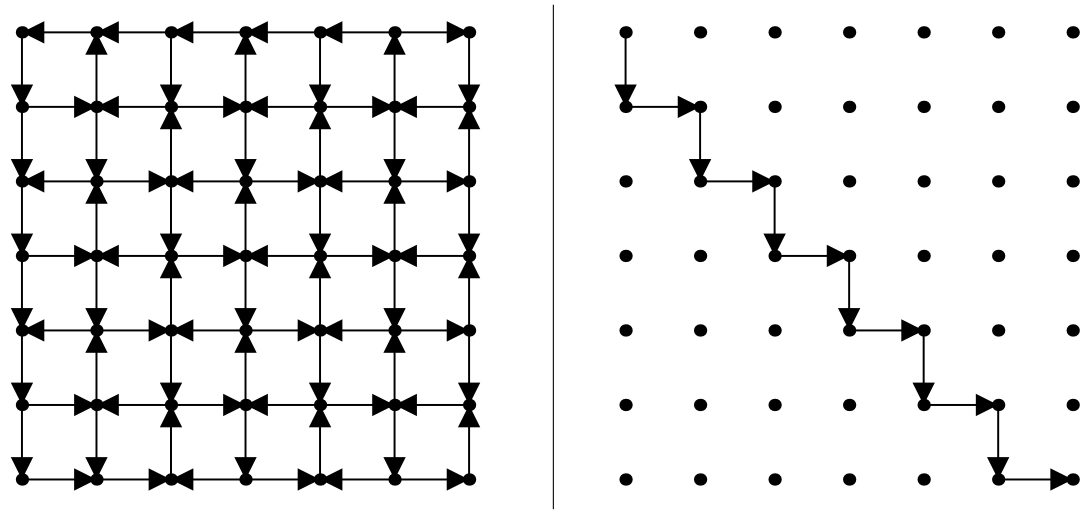

INPUT OUTPUT

15.4 Shortest Path

Input description: An edge-weighted graph G, with vertices s and t.

Problem description: Find the shortest path from s to t in G.

Discussion: The problem of finding shortest paths in a graph has several applications, some quite surprising:

- The most obvious applications arise in transportation or communications, such as finding the best route to drive between Chicago and Phoenix or figuring how to direct packets to a destination across a network.
- Consider the task of partitioning a digitized image into regions containing distinct objects—a problem known as *image segmentation*. Separating lines are needed to carve the space into regions, but what path should these lines take through the grid? We may want a line that relatively directly goes from x to y, but avoids cutting through object pixels as much as possible. This grid of pixels can be modeled as a graph, with the cost of an edge reflecting the color transitions between neighboring pixels. The shortest path from x to y in this weighted graph defines the best separating line.
- *Homophones* are words that sound alike, such as *to*, *two*, and *too*. Distinguishing between homophones is a major problem in speech recognition systems.

The key is to bring some notion of grammatical constraints into the interpretation. We map each string of phonemes (recognized sounds) into words they might possibly match. We construct a graph whose vertices correspond to these possible word interpretations, with edges between neighboring word-interpretations. If we set the weight of each edge to reflect the likelihood of transition, the shortest path across this graph defines the best interpretation of the sentence. See Section 6.4 (page 212) for a more detailed account of a similar application.

- Suppose we want to draw an informative visualization of a graph. The "center" of the graph should appear near the center of the page. A good definition of the graph center is the vertex that minimizes the maximum distance to any other vertex in the graph. Identifying this center point requires knowing the distance (i.e. , shortest path) between all pairs of vertices.

The primary algorithm for finding shortest paths is *Dijkstra's algorithm*, which efficiently computes the shortest path from a given starting vertex x to all $n-1$ other vertices. In each iteration, it identifies a new vertex v for which the shortest path from x to v is known. We maintain a set of vertices S to which we currently know the shortest path from v, and this set grows by one vertex in each iteration. In each iteration, we identify the edge (u, v) where $u, u' \in S$ and $v, v' \in V - S$ such that

$$dist(x, u) + weight(u, v) = \min_{(u', v') \in E} dist(x, u') + weight(u', v')$$

This edge (u, v) gets added to a *shortest path tree*, whose root is x and describes all the shortest paths from x.

An $O(n^2)$ implementation of Dijkstra's algorithm appears in in Section 6.3.1 (page 206). Theoretically faster times can be achieved using significantly more complicated data structures, as described below. If we just need to know the shortest path from x to y, terminate the algorithm as soon as y enters S.

