16. Shortest Paths

The input to a shortest path problem is a graph (often a directed graph) in which every edge has an associated cost (sometimes called a weight or length). Our goal is to find paths of minimum cost between designated source and destination nodes, where the cost of a path is just the sum of the costs of the edges along the path. The problem is quite natural and intuitive, and one that we instinctively solve on a daily basis — any time we choose a route between two points along which to walk or drive, we typically try to pick the shortest route.

Due to its diverse applications both in practice and in theory, the shortest path problem is one of the most widely-studied problems in the field of algorithmic computer science. A surprisingly broad collection of algorithmic problems can be formulated and solved as shortest path problems, even though for some of them it may not always be immediately obvious that they have anything to do with shortest paths. For example, in Chapter 11 we have seen how many simple dynamic programming algorithms essentially involve the computation of shortest paths through a directed acyclic graph (DAG).

Three common variants of shortest path problems are typically studied:

- **Single-Source.** The most common variant of the problem is called the *single-source* or *one-to-all* shortest path problem, where we wish to find the shortest path from a designated source node s to every other node in a graph. It is such a common variant, in fact, that when people speak of "the shortest path problem", they usually refer to this variant of the problem. Note that this variant is identical to the symmetric *single-destination* or *all-to-one* shortest path problem; we can transform one into the other by reversing the direction of every edge in our graph.
- **All-Pairs.** Another common problem variant is the *all-pairs* or *all-to-all* shortest path problem, for which we need to compute the shortest path from every node to every other node. Of course, we can solve this by multiple invocations of an algorithm for the single-source problem, one for each node acting as the source. However, later in this chapter, we will see more elegant ways to approach this variant.
- **Point-to-Point.** Even though it may sound like the most fundamental variant of all, we rarely study shortest path problems with only a single source and

destination. The reason for this is simply that we currently do not know any algorithm for solving this problem any faster (in terms of worst-case running time) than the single-source variant.

We begin this chapter with a discussion of mathematical properties of shortest paths, followed by a detailed introduction to the single-source problem. We then move on the all-pairs problem, followed a discussion of effective heuristics for the point-to-point problem. Towards the end of the chapter, we also highlight two final concepts of significance to shortest path computation: the notion of "reduced" edge costs, and how we can easily modify most of our standard shortest path algorithms to solve a variety of related path-finding problems.

Problem 274 (Formulating Shortest Path Problems). Sometimes we are faced with a problem that resembles a shortest path problem but has some extra constraints or some sort of complicating structure. It is often possible to reformulate such problems as "pure" shortest path problems, allowing them to be solved efficiently by existing shortest path algorithms. Please show how to reformulate each of the problems below as standard shortest path problems, free of any extraneous constraints.

(a) In an automobile transportation network, edges represent roads and nodes represent intersections. In such networks, it often takes longer to make a left turn at an intersection than a right turn. How can we express such "turning penalty" constraints within the framework of a standard shortest path problem? [Solution]

(b) Suppose we are traveling through an transportation network in which every edge, in addition to having a specified length, may or may not charge a toll. Suppose we have a budget that permits traveling on at most T toll edges. Given the set of edges that charge tolls, show how to compute single-source shortest paths that may each contain no more than T toll edges. [Solution]

(c) Suppose every edge in our graph has an associated *failure probability*, where failures of different edges are independent events. The *maximum reliability path problem* involves computing a path of lowest failure probability from a designated source node to all other nodes. Show how to convert such a problem into a standard shortest path problem. [Solution]

(d) It may be easy to find the shortest path in a graph, but what about the second-shortest path? In other words, given a shortest path p between two nodes s and t, show how to efficiently compute the shortest path from s to t that is different from p. [Solution]

(e) Automobile transportation networks often have traffic lights that restrict the flow of traffic through nodes. Let us assume for simplicity that all traffic lights in our network have a one-minute cycle time, out of which they are green for 30 seconds and red for 30 seconds. Although the "green intervals" of different lights do not necessarily need to coincide, we assume that all lights are synchronized with a global clock and only transition from between green and red precisely on boundaries between consecutive seconds (in other words, we treat time as being discrete). How can we compute the fastest transit time between two nodes s and t in such a network? [Solution]

