the d-value of v and all nodes reachable from it to $-\infty$. If *infect* reaches a node w that already has $d[w] = -\infty$, it breaks the recursion because previous executions of *infect* have already explored all nodes reachable from w. If $d[v]$ is not set to $-\infty$ during postprocessing, we have $d[x] + c(e) \geq d[y]$ for any edge $e = (x,y)$ on any path p from s to v. Thus $d[s] + c(p) \geq d[v]$ for any path p from s to v, and hence $d[v] \leq \mu(v)$. We conclude that $d[v] = \mu(v)$.

Exercise 10.14. Show that the postprocessing runs in time $\mathrm{O}(m)$. Hint: relate *infect* to *DFS*.

Exercise 10.15. Someone proposes an alternative postprocessing algorithm: set $d[v]$ to $-\infty$ for all nodes v for which following parents does not lead to s. Give an example where this method overlooks a node with $\mu(v) = -\infty$.

```
Function BellmanFord(s : NodeId) : NodeArray×NodeArray
    d = ⟨∞,...,∞⟩ : NodeArray of ℝ∪{−∞,∞}              // distance from root
    parent = ⟨⊥,...,⊥⟩ : NodeArray of NodeId
    d[s] := 0;    parent[s] := s                       // self-loop signals root
    for i := 1 to n − 1 do
        forall e ∈ E do relax(e)                       // round i
    forall e = (u,v) ∈ E do                            // postprocessing
        if d[u] + c(e) < d[v] then infect(v)
    return (d, parent)

Procedure infect(v)
    if d[v] > −∞ then
        d[v] := −∞
        foreach (v,w) ∈ E do infect(w)
```

Fig. 10.9. The Bellman–Ford algorithm for shortest paths in arbitrary graphs

Exercise 10.16 (arbitrage). Consider a set of currencies C with an exchange rate of r_{ij} between currencies i and j (you obtain r_{ij} units of currency j for one unit of currency i). A *currency arbitrage* is possible if there is a sequence of elementary currency exchange actions (*transactions*) that starts with one unit of a currency and ends with more than one unit of the same currency. (a) Show how to find out whether a matrix of exchange rates admits currency arbitrage. Hint: $\log(xy) = \log x + \log y$. (b) Refine your algorithm so that it outputs a sequence of exchange steps that maximizes the average profit *per transaction*.

Section 10.10 outlines further refinements of the Bellman–Ford algorithm that are necessary for good performance in practice.

10.7 All-Pairs Shortest Paths and Node Potentials

The all-pairs problem is tantamount to n single-source problems and hence can be solved in time $\mathrm{O}(n^2 m)$. A considerable improvement is possible. We shall show that it suffices to solve one general single-source problem plus n single-source problems with nonnegative edge costs. In this way, we obtain a running time of $\mathrm{O}(nm + n(m + n\log n)) = \mathrm{O}(nm + n^2 \log n)$. We need the concept of node potentials.

A *(node) potential function* assigns a number $pot(v)$ to each node v. For an edge $e = (v,w)$, we define its *reduced cost* $\bar{c}(e)$ as

$$\bar{c}(e) = pot(v) + c(e) - pot(w) \ .$$

Lemma 10.9. *Let p and q be paths from v to w. Then $\bar{c}(p) = pot(v) + c(p) - pot(w)$ and $\bar{c}(p) \leq \bar{c}(q)$ iff $c(p) \leq c(q)$. In particular, the shortest paths with respect to $\bar{c}$ are the same as those with respect to c.*

All-Pairs Shortest Paths in the Absence of Negative Cycles
add a new node s and zero length edges (s,v) for all v // no new cycles, time $\mathrm{O}(m)$
compute $\mu(v)$ for all v with Bellman–Ford // time $\mathrm{O}(nm)$
set $pot(v) = \mu(v)$ and compute reduced costs $\bar{c}(e)$ for $e \in E$ // time $\mathrm{O}(m)$
forall nodes x **do** // time $\mathrm{O}(n(m + n\log n))$
  use Dijkstra's algorithm to compute the reduced shortest-path distances $\bar{\mu}(x,v)$
  using source x and the reduced edge costs $\bar{c}$
// translate distances back to original cost function // time $\mathrm{O}(m)$
forall $e = (v,w) \in V \times V$ **do** $\mu(v,w) := \bar{\mu}(v,w) + pot(w) - pot(v)$ // use Lemma 10.9