Dijkstra's algorithm is the right choice for single-source shortest path on positively weighted graphs. However, special circumstances dictate different choices:

- *Is your graph weighted or unweighted?* – If your graph is unweighted, a simple breadth-first search starting from the source vertex will find the shortest path to all other vertices in linear time. Only when edges have different weights do you need more sophisticated algorithms. A breadth-first search is both simpler and faster than Dijkstra's algorithm.

- *Does your graph have negative cost weights?* – Dijkstra's algorithm assumes that all edges have positive cost. For graphs with edges of negative weight, you must use the more general, but less efficient, Bellman-Ford algorithm. Graphs with negative cost cycles are an even bigger problem. Observe that the shortest x to y path in such a graph is not defined because we can detour

from x to the negative cost cycle and repeatedly loop around it, making the total cost arbitrarily small.

Note that adding a fixed amount of weight to make each edge positive *does not* solve the problem. Dijkstra's algorithm will then favor paths using a fewer number of edges, even if those were not the shortest weighted paths in the original graph.

- *Is your input a set of geometric obstacles instead of a graph?* – Many applications seek the shortest path between two points in a geometric setting, such as an obstacle-filled room. The most straightforward solution is to convert your problem into a graph of distances to feed to Dijkstra's algorithm. Vertices will correspond the vertices on the boundaries of the obstacles, with edges defined only between pairs of vertices that "see" each other.

  Alternately, there are more efficient geometric algorithms that compute the shortest path directly from the arrangement of obstacles. See Section 17.14 (page 610) on motion planning and the Notes section for pointers to such geometric algorithms.

- *Is your graph acyclic—i.e. , a DAG?* – Shortest paths in directed acyclic graphs can be found in linear time. Perform a topological sort to order the vertices such that all edges go from left to right starting from source s. The distance from s to itself, $d(s, s)$, clearly equals 0. We now process the vertices from left to right. Observe that

$$d(s, j) = \min_{(x,i) \in E} d(s, i) + w(i, j)$$

  since we already know the shortest path $d(s, i)$ for all vertices to the left of j. Indeed, most dynamic programming problems can be formulated as shortest paths on specific DAGs. Note that the same algorithm (replacing min with max) also suffices to find the *longest path* in a DAG, which is useful in many applications like scheduling (see Section 14.9 (page 468)).

- *Do you need the shortest path between all pairs of points?* – If you are interested in the shortest path between all pairs of vertices, one solution is to run Dijkstra n times, once with each vertex as the source. The Floyd-Warshall algorithm is a slick $O(n^3)$ dynamic programming algorithm for all-pairs shortest path, which is faster and easier to program than Dijkstra. It works with negative cost edges but not cycles, and is presented with an implementation in Section 6.3.2 (page 210). Let M denote the distance matrix, where $M_{ij} = \infty$ if there is no edge (i, j):

$$D^0 = M$$
$$\text{for } k = 1 \text{ to } n \text{ do}$$
$$\quad \text{for } i = 1 \text{ to } n \text{ do}$$

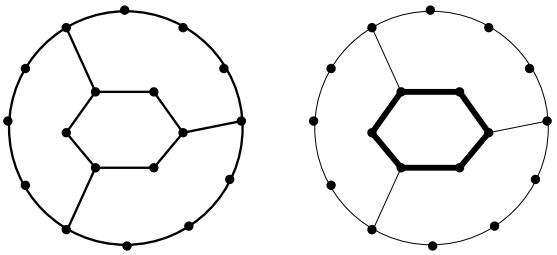

Figure 15.1: The girth, or shortest cycle, in a graph

---

for $j = 1$ to n do
$D_{ij}^k = \min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})$
Return D^n

The key to understanding Floyd's algorithm is that D_{ij}^k denotes "the length of the shortest path from i to j that goes through vertices $1, \ldots, k$ as possible intermediate vertices." Note that $O(n^2)$ space suffices, since we only must keep D^k and D^{k-1} around at time k.

- *How do I find the shortest cycle in a graph?* – One application of all-pairs shortest path is to find the shortest cycle in a graph, called its *girth*. Floyd's algorithm can be used to compute d_{ii} for $1 \leq i \leq n$, which is the length of the shortest way to get from vertex i to i—in other words, the shortest cycle through i.

