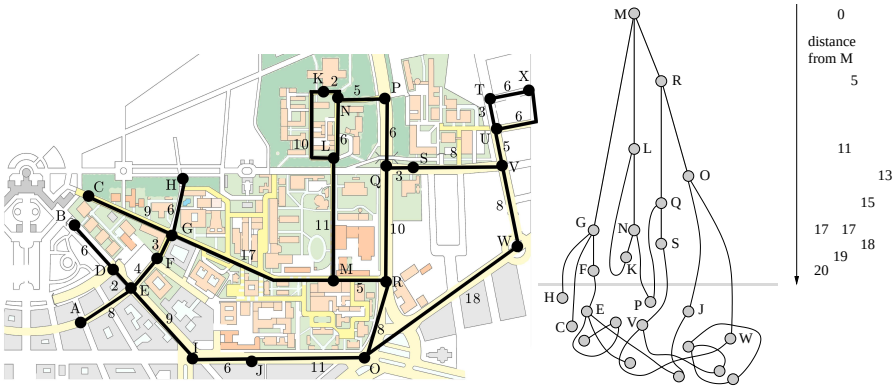
10

Shortest Paths



The problem of finding the shortest, quickest or cheapest path between two locations 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 all-destinations problem. Sometimes we may even want a complete distance table from everywhere to everywhere – the all-pairs problem. In an old fashioned 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 (all roads are assumed to be two-way roads) 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 illustration above shows a map of the campus of the Karlsruhe Institute of Technology[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. Many other problems profit from shortest-path computations, once an appropriate graph model is defined. For example, Ahuja et al. [9] mention such diverse applications as planning flows in networks, planning 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.

[1] © KIT, Institut für Photogrammetrie und Fernerkundung.

The most general formulation of the shortest-path problem considers a directed graph $G = (V, E)$ and a cost function c that maps edges to arbitrary real numbers. It is fairly expensive to solve and many of the applications mentioned above do not need the full generality. For example, roads always have positive length. 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 on 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 based on the worst-case analysis. 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. In Sect. 10.8, we go back to our original question about a shortest path between two specific nodes, in particular, in the context of road networks. Finally, we discuss parallel shortest path algorithms in Sect. 10.9.

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., the cost of the path $p = \langle e_1, e_2, \ldots, e_k \rangle$ is equal to $c(p) = \sum_{1 \le i \le k} c(e_i)$. The empty path has cost 0.

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, assume 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., $c(C) < 0$, there is no shortest path from s to v, since the set $\{c(p) + c(q) - i \cdot |c(C)| : i \geq 0\}$ contains numbers smaller than any fixed number. We shall show next that shortest paths exist if there are no negative cycles.

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 p_0 be a shortest *simple* path from s to v. Note that there are only finitely many simple paths from s to v. If p_0 is not a shortest path from s to v, there is a shorter path r from s to v. Then r is nonsimple. 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 $c(p_0) \leq c(pq)$ and $c(pq) + c(C) = c(r) < c(p_0) \leq c(pq)$. So $c(C) < 0$ and we have shown the existence of a negative cycle, a contradiction to the assumption of the lemma. Thus there can be no path shorter than p_0. □

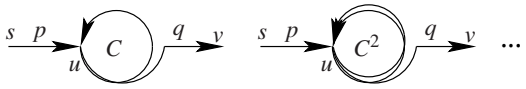


Fig. 10.1. A nonsimple path pCq from s to v.

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 if and only if 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*, or simply *distance*, $\mu(s,v)$ from s to v as

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

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 necessarily unique.

Exercise 10.4 (alternative definition). Show that

$$\mu(s,v) = \inf\{c(p) : p \text{ is a path from } s \text{ to } v\},$$

where the infimum of the empty set is $+\infty$.

Our strategy for finding shortest paths from a source node s is a generalization of the BFS algorithm shown in Fig. 9.2. 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 refer to $d[v]$ as the *tentative distance* of v. Initially, $d[s] = 0$ and *parent*$[s] = s$. All other nodes have tentative distance "infinity" 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** $d[v] := d[u] + c(e)$; $\quad parent[v] := u$

Arithmetic with ∞ is done in the natural way: $a < \infty$ and $\infty + a = \infty$ for all reals a, and $\infty \not< \infty$.

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 0 from s to s, and all other nodes have infinite distance. Consider next a relaxation of an edge $e = (u,v)$. If the relaxation does not change $d[v]$, there is nothing to show. Otherwise, $d[u] + c(e) < d[v]$ and hence $d[u] < \infty$. By the induction hypothesis, there is a path p of length $d[u]$ from s to u. Then 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 has been discovered. For example, the (reversed) thick (solid and dashed) 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 reachable from s so that their shortest-path cost is $-\infty$. Node a is attached to this cycle, and

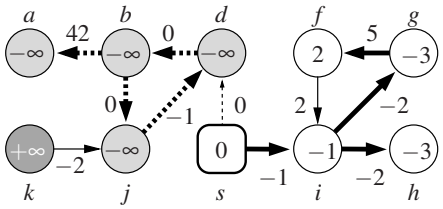


Fig. 10.2. A graph with source node s and shortest-path distances $\mu(v)$. Edge costs are shown as edge labels, and the distances are shown inside the nodes. The thick solid edges indicate shortest paths starting in s. The dashed edges and the light gray nodes belong to an infinite collection of paths with arbitrary small negative costs. The thick dashed edges (d,b), (b,j), and (j,d) form a negative cycle. The reversals of the thick solid and dashed edges indicate the parent function after a sufficient number of iterations. Node k is not reachable from s.

thus $\mu(a) = -\infty$. The edge (a,d) proves that d and hence d, b, j, and a are reachable from s, but this is not recorded in the parent information.

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:

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

We have $\mu(v) = \sum_{1 \leq j \leq k} c(e_j)$. 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 $d[v_i] \leq \sum_{1 \leq j \leq i} c(e_j)$ after time t_i, as a simple induction shows. This is clear for $i = 0$, since $d[s]$ is initialized to 0 and d-values are only decreased. After the relaxation of $e_i = R[t_i]$ for $i > 0$, we have $d[v_i] \leq d[v_{i-1}] + c(e_i) \leq \sum_{1 \leq j \leq i} c(e_j)$. Thus, after time t_k, we have $d[v] \leq \mu(v)$. 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 $1 \leq$

$i \le k$. So at some point in time, some operation $relax(u, v_i)$ sets $d[v_i]$ to $\mu(v_i)$ and $parent[v_i]$ to u. Note that this point of time may be before time t_i; it cannot be after t_i. By the proof of Lemma 10.2, there is a path from s to v_i of length $\mu(v_i)$ ending in the edge (u, v_i). Since, by assumption, the shortest path from s to v_i is unique, we must have $u = v_{i-1}$. So the *relax* operation sets $parent[v_i]$ to v_{i-1}. Later relax operations do not change this value since $d[v_i]$ is not decreased further. □

Exercise 10.5. Redo the second paragraph in the proof above, but without the assumption that shortest paths are unique.