Fig. 10.10. Algorithm for all-pairs shortest paths in the absence of negative cycles

Proof. The second and the third claim follow from the first. For the first claim, let $p = \langle e_0, \ldots, e_{k-1} \rangle$, where $e_i = (v_i, v_{i+1})$, $v = v_0$, and $w = v_k$. Then

$$\begin{aligned} \bar{c}(p) &= \sum_{i=0}^{k-1} \bar{c}(e_i) = \sum_{0 \le i < k} (pot(v_i) + c(e_i) - pot(v_{i+1})) \\ &= pot(v_0) + \sum_{0 \le i < k} c(e_i) - pot(v_k) = pot(v_0) + c(p) - pot(v_k)\,. \end{aligned}$$ □

Exercise 10.17. Node potentials can be used to generate graphs with negative edge costs but no negative cycles: generate a (random) graph, assign to every edge e a (random) nonnegative cost $c(e)$, assign to every node v a (random) potential $pot(v)$, and set the cost of $e = (u,v)$ to $\bar{c}(e) = pot(u) + c(e) - pot(v)$. Show that this rule does not generate negative cycles.

Lemma 10.10. *Assume that G has no negative cycles and that all nodes can be reached from s. Let $pot(v) = \mu(v)$ for $v \in V$. With these node potentials, the reduced edge costs are nonnegative.*

Proof. Since all nodes are reachable from s and since there are no negative cycles, $\mu(v) \in \mathbb{R}$ for all v. Thus the reduced costs are well defined. Consider an arbitrary edge $e = (v,w)$. We have $\mu(v) + c(e) \ge \mu(w)$, and thus $\bar{c}(e) = \mu(v) + c(e) - \mu(w) \ge 0$. □

Theorem 10.11. *The all-pairs shortest-path problem for a graph without negative cycles can be solved in time $\mathrm{O}\left(nm + n^2 \log n\right)$.*

Proof. The algorithm is shown in Fig. 10.10. We add an auxiliary node s and zero-cost edges (s,v) for all nodes of the graph. This does not create negative cycles and does not change $\mu(v,w)$ for any of the existing nodes. Then we solve the single-source problem for the source s, and set $pot(v) = \mu(v)$ for all v. Next we compute the reduced costs and then solve the single-source problem for each node x by means of Dijkstra's algorithm. Finally, we translate the computed distances back to the original cost function. The computation of the potentials takes time $\mathrm{O}(nm)$, and the n shortest-path calculations take time $\mathrm{O}(n(m + n\log n))$. The preprocessing and postprocessing take linear time. □

The assumption that G has no negative cycles can be removed [133].

Exercise 10.18. The *diameter* D of a graph G is defined as the largest distance between any two of its nodes. We can easily compute it using an all-pairs computation. Now we want to consider ways to *approximate* the diameter of a strongly connected graph using a constant number of single-source computations. (a) For any starting node s, let $D'(s) := \max_{u \in V} \mu(u)$. Show that $D'(s) \leq D \leq 2D'(s)$ for undirected graphs. Also, show that no such relation holds for directed graphs. Let $D''(s) := \max_{u \in V} \mu(u,s)$. Show that $\max(D'(s), D''(s)) \leq D \leq D'(s) + D''(s)$ for both undirected and directed graphs. (b) How should a graph be represented to support both forward and backward search? (c) Can you improve the approximation by considering more than one node s?

10.8 Shortest-Path Queries

We are often interested in the shortest path from a specific source node s to a specific target node t; route planning in a traffic network is one such scenario. We shall explain some techniques for solving such *shortest-path queries* efficiently and argue for their usefulness for the route-planning application.

We start with a technique called *early stopping*. We run Dijkstra's algorithm to find shortest paths starting at s. We stop the search as soon as t is removed from the priority queue, because at this point in time the shortest path to t is known. This helps except in the unfortunate case where t is the node farthest from s. On average, early stopping saves a factor of two in scanned nodes in any application. In practical route planning, early stopping saves much more because modern car navigation systems have a map of an entire continent but are mostly used for distances up to a few hundred kilometers.