16.1 The Structure of Shortest Paths

Shortest paths satisfy many useful properties, and it will be to our advantage to discuss some of these before we dive into a discussion of algorithms for computing

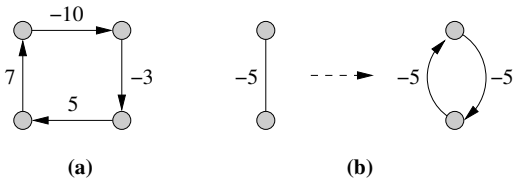

FIGURE 16.1: (a) A negative-cost directed cycle, and (b) how a negative-cost directed cycle can be created when replacing a negative-cost undirected edge with a pair of directed edges.

shortest paths. As a warm-up, here is a simple example: subpaths of shortest paths are also shortest paths. That is, if we take any smaller piece of any shortest path, then that subpath must also be a shortest path between its respective endpoints [Short justification]. This is in example of the *optimal substructure property* we have seen in previous chapters, which opens the door to the possibility of using greedy or dynamic programming algorithms. It allows us to build up shortest paths consisting of many edges from shortest paths consisting of fewer edges.

16.1.1 Negative-Cost Edges and Cycles

Many shortest path applications in practice involve nonnegative edge costs; for example, edge costs typically represent distances or travel times. However, negative-cost edges are also surprisingly common in some settings, such as when shortest path computation is used as a subroutine as a part of a more sophisticated algorithm, such as the "successive shortest path" family of algorithms for computing minimum-cost network flows (Chapter 18).

Negative-cost edges present us with some particular challenges when we study shortest paths. We must not only use more powerful (i.e., slower) algorithms to compute shortest paths in the presence of negative edge costs, but we must be on the lookout for *negative-cost cycles*. Any time negative-cost edges might be present in our graph, we adopt an important standard assumption that there cannot be any directed cycles in our graph having negative total cost, as shown for example in Figure 16.1(a). If there were such a cycle in our graph, a shortest path would prefer to spin endlessly around it en route to its destination, resulting in somewhat ill-defined shortest paths having cost $-\infty$. We could attempt to remedy this situation by asserting that all shortest paths must be *simple* (containing no cycles). However, this is also no good because the shortest simple path problem turns out to be NP-hard in graphs having negative-cost directed cycles [Why?]. Therefore, we must insist that if negative-cost edges are present in our graph, they must not contribute to any negative-cost directed cycles[1]. Some of the algorithms we present later are able to detect the presence of negative-cost cycles, so they can at least identify problem instances for which shortest path computation is not tractable.

[1]For the single-source shortest path problem, a negative-cost cycle is technically "harmless" if we cannot reach it from the source node, so we could also make our assumption more precise by only excluding negative-cost cycles reachable from the source.

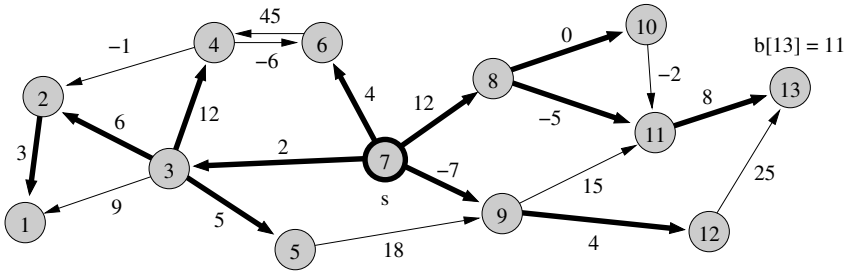

FIGURE 16.2: An example shortest path tree emanating from a source node $s = 7$. The cost of every edge is shown, and edges in the shortest path tree are highlighted. Notice how this tree is not necessarily unique, since edge $(3, 1)$ could take the place of the edge $(2, 1)$.

For any pair of nodes i and j, the cost of a shortest $i \rightsquigarrow j$ path is a unique quantity even though there might be several different shortest paths from i to j. Some of these paths might even contain cycles. Shortest paths certainly cannot contain positive-cost cycles, since any such cycle could be "spliced out" yielding a path of even lower cost. They also cannot contain negative-cost cycles, since by assumption negative cycles are not welcome in our graphs. Nothing prohibits zero-cost cycles from being included in a shortest path; however, we can always find at least one simple $i \rightsquigarrow j$ shortest path by splicing out zero-cost cycles. We can therefore focus our attention on finding *simple* shortest paths.