Exercise 10.6. 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 $\mu(v) \neq -\infty$ 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. 9.3.1 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 depth-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. One may even ignore edges that emanate from nodes before s in the topological order. Since every edge relaxation takes constant time, we obtain a total execution time of $O(m+n)$.

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

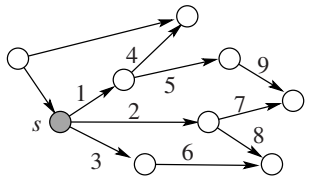


Fig. 10.3. Order of edge relaxations for the computation of the shortest paths from node s in a DAG. The topological order of the nodes is given by their x-coordinates. There is no need to relax the edges out of nodes "to the left of" s.

Exercise 10.7 (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? (d) Suppose there are minimum connection times at some of the stops. How can you incorporate them into the model? (e) How do you find the quickest connection with at most two intermediate stops?

Exercise 10.5. Instantiate the parallel DAG processing framework presented in Sect. 9.4 to compute shortest paths in DAGs.

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, at least in the case where all edge costs are positive. 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 smallest tentative distance, the true and tentative distances agree. We shall prove this in Theorem 10.6. 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.

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 smallest tentative distance
  relax all edges (u,v) out of u and declare u scanned

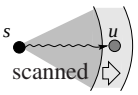


Fig. 10.4. Dijkstra's shortest-path algorithm for nonnegative edge weights

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$\rightsquigarrow(c,7)$	$\langle\rangle$

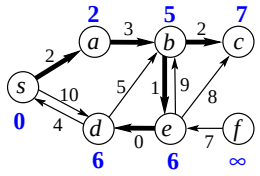


Fig. 10.5. Example run of Dijkstra's algorithm on the graph given on the *right*. The bold edges form the shortest-path tree, and the numbers in bold indicate shortest-path distances. The table on the *left* illustrates the execution. The *queue* contains all pairs $(v,d[v])$ with v reached and unscanned. 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.

Note that Dijkstra's algorithm, when applied to undirected graphs 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 10.6. *Dijkstra's algorithm solves the single-source shortest-path problem for graphs with nonnegative edge costs.*

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 path $p = \langle s = v_1, v_2, \ldots, v_k = v\rangle$ from s to v, and let i be smallest 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} is scanned. When v_{i-1} is scanned, the edge (v_{i-1}, v_i) is relaxed. After this operation, we have $d[v_i] \leq d[v_{i-1}] + c(v_{i-1}, v_i) < \infty$. 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 at which a node v is scanned whose tentative distance $d[v]$ at time t is larger than its true distance $\mu[v]$. Consider a shortest path $p = \langle s = v_1, v_2, \ldots, v_k = v\rangle$ from s to v, and let i be smallest 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, v_{i-1} is scanned before time

t. Hence $d[v_{i-1}] \leq \mu(v_{i-1})$ when v_{i-1} is scanned. By Lemma 10.2, we then even have $d[v_{i-1}] = \mu(v_{i-1})$, and $d[v_{i-1}]$ does not change anymore afterwards. After v_{i-1} has been scanned, we always have $d[v_i] \leq d[v_{i-1}] + c(v_{i-1}, v_i) \leq \mu(v_{i-1}) + c(v_{i-1}, v_i)$, even if $d[v_i]$ should decrease. The last expression is at most $c(\langle v_1, v_2, \ldots, v_i \rangle)$, which in turn is at most $c(p)$, since all edge costs are nonnegative. Since $c(p) = \mu(v) < d[v]$, it follows that $d[v_i] < d[v]$. Hence at time t the algorithm scans some node different from v, a contradiction. □

Exercise 10.8. 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 nondecreasing shortest-path distance.

Exercise 10.9 (checking of shortest-path distances (positive edge costs)). 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 0?

Exercise 10.10 (checking of shortest-path distances (nonnegative edge costs)). Assume that all edge costs are non-negative, that all nodes are reachable from s, that *parent* encodes a tree rooted at s (for each node v, *parent*(v) is the parent of v in this tree), and that d is a node array of nonnegative reals satisfying $d[s] = 0$ and $d[v] = d[\mathit{parent}(v)] + c(\mathit{parent}(v), v) = \min_{(u,v)\in E} d[u] + c(u,v)$ for $v \neq s$. Show that $d[v] = \mu(v)$ for all v.

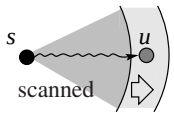
We now turn to the implementation of Dijkstra's algorithm. We store all unscanned reached nodes in an addressable priority queue (see Sect. 6.2) using their tentative-distance values as keys. Thus, we can extract the next node to be scanned using the priority queue operation *deleteMin*. We need a variant 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. If the items of the priority queue are objects, we may store them 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

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

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 $|Q| = O(n)$.

```
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, d[v])
                else Q.insert(v)
    return (d, parent)
```



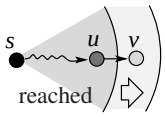


Fig. 10.6. Pseudocode for Dijkstra's algorithm

Exercise 10.11. Assume $n-1 \le m \le n(n-1)/2$. 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 unreached or reached and unscanned or scanned. Then *insert* and *decreaseKey* take time $O(1)$. A *deleteMin* takes time $O(n)$, 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

$$T_{Dijkstra59} = O\bigl(m+n^2\bigr)\,.$$

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

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

and

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

respectively. Asymptotically, the Fibonacci heap implementation is superior except for sparse graphs with $m = O(n)$. 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 $O(n \log(m/n))$.

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 multiset $C(v)$ 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 the $\prod_{v \in V} indegree(v)!$ possible assignments of edge costs to edges. Each such assignment has the same probability. 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 10.7. *Under the assumptions above, the expected number of decreaseKey operations is* $O(n \log(m/n))$.

Proof. We present a proof due to Noshita [244].