Another simple and general heuristic is *bidirectional search*, from s forward and from t backward until the search frontiers meet. More precisely, we run two copies of Dijkstra's algorithm side by side, one starting from s and one starting from t (and running on the reversed graph). The two copies have their own queues, say Q_s and Q_t, respectively. We grow the search regions at about the same speed, for example by removing a node from Q_s if $\min Q_s \leq \min Q_t$ and a node from Q_t otherwise.

It is tempting to stop the search once the first node u has been removed from both queues, and to claim that $\mu(t) = \mu(s,u) + \mu(u,t)$. Observe that execution of Dijkstra's algorithm on the reversed graph with a starting node t determines $\mu(u,t)$. This is not quite correct, but almost so.

Exercise 10.19. Give an example where u is *not* on the shortest path from s to t.

However, we have collected enough information once some node u has been removed from both queues. Let d_s and d_t denote the tentative-distance labels at the time of termination in the runs with source s and source t, respectively. We show that $\mu(t) < \mu(s,u) + \mu(u,t)$ implies the existence of a node $v \in Q_s$ with $\mu(t) = d_s[v] + d_t[v]$.

Let $p = \langle s = v_0, \ldots, v_i, v_{i+1}, \ldots, v_k = t \rangle$ be a shortest path from s to t. Let i be maximal such that v_i has been removed from Q_s. Then $d_s[v_{i+1}] = \mu(s, v_{i+1})$ and $v_{i+1} \in Q_s$ when the search stops. Also, $\mu(s,u) \leq \mu(s, v_{i+1})$, since u has already been removed from Q_s, but v_{i+1} has not. Next, observe that

$$\mu(s, v_{i+1}) + \mu(v_{i+1}, t) = c(p) < \mu(s,u) + \mu(u,t) \ ,$$

since p is a shortest path from s to t. By subtracting $\mu(s, v_{i+1})$, we obtain

$$\mu(v_{i+1}, t) < \underbrace{\mu(s,u) - \mu(s, v_{i+1})}_{\leq 0} + \mu(u,t) \leq \mu(u,t)$$

and hence, since u has been scanned from t, v_{i+1} must also have been scanned from t, i.e., $d_t[v_{i+1}] = \mu(v_{i+1}, t)$ when the search stops. So we can determine the shortest distance from s to t by inspecting not only the first node removed from both queues, but also all nodes in, say, Q_s. We iterate over all such nodes v and determine the minimum value of $d_s[v] + d_t[v]$.

Dijkstra's algorithm scans nodes in order of increasing distance from the source. In other words, it grows a circle centered on the source node. The circle is defined by the shortest-path metric in the graph. In the route-planning application for a road network, we may also consider geometric circles centered on the source and argue that shortest-path circles and geometric circles are about the same. We can then estimate the speedup obtained by bidirectional search using the following heuristic argument: a circle of a certain diameter has twice the area of two circles of half the diameter. We could thus hope that bidirectional search will save a factor of two compared with unidirectional search.

Exercise 10.20 (bidirectional search). (a) Consider bidirectional search in a grid graph with unit edge weights. How much does it save over unidirectional search? (*****b) Try to find a family of graphs where bidirectional search visits exponentially fewer nodes on average than does unidirectional search. Hint: consider random graphs or hypercubes. (c) Give an example where bidirectional search in a real road network takes *longer* than unidirectional search. Hint: consider a densely inhabitated city with sparsely populated surroundings. (d) Design a strategy for switching between forward and backward search such that bidirectional search will *never* inspect more than twice as many nodes as unidirectional search.

We shall next describe two techniques that are more complex and less generally applicable: however, if they are applicable, they usually result in larger savings. Both techniques mimic human behavior in route planning.