16.1.2 Directed Versus Undirected Graphs

All of our fundamental shortest path algorithms operate on directed graphs, so any undirected graph given as input must first be converted to a directed graph using the standard transformation where we replace each undirected edge with a pair of oppositely-oriented directed edges having the same cost as the undirected edge. Tragically, however, if any undirected edge has negative cost, this transformation creates an unwanted negative-cost directed cycle, as shown in Figure 16.1(b), rendering the problem intractable. Let us therefore assume, at least for now, that any undirected graph given to us as input contains no negative-cost edges. As it turns out, the undirected shortest path problem in the presence of negative-cost edges (but no negative-cost cycles) is solvable in polynomial time, but this problem is a bit more complicated than one might initially expect and we will need to wait until Chapter 20 before we will have the algorithmic machinery to solve it. For the remainder of this chapter, let us assume that we are dealing exclusively with directed graphs, except when noted.

16.1.3 Shortest Versus Longest Paths

Knowing the evils of negative-cost directed cycles, it is now easier to understand why computer scientists study shortest path problems instead of *longest path* problems. A longest path problem can be transformed into an equivalent shortest path problem

by negating all of its edge costs. Therefore, directed cycles of positive cost introduce the same complications with longest path problems as do negative-cost cycles for shortest path problems. Since most graphs in the world have directed cycles of positive cost, the longest path problem is therefore rarely tractable.

A notable exception to this situation is the case of a directed acyclic graph (DAG). Since DAGs have no directed cycles to begin with, we never encounter any difficulty due to negative or positive cycles, and we can freely solve all instances of shortest and longest path problems on these graphs. As you may recall from Chapter 11, the single-source shortest and longest path problems can be solved in linear time in a DAG via dynamic programming.

16.1.4 Shortest Path Trees

Shortest path algorithms often focus only on computing the *costs* of shortest paths, since these are uniquely determined by the endpoints of the path. However, we often want to compute the paths themselves, which we can represent in a succinct fashion in the form of a *shortest path tree*[2]. Just like a breadth-first search tree or depth-first search tree, the shortest path tree encodes a set of shortest paths emanating from a source node s in the form of a tree directed outward from s, an example of which is shown in Figure 16.2. As with a BFS or DFS tree, we specify a shortest path tree by having each node i maintain a "backpointer" $b[i]$ to its parent, which is the second-to-last node on an $s \rightsquigarrow i$ shortest path. We can therefore trace out an $s \rightsquigarrow i$ shortest path (in reverse) by following successive backpointers from i back to s, the same general process we use to find a path through a BFS or DFS tree, or to reconstruct the solution of a dynamic programming problem.

For the all-pairs variant, there will be a different shortest path tree emanating from every source node. In this case, to store the structure of shortest paths we need an $n \times n$ matrix of backpointers where $b[i, j]$ gives the second-to-last node on a shortest path from i to j. One can then reconstruct a shortest path between any two nodes i and j by following these backpointers from j back to i.

16.1.5 The Triangle Inequality

You may recall the triangle inequality from elementary geometry: the sum of any two side lengths of a triangle must be at least as large as the length of the third side, as shown in Figure 16.3(a). The triangle inequality is perhaps the most important property satisfied by shortest path costs, since it gives us a simple way to characterize an optimal solution to a shortest path problem.

In the single-source shortest path problem, our goal is to determine for every node i the cost $c[i]$ of the shortest path from s to i. The quantity $c[i]$ is commonly known as the *distance label* of node i, particularly for applications where edge costs represent lengths or distances. We'll call it a *cost* label in order to maintain consistency with our existing notation. The triangle inequality for the single-source problem

[2]Note that the existence of a tree of shortest paths follows easily from the facts that (i) shortest paths can be assumed to be simple, and (ii) subpaths of shortest paths are also shortest paths. Note also that there can be many valid shortest path trees leaving a particular source node, since shortest paths themselves are not necessarily unique.