We need to start with a technical remark before we can enter the proof proper. For the analysis of the worst-case running time, it is irrelevant how nodes with the same tentative distance are treated by the priority queue. Any node with smallest tentative distance may be returned by the *deleteMin* operation. Indeed, the specification of the *deleteMin* operation leaves it open which element of smallest key is returned. For this proof, we need to be more specific and need to assume that nodes with equal tentative distance are returned in some *consistent* order. The detailed requirements will become clear in the proof. Consistency can, for example, be obtained by using the keys (tentative distance, node name) under lexicographic ordering instead of simply the tentative distances. Then nodes with the same tentative distance are removed in increasing order of node name.

Consider a particular node $v \neq s$ and assume that the costs of all edges not ending in v are already assigned. Only the assignment of the edge costs in $C(v)$ is open and will be determined randomly. We shall show that the expected number of $decreaseKey(v, *)$ operations is bounded by $\ln(indegree(v))$. We do so by relating the number of *decreaseKey* operations to the number of left-to-right maxima in a random sequence of length $indegree(v)$ (see Sect. 2.11).

The main difficulty in the proof is dealing with the fact that the order in which the edges into v are relaxed may depend on the assignment of the edge costs to these edges. The crucial observation is that up to the time when v is scanned, this order is fixed. Once v is scanned, the order may change. However, no further *decreaseKey* operations occur once v is scanned. In order to formalize this observation, we consider the execution of Dijkstra's algorithm on $G \setminus v$ (G without v and all edges incident to

v) and on G with the same assignment of costs to the edges not incident to v and an arbitrary assignment of costs to the edges incident to v. Before v is scanned in the run on G, exactly the same nodes are scanned in both executions and these scans are in the same order. This holds because the tentative distance of v has no influence on the other nodes before v is scanned. Also, the presence of v in the priority queue has no influence on the results of *deleteMin* operations before v is scanned in the run on G. This property is the consistency requirement. Of course, the time when v is scanned depends on the assignment of edge costs to the edges into v.

Let $u_1, \ldots, u_k$, where $k = \mathit{indegree}(v)$, be the source nodes of the edges into v in the order in which they are scanned in a run of Dijkstra's algorithm on $G \setminus v$ and let $\mu'(u_i)$ be the distance from s to u_i in $G \setminus v$. Nodes u_i that cannot be reached from s in $G \setminus v$ have infinite distance and come at the end of this ordering. According to Exercise 10.8, we have $\mu(u_1) \leq \mu(u_2) \leq \ldots \leq \mu(u_k)$. In the run on G, the edges $e_1 = (u_1, v)$, $e_2 = (u_2, v)$, $\ldots$, $e_\ell = (u_\ell, v)$ are relaxed in that order until v is scanned; ℓ is a random variable. We do not know in what order the remaining edges into v are relaxed. However, none of them leads to a *decreaseKey*$(v, *)$ operation. We have $\mu'(u_i) = \mu(u_i)$ for $1 \leq i \leq \ell$. If e_i causes a *decreaseKey* operation, then $1 < i \leq \ell$ (the relaxation of e_1 causes an *insert*(v) operation) and

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

Since $\mu(u_j) \leq \mu(u_i)$, 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. Left-to-right minima are defined analogously to left-to-right maxima; see Sect. 2.11. 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 1; 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. 2.11, 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 $\ln k = \ln \mathit{indegree}(v)$.

By the linearity of expectations, we may sum this bound over all nodes to obtain the following bound for the expected number of *decreaseKey* operations:

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

where the last inequality follows from the concavity of the ln function (see (A.16)). □

We conclude that the expected running time is $O(m + n \log(m/n) \log n)$ 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 10.12. Show that $E[T_{\mathit{DijkstraBHeap}}] = O(m)$ if $m = \Omega(n \log n \log\log n)$.

10.5 Monotone Integer Priority Queues

Dijkstra's algorithm scans nodes in order of nondecreasing distance values. Hence, a monotone priority queue (see Chap. 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, which we assume 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 (0 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 3.12). 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* tentative 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

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

***Exercise 10.13 (Dinic's refinement of bucket queues [?]).** 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 [10] improve on the bucket queue implementation by using buckets of different widths. Narrow buckets are used for tentative distances close to *min*, and

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

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

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 $a = \sum_{i \geq 0} \alpha_i 2^i$ and $b = \sum_{i \geq 0} \beta_i 2^i$, we define the most significant distinguishing index $msd(a,b)$ as the largest i with $\alpha_i \neq \beta_i$, and let it be -1 if $a = b$. If $a < b$, then a has a 0-bit in position $i = msd(a,b)$ and b has a 1-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:

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 1-bit in position $i \in 0..K-1$, the corresponding bucket $B[i]$ is empty. This holds since any $d[v]$ with $i = msd(min, d[v])$ would have a 0-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 $a \neq b$, the bitwise exclusive OR $a \oplus b$ of a and b has its most significant 1 in position i and hence represents an integer whose value is at least 2^i and less than 2^{i+1}. Thus $msd(a,b) =$

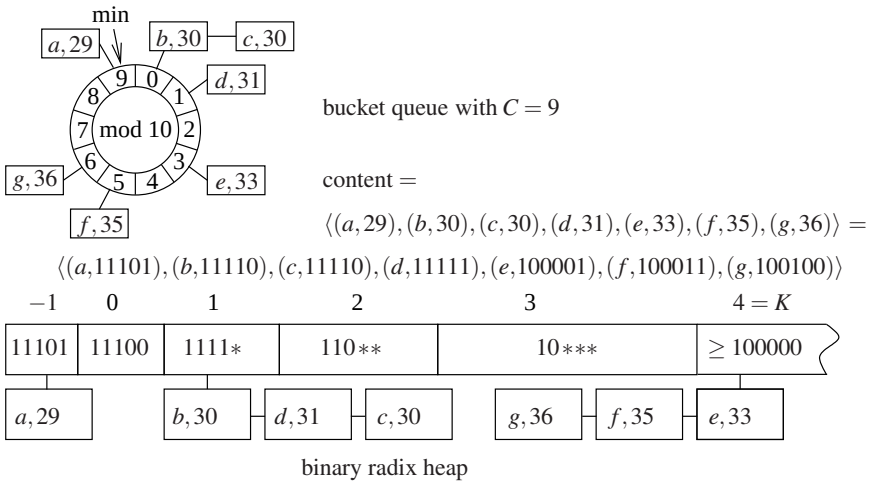


Fig. 10.7. Example of a bucket queue (*upper part*) and a radix heap (*lower part*). Since $C = 9$, we have $K = 1 + \lfloor \log C \rfloor = 4$. The bit patterns in the buckets of the radix heap indicate the set of keys they can accommodate.