10.8.1 Goal-Directed Search

The idea is to bias the search space such that Dijkstra's algorithm does not grow a disk but a region protruding toward the target; see Fig. 10.11. Assume we know a function $f : V \to \mathbb{R}$ that estimates the distance to the target, i.e., $f(v)$ estimates

$\mu(v,t)$ for all nodes v. We use the estimates to modify the distance function. For each $e = (u,v)$, let[4] $\bar{c}(e) = c(e) + f(v) - f(u)$. We run Dijkstra's algorithm with the modified distance function. We know already that node potentials do not change shortest paths, and hence correctness is preserved. Tentative distances are related via $\bar{d}[v] = d[v] + f(v) - f(s)$. An alternative view of this modification is that we run Dijkstra's algorithm with the original distance function but remove the node with minimal value $d[v] + f(v)$ from the queue. The algorithm just described is known as *A^*-search*.

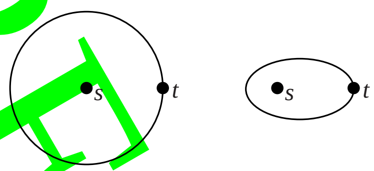


Fig. 10.11. The standard Dijkstra search grows a circular region centered on the source; goal-directed search grows a region protruding toward the target

Before we state requirements on the estimate f, let us see one specific example. Assume, in a thought experiment, that $f(v) = \mu(v,t)$. Then $\bar{c}(e) = c(e) + \mu(v,t) - \mu(u,t)$ and hence edges on a shortest path from s to t have a modified cost equal to zero and all other edges have a positive cost. Thus Dijkstra's algorithm only follows shortest paths, without looking left or right.

The function f must have certain properties to be useful. First, we want the modified distances to be nonnegative. So, we need $c(e) + f(v) \geq f(u)$ for all edges $e = (u,v)$. In other words, our estimate of the distance from u should be at most our estimate of the distance from v plus the cost of going from u to v. This property is called consistency of estimates. We also want to be able to stop the search when t is removed from the queue. This works if f is a lower bound on the distance to the target, i.e., $f(v) \leq \mu(v,t)$ for all $v \in V$. Then $f(t) = 0$. Consider the point in time when t is removed from the queue, and let p be any path from s to t. If all edges of p have been relaxed at termination, $d[t] \leq c(p)$. If not all edges of p have been relaxed at termination, there is a node v on p that is contained in the queue at termination. Then $d(t) + f(t) \leq d(v) + f(v)$, since t was removed from the queue but v was not, and hence

$$d[t] = d[t] + f(t) \leq d[v] + f(v) \leq d[v] + \mu(v,t) \leq c(p) .$$

In either case, we have $d[t] \leq c(p)$, and hence the shortest distance from s to t is known as soon as t is removed from the queue.

What is a good heuristic function for route planning in a road network? Route planners often give a choice between *shortest* and *fastest* connections. In the case

[4] In Sect. 10.7, we added the potential of the source and subtracted the potential of the target. We do exactly the opposite now. The reason for changing the sign convention is that in Lemma 10.10, we used $\mu(s,v)$ as the node potential. Now, f estimates $\mu(v,t)$.

of shortest paths, a feasible lower bound $f(v)$ is the straight-line distance between v and t. Speedups by a factor of roughly four are reported in the literature. For fastest paths, we may use the geometric distance divided by the speed assumed for the best kind of road. This estimate is extremely optimistic, since targets are frequently in the center of a town, and hence no good speedups have been reported. More sophisticated methods for computing lower bounds are known; we refer the reader to [77] for a thorough discussion.