  This *might* be what you want. The shortest cycle through x is likely to go from x to y back to x, using the same edge twice. A *simple* cycle is one that visits no edge or vertex twice. To find the shortest simple cycle, the easiest approach is to compute the lengths of the shortest paths from i to all other vertices, and then explicitly check whether there is an acceptable edge from each vertex back to i.

  Finding the *longest* cycle in a graph includes Hamiltonian cycle as a special case (see Section 16.5), so it is NP-complete.

The all-pairs shortest path matrix can be used to compute several useful invariants related to the center of graph G. The *eccentricity* of vertex v in a graph is the shortest-path distance to the farthest vertex from v. From the eccentricity come other graph invariants. The *radius* of a graph is the smallest eccentricity of any vertex, while the *center* is the set of vertices whose eccentricity is the radius. The *diameter* of a graph is the maximum eccentricity of any vertex.

Implementations: The highest performance shortest path codes are due to Andrew Goldberg and his collaborators, at *http://www.avglab.com/andrew/soft.html*.

In particular, MLB is a C++ short path implementation for non-negative, integer-weighted edges. See [Gol01] for details of the algorithm and its implementation. Its running time is typically only 4-5 times that of a breadth-first search, and it is capable of handling graphs with millions of vertices. High-performance C implementations of both Dijkstra and Bellman-Ford are also available [CGR99].

All the C++ and Java graph libraries discussed in Section 12.4 (page 381) include at least an implementation of Dijkstra's algorithm. The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) has a particularly broad collection, including Bellman-Ford and Johnson's all-pairs shortest-path algorithm. LEDA (see Section 19.1.1 (page 658)) provides good implementations in C++ for all of the shortest-path algorithms we have discussed, including Dijkstra, Bellman-Ford, and Floyd's algorithms. JGraphT (*http://jgrapht.sourceforge.net/*) provides both Dijkstra and Bellman-Ford in Java. C language implementations of Dijkstra and Floyd's algorithms are provided in the library from this book. See Section 19.1.10 (page 661) for details.

Shortest-path algorithms was the subject of the 9th DIMACS Implementation Challenge, held in October 2006. Implementations of efficient algorithms for several aspects of finding shortest paths were discussed. The papers, instances, and implementations are available at *http://dimacs.rutgers.edu/Challenges/*.

Notes: Good expositions on Dijkstra's algorithm [Dij59], the Bellman-Ford algorithm [Bel58, FF62], and Floyd's all-pairs-shortest-path algorithm [Flo62] include [CLRS01]. Zwick [Zwi01] provides an up-to-date survey on shortest path algorithms. Geometric shortest-path algorithms are surveyed by Mitchell [PN04].

The fastest algorithm known for single-source shortest-path for positive edge weight graphs is Dijkstra's algorithm with Fibonacci heaps, running in $O(m + n \log n)$ time [FT87]. Experimental studies of shortest-path algorithms include [DF79, DGKK79]. However, these experiments were done before Fibonacci heaps were developed. See [CGR99] for a more recent study. Heuristics can be used to enhance the performance of Dijkstra's algorithm in practice. Holzer, et al. [HSWW05] provide a careful experimental study of how four such heuristics interact together.

Online services like Mapquest quickly find at least an approximate shortest path between two points in enormous road networks. This problem differs somewhat from the shortest-path problems here in that (1) preprocessing costs can be amortized over many point-to-point queries, (2) the backbone of high-speed, long-distance highways can reduce the path problem to identifying the best place to get on and off this backbone, and (3) approximate or heuristic solutions suffice in practice.

The A^*-algorithm performs a best-first search for the shortest path coupled with a lower-bound analysis to establish when the best path we have seen is indeed the shortest-path in the graph. Goldberg, Kaplan, and Werneck [GKW06, GKW07] describe an implementation of A^* capable of answering point-to-point queries in one millisecond on national-scale road networks after two hours of preprocessing.

Many applications demand multiple alternative short paths in addition to the optimal path. This motivates the problem of finding the k shortest paths. Variants exist depending upon whether the paths must be simple, or can contain cycles. Eppstein [Epp98] generates an implicit representation of these paths in $O(m+n \log n+k)$ time, from which each path

can be reconstructed in $O(n)$ time. Hershberger, et al. [HMS03] present a new algorithm and experimental results.

Fast algorithms for computing the girth are known for both general [IR78] and planar graphs [Dji00].

Related Problems: Network flow (see page 509), motion planning (see page 610).