$\lfloor \log(a \oplus b) \rfloor$, 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.[2] Alternatively, we can use lookup tables or yet other solutions. From now on, we shall assume that *msd* can be evaluated in constant time.

Exercise 10.14. There is another way to describe the distribution of nodes over buckets. Let $min = \sum_j \mu_j 2^j$, let k be the smallest index greater than K with $\mu_k = 0$, and let $M_i = \sum_{j>i} \mu_j 2^j$. B_{-1} contains all nodes $v \in Q$ with $d[v] = min$, for $0 \leq i < K$, $B_i = \emptyset$ if $\mu_i = 1$, and $B_i = \left\{ v \in Q : M_i + 2^i \leq d[x] < M_i + 2^{i+1} - 1 \right\}$ if $\mu_i = 0$, and $B_K = \left\{ v \in Q : M_k + 2^k \leq d[x] \right\}$. Prove that this description is correct.

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 $i < j \leq K$, any key a in bucket $B[j]$ 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 10.8 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 the *insert*(v) operation 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 10.8. *Let i be the smallest index 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 $x \in B[i]$.*

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

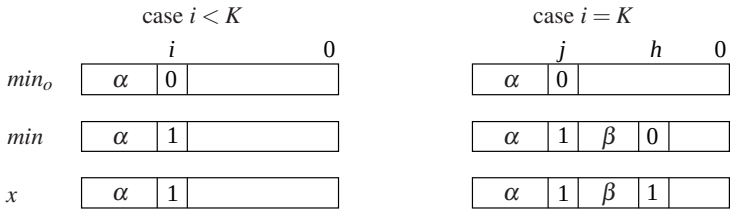


Fig. 10.8. The structure of the keys relevant to the proof of Lemma 10.8. In the proof, it is shown that β starts with $j-K$ 0's.

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 0 in bit position i, and all $x \in B[i]$ have a 1 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 $j = msd(min_o, min)$. Since $min \in B[K]$, we have $j \geq K$. 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 0. The binary representation of A is α followed by $j+1$. Since the jth bit of min_o is 0 and that of min is 1 (j is the most significant distinguishing index and $min_0 < min$), we have $min_o < A + 2^j \leq min$. 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 $j-K$ many 0's, 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 1. 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 jth bit of x would be 0 and hence $x < min$. Since min_o has a 0 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 $j-K$ many 0's, followed by some K-bit string. Since $min \leq x$, the $j-K$ many 0's must also be present in the binary representation of min. □

***Exercise 10.15.** 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 ith 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 $O(m+n(b+\log C/\log b))$ 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 [129, 226]. 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 is known to be equal to their exact distance from s (we shall see below how one can learn this), 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 10.9. *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 $O(m+n+\log C)$.*

Proof. We still need to argue the bound on the running time. To do this, we modify the amortized analysis of plain radix heaps. Initialization of the heap takes time $O(K) = O(\log C)$. Consider now an arbitrary node v and how it moves through the buckets. It starts out in some bucket $B[j]$ with $j \leq K$. When it has just been moved to a new bucket but not yet to F, $d[v] \geq min + c^{\text{in}}_{\min}(v) + 1$, and hence the index i of the new bucket satisfies $i \geq \lfloor \log(c^{\text{in}}_{\min}(v)+1) \rfloor + 1$. Therefore, in order to pay for all moves of node v between buckets, it suffices to charge $K - (\lfloor \log(c^{\text{in}}_{\min}(v)+1) \rfloor + 1) + 1 = K - \lfloor \log(c^{\text{in}}_{\min}(v)+1) \rfloor$ tokens to v. Summing over all nodes, we obtain a total payment of

$$\sum_v \left(K - \left\lfloor \log(c^{\text{in}}_{\min}(v)+1) \right\rfloor \right) = n + \sum_v \left(K - \left\lceil \log(c^{\text{in}}_{\min}(v)+1) \right\rceil \right).$$

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.,

$$\sum_v \left(K - \left\lceil \log(c^{\text{in}}_{\min}(v)+1) \right\rceil \right) \leq \sum_e (K - \lceil \log(c(e)+1) \rceil).$$

Now, $K - \lceil \log(c(e)+1) \rceil$ is the number of leading 0's 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

$$\text{prob}(K - \lceil \log(c(e)+1) \rceil \geq k) \leq \frac{|0..2^{K-k}-1|}{C+1} = \frac{2^{K-k}}{C+1} \leq 2^{-(k-1)}.$$

The last inequality follows from $C \geq 2^{K-1}$. Using (A.2) and (A.15), we obtain for each edge e

$$\mathrm{E}[K - \lceil \log(c(e)+1) \rceil] = \sum_{k \geq 1} \text{prob}(K - \lceil \log(c(e)+1) \rceil \geq k) \leq \sum_{k \geq 1} 2^{-(k-1)} = 2.$$

By the linearity of expectations, we obtain further

$$\mathrm{E}\left[\sum_e (K - \lceil \log(c(e)+1) \rceil)\right] = \sum_e \mathrm{E}[K - \lceil \log(c(e)+1) \rceil] \leq \sum_e 2 = 2m = \mathrm{O}(m).$$

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

Observe that the preceding proof does not require edge costs to be independent. It suffices that the cost of each edge is chosen uniformly at random in $0..C$. Theorem 10.9 can be extended to real-valued edge costs.