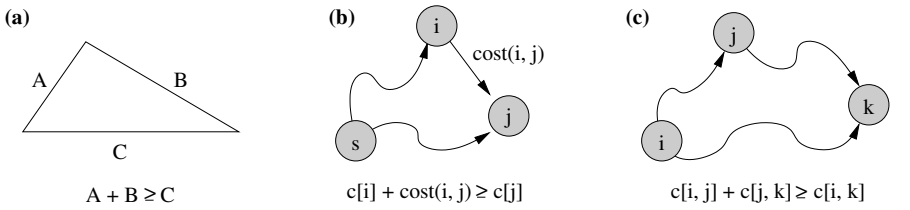

FIGURE 16.3: Illustrations of the triangle inequality (a) in classical geometry, (b) for the single-source shortest path problem, with (i, j) being an edge, and (c) for the single-source shortest path problem, with i, j, and k being any three nodes in our graph.

is illustrated in Figure 16.3(b): for every directed edge (i, j), we must have $c[i] + cost(i, j) \geq c[j]$; otherwise, $c[j]$ cannot be the correct shortest path cost from s to j, since the extension of a shortest $s \rightsquigarrow i$ path by the edge (i, j) would give us a better alternative.

For the all-pairs problem, our goal is to determine the shortest path cost $c[i, j]$ between every pair of nodes i and j. Here, if we take any three nodes i, j, and k, shortest path costs between these nodes must satisfy the triangle inequality as illustrated in Figure 16.3(c). If the triangle inequality were not satisfied and $c[i, j] + c[j, k] < c[i, k]$, then we know $c[i, k]$ cannot represent the correct shortest path cost from i to k, since a better path is available by following the shortest path from i to j and then from j to k.

Recall from Section 2.5 that a *metric* is a function that measures pairwise distances among elements of a set, that is nonnegative, symmetric, and that satisfies the triangle inequality. In an undirected graph, shortest path costs therefore give us a metric over the nodes in our graph.

Problem 275 (Shortest Path Optimality Conditions). We have just argued that the triangle inequality is *necessary* for a set of shortest path costs to be optimal. Consider the following two questions for the single-source shortest path problem (if you like, you can try to answer the equivalent questions for the all-pairs problem too):

(a) Prove that the triangle inequality is also a *sufficient* condition for optimality. That is, show that a set of cost labels represents the correct set of optimal path costs from the source to every node if and only if:
   (i) The cost labels satisfy the triangle inequality: for every edge (i, j), we have $c[i] + cost(i, j) \geq c[j]$,
   (ii) Each cost label $c[i]$ gives the cost of some path from s to i, and
   (iii) $c[s] = 0$.

   These conditions are known as the *optimality conditions* for shortest paths. How quickly can we check if a set of cost labels satisfies these conditions? That is, how fast can we verify correctness of a solution to a shortest path problem? [Solution]

(b) Show that it is possible to satisfy the optimality conditions above if and only if our graph has no directed cycles of negative cost reachable from the source. [Solution]

Problem 276 (Solving Systems of Difference Constraints). Suppose we are given a system of linear equations and inequalities in which the left-hand side of every

equation is a difference of two variables or a single variable by itself. For example:

$$\begin{aligned} x_3 - x_1 &= 10 \\ x_1 - x_2 &\leq -3 \\ x_2 &\leq 6 \\ x_3 &= 12 \end{aligned}$$

Knowing what you know about the triangle inequality, show how to solve such a system using a carefully-formulated shortest path problem. [Solution]

Problem 277 (Shift Scheduling). Suppose you are in charge of scheduling employees for 8-hour "shifts" over the course of a day. Specifically, you have a 24-hour time period during which you may schedule employees, each employee must work an 8-hour consecutive shift, and you would like to minimize the total number of employees scheduled subject to two types of constraints: *packing constraints* and *covering constraints*. An example of a packing constraint is "at most 5 employees can be working at time t" and a covering constraint looks like "at least 15 employees must be working at time t". For consistency, let us adopt the convention that an employee whose shift lasts from time t to time $t + 8$ is considered to be working at the instant of time t, but not at $t + 8$. See if you can compute an optimal schedule using shortest path techniques; the result from the preceding problem may be of help. You may assume the office is closed at midnight, so there are no shifts that "wrap around" from one day to the next. Observe that this result is sufficiently general to allow you to use shortest paths to solve any linear program whose constraint matrix has rows satisfying the "consecutive ones" property (see Section 12.3). [Solution]