10.8.2 Hierarchy

Road networks usually contain a hierarchy of roads: throughways, state roads, county roads, city roads, and so on. Average speeds are usually higher on roads of higher status, and therefore the fastest routes frequently follow the pattern that one starts on a road of low status, changes several times to roads of higher status, drives the largest fraction of the path on a road of high status, and finally changes down to lower-status roads near the target. A heuristic approach may therefore restrict the search to high-status roads except for suitably chosen neighborhoods of the source and target. Observe, however, that the choice of neighborhood is nonobvious, and that this heuristic sacrifices optimality. You may be able to think of an example from your driving experience where shortcuts over small roads are required even far away from the source and target. Exactness can be combined with the idea of hierarchies if the hierarchy is defined algorithmically and is not taken from the official classification of roads. We now outline one such approach [165], called *highway hierarchies*. It first defines a notion of locality, say anything within a distance of 10 km from either the source or the target. An edge $(u,v) \in E$ is a *highway edge* with respect to this notion of locality if there is a source node s and a target node t such that (u,v) is on the fastest path from s to t, v is not within the local search radius of s, and u is not within the local (backward) search radius of t. The resulting network is called the *highway network*. It usually has many vertices of degree two. Think of a fast road to which a slow road connects. The slow road is not used on any fastest path outside the local region of a nearby source or nearby target, and hence will not be in the highway network. Thus the intersection will have degree three in the original road network, but will have degree two in the highway network. Two edges joined by a degree-two node may be collapsed into a single edge. In this way, the *core* of the highway network is determined. Iterating this procedure of finding a highway network and contracting degree-two nodes leads to a hierarchy of roads. For example, in the case of the road networks of Europe and North America, a hierarchy of up to ten levels resulted. Route planning using the resulting highway hierarchy can be several thousand times faster than Dijkstra's algorithm.

10.8.3 Transit Node Routing

Using another observation from daily life, we can get even faster [15]. When you drive to somewhere "far away", you will leave your current location via one of only a few "important" traffic junctions. It turns out that in real-world road networks about

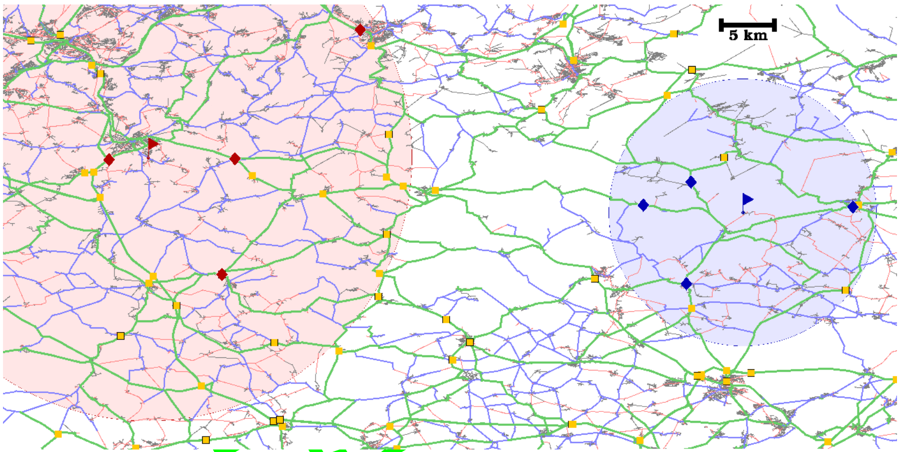


Fig. 10.12. Finding the optimal travel time between two points (the flags) somewhere between Saarbrücken and Karlsruhe amounts to retrieving the 2×4 *access nodes* (diamonds), performing 16 table lookups between all pairs of access nodes, and checking that the two disks defining the *locality filter* do not overlap. The small squares indicate further transit nodes

99% of all quickest paths go through about $O(\sqrt{n})$ important *transit nodes* that can be automatically selected, for example using highway hierarchies. Moreover, for each particular source or target node, all long-distance connections go through one of about ten of these transit nodes – the *access nodes*. During preprocessing, we compute a complete distance table between the transit nodes, and the distances from all nodes to their access nodes. Now, suppose we have a way to tell that a source s and a target t are sufficiently far apart.[5] There must then be access nodes a_s and a_t such that $\mu(t) = \mu(a_s) + \mu(a_s, a_t) + \mu(a_t, t)$. All these distances have been precomputed and there are only about ten candidates for a_s and for a_t, respectively, i.e., we need (only) about 100 accesses to the distance table. This can be more than 1 000 000 times faster than Dijkstra's algorithm. Local queries can be answered using some other technique that will profit from the closeness of the source and target. We can also cover local queries using additional precomputed tables with more local information. Figure 10.12 from [15] gives an example.