****Exercise 10.16.** 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 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 10.13. 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 [38, 108] 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 w with $\mu(w) = -\infty$. The following lemma characterizes these nodes.

```
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

Lemma 10.10. *After $n-1$ rounds of edge relaxations, we have $\mu(w) = -\infty$ if and only if there is an edge $e = (u,v)$ with $d[u] + c(e) < d[v]$ such that w is reachable from v.*

Proof. If $d[u] + c(e) < d[v]$, then $d[u] < \infty$ and hence u and v are reachable from s. A further relaxation of e would further decrease $d[v]$ and hence $\mu(v) < d[v]$. Since $\mu(v) = d[v]$ also for all nodes v with $\mu(v) > -\infty$, we must have $\mu(v) = -\infty$. Then $\mu(w) = -\infty$ for any node reachable from v.

Assume conversely that $\mu(w) = -\infty$. Then there is a cycle $C = \langle v_0, v_1, \ldots, v_k \rangle$ with $v_k = v_0$ of negative cost that is reachable from s and from which w can be reached. Since C can be reached from s, we have $d[v_i] < \infty$ for all i. We claim that there must be at least one i, with $d[v_i] + c(v_i, v_{i+1}) < d[v_{i+1}]$. Assume otherwise, i.e., $d[v_i] + c(v_i, v_{i+1}) \geq d[v_{i+1}]$ for $0 \leq i < k$. Summing over all i yields $\sum_{0 \leq i < k} d[v_i] + \sum_{0 \leq i < k} c(v_i, v_{i+1}) \geq \sum_{0 \leq i < k} d[v_{i+1}]$. Since $v_0 = v_k$, the two summations over tentative distances are equal and we conclude that $c(C) = \sum_{0 \leq i < k} c(v_i, v_{i+1}) \geq 0$, a contradiction to the fact that C is a cycle of negative cost. □

The pseudocode implements the lemma using a recursive function *infect*(v). For any edge $e = (u,v)$ with $d[u] + c(e) < d[v]$, 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.

Exercise 10.17. Show that the postprocessing runs in time $O(m)$. Hint: Relate *infect* to *DFS*.

Exercise 10.18. 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 $\mu(v) = -\infty$.

Exercise 10.19 (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 10.11 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 $O(n^2m)$. 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 $O(nm + n(m + n\log n)) = O(nm + n^2 \log n)$. 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 10.11. *Let p and q be paths from v to w. Then $\bar{c}(p) = pot(v) + c(p) - pot(w)$ and $\bar{c}(p) \leq \bar{c}(q)$ if and only if $c(p) \leq c(q)$. In particular, the shortest paths with respect to $\bar{c}$ are the same as those with respect to c.*

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

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

□

Exercise 10.20. 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 10.12. *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.*

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 $O(m)$
compute $\mu(v)$ for all v with Bellman–Ford // time $O(nm)$
set $pot(v) = \mu(v)$ and compute reduced costs $\bar{c}(e)$ for $e \in E$ // time $O(m)$
forall nodes x **do** // time $O(n(m+n\log n))$
    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 $O(m)$
forall $e = (v,w) \in V \times V$ **do** $\mu(v,w) := \bar{\mu}(v,w) + pot(w) - pot(v)$ // use Lemma 10.11

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

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 $\mu(v) + c(e) \geq \mu(w)$, and thus $\bar{c}(e) = \mu(v) + c(e) - \mu(w) \geq 0$. □

Theorem 10.13. *The all-pairs shortest-path problem for a graph without negative cycles can be solved in time* $O\big(nm + n^2 \log n\big)$.

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 $O(nm)$, and the n shortest-path calculations take time $O(n(m+n\log n))$. The preprocessing and postprocessing take linear time $O(n+m)$. □

The assumption that G has no negative cycles can be removed [219].

Exercise 10.21. 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 $D'(s) := \max_{u \in V} \mu(s,u)$ be the maximum distance from s to any node u. Show that $D'(s) \leq D \leq 2D'(s)$ for undirected graphs. Also, show that no such relation holds for directed graphs. Let $D''(s) := \max_{u \in V} \mu(u,s)$ be the maximum distance from any node u to 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 shortest-path computations for source nodes s as well as target node s? (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. Edge costs are assumed to be nonnegative in this section.

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, assuming that every target node is equally likely, early stopping saves a factor of two in scanned nodes. 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 of 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 10.22. 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 distances 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 the largest index 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

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

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 disk centered at the source node. The disk is defined by the shortest-path metric in the graph. In the route-planning application for a road network, we may also consider geometric disks centered on the source and argue that shortest-path disks and geometric disks are about the same. We can then estimate the speedup obtained by bidirectional search using the following heuristic argument: A disk of a certain diameter has twice the area of two disks of half the diameter. We could thus hope that bidirectional search will save a factor of two compared with unidirectional search.

Exercise 10.23 (bidirectional search). (a) Consider bidirectional search in a grid graph with unit edge weights. How much does it save compared with unidirectional search? (b) Give an example where bidirectional search in a real road network takes *longer* than unidirectional search. Hint: Consider a densely inhabited city with sparsely populated surroundings. (c) Design a strategy for switching between forward (starting in s) and backward (starting in t) search such that bidirectional search will *never* inspect more than twice as many nodes as unidirectional search. (*d) 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.

We shall next describe several techniques that are more complex and less generally applicable. However, if they are applicable, they usually result in larger savings. These techniques mimic human behavior in route planning. The most effective variants of these speedup techniques are based on storing *preprocessed* information that depends on the graph but not on the source and target nodes. Note that with extreme preprocessing (compute the complete distance table using an all-pairs shortest-path computation), queries can answered very fast (time $O(\text{path length})$). The drawback is that the distance table needs space $\Theta(n^2)$. Running Dijkstra's algorithm for every query needs no extra space, but is slow. We shall discuss compromises between these extremes.

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 towards 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[3] $\bar{c}(e) = c(e) + f(v) - f(u)$. We run Dijkstra's algorithm with the modified distance function (assuming for the moment that it is nonnegative). We know