16.1.6 Tight Edges

Consider an optimal set of path costs $c[1 \ldots n]$ for an instance of the single-source problem. We call an edge (i, j) *tight* if it satisfies the triangle inequality with equality; in other words, if $c[i] + cost(i, j) = c[j]$. All other edges (for which the triangle inequality is a strict inequality) are said to be *slack*. Tight edges are special because they are the edges that make up shortest paths. That is, any path consisting solely of tight edges must be a shortest path. Moreover, if we delete all the slack edges from our graph, we will be left with just the shortest path tree, plus any other edges that happen to be tight as well (if such edges exist, then our shortest path tree will not be unique).

For nice physical interpretation of the notion of tight and slack edges, think of a graph as a collection of balls (nodes) held together by strings (edges), where the cost of an edge corresponds to the length of a string. Here, if we suspend the graph by holding it from the source node, the strings along shortest paths will stretch tight.

Problem 278 (Redundancy in Cost Labels and Backpointers). Although a single-source shortest path algorithm should ideally compute both an optimal cost label $c[i]$ and a backpointer $b[i]$ for every node i, it suffices to compute only one of these quantities for every node. Show that if we know the $c[i]$'s, we can compute a valid set of $b[i]$'s in linear time, and similarly show how to compute the $c[i]'s$ from the $b[i]$'s in linear time. [Solution]

Problem 279 (Shortest Path Sensitivity Analysis). Suppose we have solved an instance of the single source shortest path problem, computing the shortest path cost

$c[i]$ from the source to every node i in our graph. We wish to determine how sensitive this solution is to small changes in edge costs. Please give a linear-time algorithm that determines for every edge the largest possible range of values that can be added or subtracted from the cost of the edge without causing a change in the cost of any shortest path. You may want to consider first the simpler case where all edge costs in the initial graph are strictly positive. [Solution]

Problem 280 (Counting Shortest Paths). Consider a single-source shortest path problem with strictly positive edge costs, and suppose we have already computed the shortest path cost $c[i]$ from the source to every node i. Show how to count the number of different $s \rightsquigarrow i$ shortest paths to every node i in linear time. Since this problem might involve numbers that are too large to store in a single $\Theta(\log n)$-bit word (the usual assumption with the RAM model), please assume for simplicity either that we have a RAM with unbounded word size, or that we are using the real RAM model. [Solution]

16.2 Single-Source Shortest Path Algorithms

We now know everything about shortest paths except how to actually compute them. In this section, we discuss algorithms for the single-source shortest path problem, Let us first briefly recall special cases of the single-source problem we already know how to solve:

- **Unweighted Graphs.** If the edges in a graph have no costs, then a shortest path is simply a path with the fewest edges. In Chapter 15, we learned that breadth-first search solves this problem in linear time[3].
- **Directed Acyclic Graphs.** Iin Chapter 11, we showed how many dynamic programming problems can be interpreted as the computation of single-source shortest paths through a DAG, and we also showed how to solve the single-source shortest path and longest path problems in linear time on a DAG.

We next describe two closely-related approaches for the single-source shortest path problem: the class of *label-correcting* algorithms, and the *Bellman-Ford* algorithm. After that, we discuss *Dijkstra's* algorithm, which gives a much faster solution in the special case that all edges have nonnegative costs. Finally, we discuss further improvements for RAM model of computation, where we can assume edge costs are integer-valued.

16.2.1 Label-Correcting Algorithms

Back in Chapter 3, we introduced the notion of an "iterative refinement" algorithm: we start with some (non-optimal) solution, and as long as we can identify part of it that isn't optimal we keep improving the solution until it eventually becomes optimal. This simple method leads us to our first class of single-source shortest path algorithms, known as *label-correcting* algorithms.

[3]Recall that we generally assume for simplicity all our graphs are at least connected (in the undirected sense), so we can write the running time of a "linear time" graph algorithm as $O(m)$ rather than $O(m+n)$, since $m \geq n-1$.