10.9 Implementation Notes

Shortest-path algorithms work over the set of extended reals $\mathbb{R} \cup \{+\infty, -\infty\}$. We may ignore $-\infty$, since it is needed only in the presence of negative cycles and, even there, it is needed only for the output; see Sect. 10.6. We can also get rid of $+\infty$ by noting that $parent(v) = \perp$ iff $d[v] = +\infty$, i.e., when $parent(v) = \perp$, we assume that $d[v] = +\infty$ and ignore the number stored in $d[v]$.

[5] We may need additional preprocessing to decide this.

A refined implementation of the Bellman–Ford algorithm [187, 131] explicitly maintains a current approximation T of the shortest-path tree. Nodes still to be scanned in the current iteration of the main loop are stored in a set Q. Consider the relaxation of an edge $e=(u,v)$ that reduces $d[v]$. All descendants of v in T will subsequently receive a new d-value. Hence, there is no reason to scan these nodes with their current d-values and one may remove them from Q and T. Furthermore, negative cycles can be detected by checking whether v is an ancestor of u in T.

10.9.1 C++

LEDA [118] has a special priority queue class *node_pq* that implements priority queues of graph nodes. Both LEDA and the Boost graph library [27] have implementations of the Dijkstra and Bellman–Ford algorithms and of the algorithms for acyclic graphs and the all-pairs problem. There is a graph iterator based on Dijkstra's algorithm that allows more flexible control of the search process. For example, one can use it to search until a given set of target nodes has been found. LEDA also provides a function that verifies the correctness of distance functions (see Exercise 10.8).

10.9.2 Java

JDSL [78] provides Dijkstra's algorithm for integer edge costs. This implementation allows detailed control over the search similarly to the graph iterators of LEDA and Boost.

10.10 Historical Notes and Further Findings

Dijkstra [56], Bellman [18], and Ford [63] found their algorithms in the 1950s. The original version of Dijkstra's algorithm had a running time $\mathrm{O}\left(m+n^2\right)$ and there is a long history of improvements. Most of these improvements result from better data structures for priority queues. We have discussed binary heaps [208], Fibonacci heaps [68], bucket heaps [52], and radix heaps [9]. Experimental comparisons can be found in [40, 131]. For integer keys, radix heaps are not the end of the story. The best theoretical result is $\mathrm{O}(m+n\log\log n)$ time [194]. Interestingly, for *undirected* graphs, linear time can be achieved [190]. The latter algorithm still scans nodes one after the other, but not in the same order as in Dijkstra's algorithm.

Meyer [139] gave the first shortest-path algorithm with linear average-case running time. The algorithm ALD was found by Goldberg [76]. For graphs with bounded degree, the Δ-stepping algorithm [140] is even simpler. This uses bucket queues and also yields a good parallel algorithm for graphs with bounded degree and small diameter.

Integrality of edge costs is also of use when negative edge costs are allowed. If all edge costs are integers greater than $-N$, a *scaling algorithm* achieves a time $\mathrm{O}(m\sqrt{n}\log N)$ [75].

In Sect. 10.8, we outlined a small number of speedup techniques for route planning. Many other techniques exist. In particular, we have not done justice to advanced goal-directed techniques, combinations of different techniques, etc. Recent overviews can be found in [166, 173]. Theoretical performance guarantees beyond Dijkstra's algorithm are more difficult to achieve. Positive results exist for special families of graphs such as planar graphs and when approximate shortest paths suffice [60, 195, 192].

There is a generalization of the shortest-path problem that considers several cost functions at once. For example, your grandfather might want to know the fastest route for visiting you but he only wants routes where he does not need to refuel his car, or you may want to know the fastest route subject to the condition that the road toll does not exceed a certain limit. Constrained shortest-path problems are discussed in [86, 135].

Shortest paths can also be computed in geometric settings. In particular, there is an interesting connection to optics. Different materials have different refractive indices, which are related to the speed of light in the material. Astonishingly, the laws of optics dictate that a ray of light always travels along a shortest path.

Exercise 10.21. An ordered semigroup is a set S together with an associative and commutative operation $+$, a neutral element 0, and a linear ordering $\leq$ such that for all x, y, and z, $x \leq y$ implies $x+z \leq y+z$. Which of the algorithms of this chapter work when the edge weights are from an ordered semigroup? Which of them work under the additional assumption that $0 \leq x$ for all x?