[3] 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 10.12, we used $\mu(s,v)$ as the node potential. Now, f estimates $\mu(v,t)$.

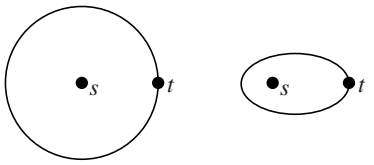


Fig. 10.11. The standard Dijkstra search grows a circular region centered on the source; goal-directed search grows a region protruding towards the target.

already (Lemma 10.11) 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 smallest value $d[v] + f(v)$ from the queue. The algorithm just described is known as *A^*-search*. In this context, f is frequently called the *heuristic* or the *heuristic distance estimate*.

Before we state requirements on the estimate f, let us see one specific example that illustrates the potential usefulness of heuristic information. 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 0. Thus any node ever removed from the queue has a modified tentative distance equal to 0. Consider any node that does not lie on a shortest path from s to t. Any shortest path from s to such a node must contain an edge $e = (u,v)$ such that u lies on a shortest path from s to t but e does not. Then $\mu(v,t) + c(e) > \mu(u,t)$ and hence $\bar{c}(e) > 0$. Thus nodes not on a shortest path from s to t have positive tentative distance, and hence Dijkstra's algorithm followsonly 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*. It has an interesting consequence. Consider any path $p = \langle v = v_0, v_1, \ldots, v_k = t \rangle$ from a node v to t. Adding the consistency relation $f(v_i) + c((v_{i-1}, v_i)) \geq f(v_{i-1})$ for all edges of the paths yields $f(t) + c(p) \geq f(v)$. Thus $c(p) \geq f(v) - f(t)$ and, further, $\mu(v,t) \geq f(v) - f(t)$. We may assume without loss of generality that $f(t) = 0$; otherwise, we simply subtract $f(t)$ from all f-values. Then $f(v)$ is a lower bound for $\mu(v,t)$.

It is still true that we can stop the search when t is removed from the queue? 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 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 which are based on preprocessing; we refer the reader to [130] for a thorough discussion.

10.8.2 Hierarchy

Road networks are structured into a hierarchy of roads, for example residential roads, urban roads, highways of different categories, and motorways. Roads of higher status typically support higher average speeds and are therefore to be preferred for a long-distance journey. A typical fastest path for a long distance journey first changes to roads of higher and higher status and then travels on a road of highest status for most of the journey until it descends again towards the target. Early industrial route planners used this observation as a heuristic acceleration technique. However, this approach sacrifices optimality. The second author frequently drives from Saarbrücken to Bonn. The fastest route uses the motorway for most of the journey but descends to a road of lower status for about 10 kilometers near the middle of the route. The reason is that one has to switch motorways and the two legs of motorway form an acute angle. It is then better to take a shortcut on a route of lesser status in the official hierarchy of German roads. Thus, for fastest-path planning, the shortcut road should be in the top-level of the hierarchy. Such misclassification can be fixed manually. However, manually classifying the importance of roads is expensive and error-prone.

An algorithmic classification is called for. Several algorithmic approaches have been developed in the last two decades which have not only achieved better and better performance but also, eventually, have become simpler and simpler. We refer our readers to [30] for an overview. We discuss two approaches, an early one and a recent one.

The early approach [275] is quite intuitive. We call a road *level one* if there is a shortest path between some source and some target that includes that road outside (!!!) the initial and final segments of, say, 10 kilometers. The level-one roads form the level-one network. The level-one network will contain vertices of degree 2. For example, if a slow road forms a three-way junction with a fast road, it is likely that the two legs of the fast road belong to the level-one network, but the slow road does not. Nodes of degree 2 may be removed by replacing the two incident edges by a single edge passing over that node of degree 2. Once the simplified level-one network is constructed, the same strategy is used to form the level-two network. And so on.

We next describe *contraction hierarchies* (CHs) [123], which are less intuitive, but very effective (up to four orders of magnitude faster than Dijkstra's algorithm)

and yet fairly easy to implement (a basic implementation is possible with a few hundred lines of code).

Suppose for now that the nodes are already ordered by increasing importance. We describe below how this is done. The idea is that nodes that lie on many shortest paths are late in the ordering and nodes that are intermediate nodes of only a few shortest paths are early in the ordering. CH preprocessing goes through $n-1$ steps of *node contraction* – removing one node after another until only the most important node is left. CHs maintain the invariant that the shortest-path distances between the remaining nodes are preserved in each step. This is achieved by inserting *shortcut edges*. Suppose node v is to be removed but the path $\langle u,v,w\rangle$ is a shortest path. Then inserting a shortcut edge (u,w) with weight $c(u,v)+c(v,w)$ preserves the shortest-path distance between u and w and hence all shortest-path distances. Deciding whether $\langle u,v,w\rangle$ is a shortest path amounts to a local shortest-path search from u, looking for *witness* paths showing that the shortcut (u,w) is not needed. We can also speed up preprocessing by stopping the witness search early – if in doubt, simply insert a shortcut. For the removal of v, it suffices to inspect all pairs of edges entering and leaving v and to insert a shortcut if they form a shortest path. CHs are effective on road networks since, with a good node ordering, few shortcuts are needed.

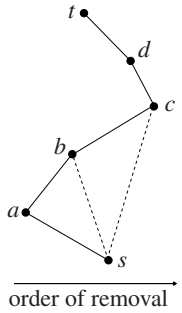


Fig. 10.12. The path $\langle s,a,b,c,d,t\rangle$ is a shortest path from s to t. The nodes are removed in the order a, b, t, s, d, c. When a is removed, the shortcut (s,b) is added. When b is removed, the shortcut (s,c) is added. When s and t are removed, no shortcut edges relevant to the shortest path from s to t are added. When d is removed, there is no shortcut edge (c,t) added since t is no longer part of the graph at this point in time. All solid and dashed edges are part of the graph H. The path $\langle s,c,d,t\rangle$ is a shortest path from s to t in H. It consists of the up-path $\langle s,c\rangle$ followed by the down-path $\langle c,d,t\rangle$. Expansion of $\langle s,c\rangle$ first yields $\langle s,b,c\rangle$ and then $\langle s,a,b,c\rangle$.