Suppose we start out with an initial solution consisting of a set of cost labels where $c[s] = 0$ and all others are set to infinity. We want to repeatedly refine this solution by "correcting" various cost labels until they eventually all become optimal. During the process, we will ensure that all non-infinite cost labels actually reflect the cost of valid paths through the graph, so all we need to do to check optimality of our solution is verify that every edge (i, j) satisfies the triangle inequality $c[i] + cost(i, j) \geq c[j]$. If any edge (i, j) violates the triangle inequality (i.e. $c[i] + cost(i, j) < c[j]$), this not only tells us that our current solution is non-optimal, but suggests a means of improving it. In this case, we have found a more optimal path to j that goes from s to i and then along the edge (i, j), so we can set $c[j]$ equal to $c[i] + cost(i, j)$ and accordingly set the backpointer of j to i.

The procedure of checking satisfaction of the triangle inequality for an edge (i, j) and using this to improve the cost label of j is a fundamental operation in many shortest path algorithms and worthy of a name. In many texts, this process is known as *relaxing* the edge (i, j). However, although this name makes sense in terms of its mathematical origins, it does not really convey the notion that the edge is being used to "tighten" the shortest path label of j, essentially pulling j closer to the source if we wish to think in terms of the physical ball and string analogy. For this reason, we will refer to this procedure as *tightening* an edge. The tightening procedure is illustrate in pseudocode in Figure 16.4(a). All of our fundamental shortest path algorithms modify cost labels using the *tighten* operation, selecting edges to tighten in some order until eventually all edges satisfy the triangle inequality, at which point we know we have reached an optimal solution. The *generic label-correcting algorithm* is perhaps the simplest incarnation of this idea, shown in pseudocode in Figure 16.4(b). Unfortunately, this approach can take exponential time in the worst case; we prove this later in problem 294.

16.2.2 The Bellman-Ford Algorithm

Suppose we perform a variation of the label-correcting algorithm in which we repeatedly scan through all m edges in the graph (in arbitrary order), tightening whenever we find a violation of the triangle inequality. It turns out that after $n-1$ such passes through all of the edges, our shortest path cost labels will be optimal! In fact, in practice we often need far fewer than $n-1$ passes — we can terminate the algorithm any time we make a pass over all of the edges and notice that all of them already satisfy the triangle inequality.

This algorithm, illustrated in pseudocode in Figure 16.4(c), is known as the *Bellman-Ford algorithm*. It runs in $O(mn)$ time as we can see above from the fact that it makes $n-1$ passes over all m edges. Correctness of the algorithm (i.e., that it converges to an optimal set of cost labels), is fairly easy to see by interpreting it as a dynamic programming algorithm [Brief details]. For further insight into the structure of this algorithm, the reader may wish to revisit problem **??** in Chapter 11. If you recall that dynamic programming algorithms can often be viewed in the context of computing shortest paths through a DAG, this problem depicts what is essentially the Bellman-Ford algorithm from the alternative viewpoint of shortest paths through a DAG. [Further details]

A nice feature of the Bellman-Ford algorithm is the fact that it can detect negative-

Tighten(i, j):
(a)
1. If $c[i] + cost(i, j) < c[j]$,
2. $\quad c[j] \leftarrow c[i] + cost(i, j)$
3. $\quad b[j] \leftarrow i$

Generic-Label-Correcting(s):
(b)
1. $c[s] \leftarrow 0, c[i \neq s] \leftarrow +\infty$
2. While any edge (i, j) satisfies $c[i] + cost(i, j) < c[j]$:
3. $\quad$ Tighten any such edge

Bellman-Ford(s):
1. $c[s] \leftarrow 0, c[i \neq s] \leftarrow +\infty$
(c)
2. Repeat $n - 1$ times:
3. $\quad$ Tighten every edge in the graph
4. If $c[i] + cost(i, j) < c[j]$ for any edge (i, j),
5. $\quad$ Output ``Negative-Cost Directed Cycle Detected''

Dijkstra(s):
1. $c[s] \leftarrow 0, c[i \neq s] \leftarrow +\infty$
(d)
2. Add all nodes to a priority queue Q, keyed on the $c[i]$'s
3. While Q is nonempty:
4. $\quad$ Remove from Q a node i having minimum label $c[i]$
5. $\quad$ Tighten all edges emanating from i