The result of preprocessing is the original graph plus the shortcuts inserted during contraction. We call this graph H; see Fig. 10.12 for an example. The crucial property of H is that, for arbitrary s and t, it contains a shortest path P from s to t that is an *up–down path*. We call an edge (u,v) *upward* if u is removed before v and *downward* otherwise. An up–down path consists of a first segment containing only upward edges and a second segment containing only downward edges. The existence of such a path is easily seen. Let $\langle\ldots,u,v,w,\ldots\rangle$ be a shortest path in H such that v was removed before u and w. Then the short-cut (u,w) was introduced when v was removed and $\langle\ldots,u,w,\ldots\rangle$ is also a path in H. Continuing in this way, we obtain an up–down path. The *up–down property* leads to a very simple and fast query algorithm. We do bidirectional search in H using a modification of Dijkstra's algorithm where forward search considers only upward edges and backward search considers only downward edges. The computed path contains shortcuts that have to be expanded. To support unpacking in time linear in the output size, it suffices to store the

midpoints of shortcuts with every shortcut, i.e., the node that was contracted when the shortcut was inserted. Expanding then becomes a recursive procedure, with edges in the input graph as the base case.

We still need to discuss the order of removal. Of course, if the ordering is bad, we will need a huge number of shortcuts, resulting in large space requirements and slow queries. Many heuristics for node ordering have been considered. Even simple ones yield reasonable performance. These heuristics compute the ordering online by identifying unimportant nodes in the remaining graph to be contracted next. To compute the importance of v, a contraction of v is simulated. In particular, it is determined how many shortcuts would be inserted. This number and other statistics gathered during the simulated contraction are condensed into a score. Nodes with low score are contracted first.

We close with a comparison of CHs with the heuristic based on the official hierarchy of roads. The official hierarchy consists of not more than 10 layers. Rather than using a small, manually defined set of layers, CHs use n different layers. This means that CHs can exploit the hierarchy inherent in the input in a much more fine-grained way. The CH query algorithm is also much more aggressive than the heuristic algorithm – it switches to a higher layer in every step. CHs can afford this aggressiveness since shortcuts repair any "error" made in defining the hierarchy.

10.8.3 Transit Node Routing

Another observation from daily life can be used to obtain very fast query times. When you drive to somewhere "far away", you will leave your start location via one of only a few "important" road junctions. For example, the second author lives in Scheidt,

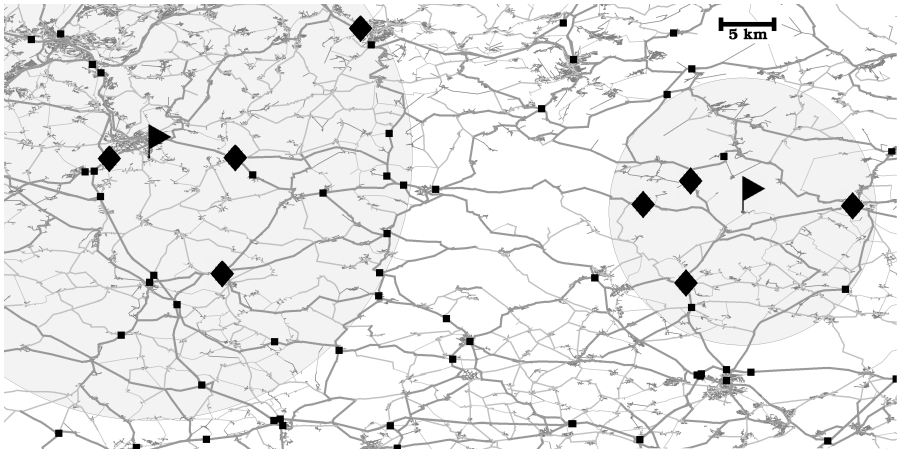


Fig. 10.13. Finding the optimal travel time between two points (the flags) somewhere between Saarbrücken and Karlsruhe amounts to retrieving the 2×4 *access nodes* (diamonds), performing 16 table lookups between all pairs of access nodes, and checking that the two disks defining the *locality filter* do not overlap. The small squares indicate further transit nodes.

a small village near Saarbrücken. For trips to the south-east to north-east, he enters the motorway system in Sankt Ingbert. For trips to the south, he enters the system at Grossblittersdorf, and so on. So all shortest paths from his home to distances far away go through a small number of transit nodes. Moreover, these transit nodes are the same for the entire population of Scheidt. As a consequence, the total number of transit nodes for all locations in Germany (or Europe, for that matter) is small. Figure 10.13 gives another example.

The notion of a transit node can be made precise and algorithmic; see [31]. In their scheme, about $\sqrt{n}$ of the most important road junctions are selected as transit nodes. Here, n is the number of nodes in the road network (a few million for the German network). The algorithmic strategy is now as follows:

(a) Select the set A of transit nodes.
(b) Compute the complete distance table between the nodes in A.
(c) For each node v, determine its transit nodes A_v conntecting v to the long-distance network. The sets A_v are typically small, usually no more than a handful of elements.
(d) For each node v, compute the distances to the nodes in A_v.

Shortest-path distances between faraway nodes can then be computed as

$$\mu(s,t) = \min_{u \in A_s} \min_{v \in A_t} \mu(s,u) + \mu(u,v) + \mu(v,t). \tag{10.1}$$

In this way, shortest-path queries are reduced to a small number of table lookups. This can be more than 1 000 000 times faster than Dijkstra's algorithm. Unfortunately, (10.1) does not work for "local" shortest paths that do not touch any transit node. Therefore, a full implementation of transit node routing needs additionally to define a *locality filter* that detects local paths and a routine for finding the local paths. This local algorithm could, for example, use CHs, the labeling method, or further layers of local transit node routing. The precomputation of the distance tables can also use all methods discussed in the preceding sections; see [183, 3, 21] for details.

10.9 Parallel Shortest Paths

Different shortest-path computations (Dijkstra, Bellman–Ford, contraction hierarchies, all-pairs) pose different challenges for parallelization. We progress from the easy to the difficult.

Computing all-pairs shortest paths is embarrassingly parallel for $p \leq n$, just assign n/p starting nodes to each PE. We only need to be careful about space consumption. We only want to replicate the distance array of the shortest-path tree currently computed and we should use a priority queue implementation that only consumes space proportional to the actual queue size rather than $\Theta(n)$.

The basic Bellman–Ford algorithm allows for a lot of fine-grained parallelism. All the edge relaxations in an iteration can be performed in parallel. We do not even have to use CAS instructions, as long as write operations are atomic. Suppose two

PEs try to update $d[v]$ to x and y, respectively, where $x < y$. The bad case is when x is written first and then overwritten by the worse value y. The overall algorithm will remain correct, since the value x will be tried again in the next iteration. Hence, as long as the bad case happens rarely, we pay only a small performance penalty. Moreover, in many situations distances decrease only rarely (see also Sect. 10.4). Therefore contention is much lower than in other situations. This *priority update principle* is also useful in many other shared memory algorithms [295]. Unfortunately, the optimizations discussed in Sect. 10.11 are more difficult to parallelize.

Many of the preprocessing techniques discussed in Sect. 10.8 can be parallelized. We outline an approach for contraction hierarchies [181, 323]. The algorithm computes a score for each node of how attractive it is to contract the node. Then it contracts in parallel all nodes which are less important than all nodes reachable within BFS distance 2. Distance 2 is necessary in order to avoid concurrent updates of adjacency lists when inserting shortcuts. Parallelizing over many CH queries is also easy. This even works on distributed memory machines where each PE knows only about a small part of the road network [181]: One stores important nodes redundantly such that the search spaces of the forward and the backward search can be computed locally. Then the forward search space is sent to the PE responsible for the backward search, which intersects the two search spaces yielding the result. In [181], preprocessing was also parallelized for distributed memory. The graph is partitioned as explained in Sect. 8.6.2, but the approach is generalized to store a *halo* of order ℓ locally: Suppose v is in some local block B according to the partition. Then it is ensured that every node within BFS distance ℓ is stored locally. This way, the local searches done during contraction can be done locally. The halo is adapted after every iteration in order to take newly inserted shortcut edges into account.

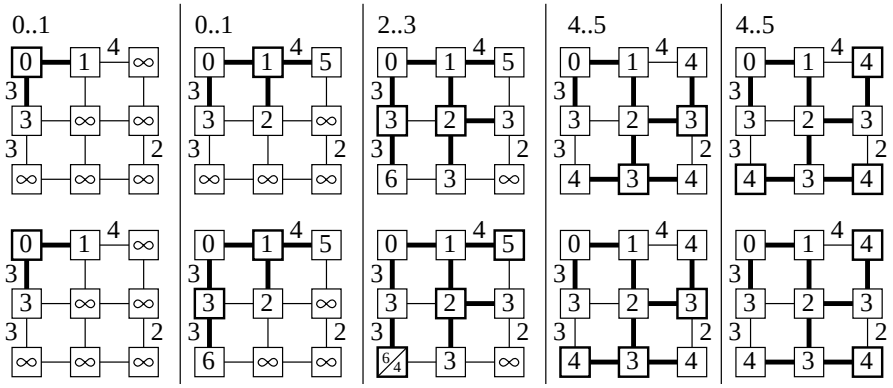


Fig. 10.14. Example execution of Δ-stepping with $\Delta = 2$ (*top*) and $\Delta = \infty$ (Bellman–Ford, *bottom*). Edges have unit weight unless a different value is explicitly given. Thick edges denote the current approximation to the shortest-path tree. Bold squares indicate nodes currently being scanned. Note that Bellman–Ford scans 11 nodes overall, whereas 2-stepping scans only 9 nodes. The range $a..b$ given in the top row indicates the bucket currently being scanned.

Unfortunately, the "bread-and-butter" shortest-path problem with a single source and nonnegative edge weights is not easy to parallelize. We may get a little bit of parallelism by scanning all nodes with smallest distance value ($\min Q$) in parallel. This can be generalized slightly by observing that a node v also has its final distance value if $d[v] \leq \min Q + \min_{(u,v)\in E} c(u,v)$ [80]. If we exploit more parallelism, for example using a parallel priority queue, it might happen that we now scan nodes v with $d[v] > \mu(v)$. We compensate for this "mistake" by rescanning nodes that are reinserted into the priority queue. This preserves correctness albeit with an increase in the work done.

We can develop this idea further. Once we abandon the nice properties of Dijkstra's algorithm, it makes sense to switch to an approximate priority queue that has less overhead. A simple way to do this is to use a bucket priority queue where key x is mapped to bucket $\lfloor x/\Delta \rfloor$, where Δ is a tuning parameter. This queue is a slight generalization of the simple integer bucket queue from Sect. 10.5.1. The idea behind the *Δ-stepping* algorithm is to achieve parallelism by scanning in parallel all nodes in the first nonempty bucket of the bucket queue. This algorithm can be shown to be work efficient at least for random edge weights. Adding further measures, that involve different treatments of edges with weight $< \Delta$, one can obtain the following bound.

Theorem 10.14 ([227]). *$1/d$-stepping with random edge weights from $[0,1]$ runs in time*

$$O\left(\frac{n+m}{p} + dL\log n + \log^2 n\right)$$

on a CRCW-PRAM, where d is the maximum degree and L is the largest shortest-path distance from the starting node.

Δ-stepping can also be viewed as a generalization of both the Bellman–Ford algorithm and Dijkstra's algorithm. For $\Delta \leq \min\{e \in E : c(e)\}$, we get Dijkstra's algorithm, and for $\Delta \geq \max\{e \in E : c(e)\}$, we get the Bellman–Ford algorithm.

Implementing a basic variant of Δ-stepping in parallel is a slight generalization of the parallel BFS described in Sect. 9.2. We now describe the distributed memory version in more detail. Each PE i maintains a local bucket priority queue Q storing reached but unscanned nodes assigned to PE i. One iteration of the main loop first finds the first bucket j that is nonempty on some PE. All PEs then work in parallel on $Q[j]$. For a node $u \in Q[j]$ and an outgoing edge $e = (u,v)$, a relaxation request $(v, d[u] + c(e))$ is sent to the PE responsible for node v. The iteration is finished by processing incoming requests. For a request (v,c), if $d[v] > c$, $d[v]$ is set to c and the queue is updated accordingly, i.e., if v is already in Q, a *decreaseKey* is performed and otherwise an insertion. Note that in Δ-stepping, a node may be reinserted after it has been scanned. Figure 10.14 gives an example.

10.10 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 $parent(v) = \perp$ if and only if $d[v] = +\infty$, i.e., when $parent(v) = \perp$, we assume that $d[v] = +\infty$ and ignore the number stored in $d[v]$.

A refined implementation of the Bellman–Ford algorithm [217, 307] explicitly maintains a current approximation T to 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.

Implementations of contraction hierarchies are available at `github.com` under `RoutingKit/RoutingKit` and `Project-OSRM/osrm-backend`.

10.10.1 C++

LEDA [194] has a special priority queue class *node_pq* that implements priority queues of graph nodes. Both LEDA and the Boost graph library [50] 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 10.9).

The Boost graph library [50] and the LEMON graph library [200] use the *visitor concept* to support graph traversal. A visitor class has user-definable methods that are called at *event points* during the execution of a graph traversal algorithm. For example, the DFS visitor defines event points similar to the operations *init*, *root*, *traverse*$*$, and *backtrack* used in our DFS template; there are more event points in Boost and LEMON.

10.10.2 Java

The JGraphT [166] library supports DFS in a very flexible way, not very much different from the visitor concept described for Boost and LEMON. There are also more specialized algorithms, for example for biconnected components.

10.11 Historical Notes and Further Findings

Dijkstra [96], Bellman [38], and Ford [108] found their algorithms in the 1950s. The original version of Dijkstra's algorithm had a running time $O(m + n^2)$ 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 [331], Fibonacci heaps [115], bucket heaps [92], and radix heaps [10]. Experimental comparisons can be found in [71, 217]. For integer keys, radix heaps are not the end of the story. The

best theoretical result is $O(m + n \log\log n)$ time [312]. For *undirected* graphs, linear time can be achieved [310]. The latter algorithm still scans nodes one after the other, but not in the same order as in Dijkstra's algorithm.

Meyer [226] gave the first shortest-path algorithm with linear average-case running time. The algorithm ALD was found by Goldberg [129].

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 $O(m\sqrt{n}\log N)$ [128].

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. A recent overview can be found in [30]. 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 [103, 312, 314].

There is a generalization of the shortest-path problem that considers several cost functions at once. For example, your grandfather may 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 [143, 222].

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 quickest path.

Exercise 10.24. 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 in 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?