DAG-Shortest-Paths(s):
(e)
1. $c[s] \leftarrow 0, c[i \neq s] \leftarrow +\infty$
2. For all nodes i in topological order:
3. $\quad$ Tighten all edges emanating from i

FIGURE 16.4: Illustrations in pseudocode of our fundamental single-source shortest paths algorithms: (a) the edge tightening procedure, (b) the generic label-correcting algorithm, (c) the Bellman-Ford algorithm, and (d) Dijkstra's algorithm. For completeness, we also list in (e) the linear-time algorithm from Chapter 11 for computing single-source shortest paths in a DAG.

cost cycles in a graph in $O(mn)$ time. After making $n - 1$ passes over all edges in our graph, the algorithm should have reached an optimal solution. If we make one more pass over all of the edges and notice that cost labels are still being corrected, then we know our graph contains a negative-cost cycle, and with a small amount of extra work we can even identify the edges in such a cycle. [Details]

Problem 281 (Label Correcting with a FIFO Queue). In this problem we investigate an enhancement of the generic label-correcting algorithm. Most of the time of this algorithm is typically spent searching for edges that violate the triangle inequality. Let us try to reduce this search time by maintaining a set of nodes L such that we only need to check edges emanating from nodes in L. Initially $L = \{s\}$ since only edges emanating from the source can violate the triangle inequality. Any time we correct the label $c[i]$ of some node i, this introduces the possibility that edges from i may now also violate the triangle inequality, so any time this happens we must add i to L (if it is not in the list already). Although this extra data structure does help performance in practice, it is still

not sufficient to get rid of the exponential worst-case running time of the generic label-correcting algorithm. However, if we choose to implement L as a FIFO queue (i.e. any new edges are added to the end of L, and we always check edges emanating from the node at the front of L first), the running time can be shown to be $O(mn)$. The fact that this is the same as the running time of Bellman-Ford is no accident. Show that this "FIFO label-correcting algorithm" is really just performing a subset of the work that would be done by the Bellman-Ford algorithm. [Solution]

As a further note on the problem above, a very popular variant of the label correcting algorithm in practice uses not a FIFO queue, but a double-ended queue (deque). Any node being inserted into the queue is placed at the end, just as with the FIFO queue, unless the node has been in the queue before (we can remember this by flagging such nodes), in which case we insert it at the beginning of the queue, so it will now be the first node removed. The idea behind this heuristic is that nodes we have seen before should be processed with higher priority, since they may end up propagating label changes to many of the other nodes in our queue. While this approach unfortunately runs in exponential time in the worst case, computational experiments have shown it to run much faster than the FIFO variant in a wide range of common practical situations.

One final application of the Bellman-Ford algorithm we wish to mention is its use in solving another prominent graph problem, the *minimum average cost cycle* problem, which asks us to find a cycle in a directed graph whose average cost is as small as possible (note that negative-cost directed cycles are allowed in our graph for this problem, and that accordingly the *maximum* average cost cycle problem can be easily transformed into the minimum average cost cycle problem by negating all the edge costs). A subtle variation of the Bellman-Ford algorithm solves this problem in $O(mn)$ time [Details], and we will leverage this result later in Section 18.4 when we study algorithms for minimum cost circulation problems.

16.2.3 Nonnegative Edge Costs: Dijkstra's Algorithm

The Bellman-Ford algorithm (equivalently, the FIFO label correcting algorithm) gives us an $O(mn)$ running time for solving the single-source shortest path problem. Since this is too slow to be practical for large graphs, and since no faster algorithms are yet known (at least in the real RAM model, where we make no assumptions about our edge costs), we need to consider special cases of the single-source problem that we can solve more efficiently. In this section, we introduce *Dijkstra's algorithm*, a particularly well-known graph algorithm that solves the single source shortest path problem with nonnegative edge costs in nearly linear time. Since nonnegative edge costs are common, Dijkstra's algorithm is one of the most widely-implemented graph algorithms in practice.

We describe Dijkstra's algorithm in both a [short animated explanation] as well as in the following text. There are several nice ways to explain Dijkstra's algorithm; perhaps the simplest is as a generalized form of breadth-first search. Recall how breadth-first search visits the nodes in a graph in order of increasing distance from the source node. After examining the source (distance 0) and the edges emanating from it, it has built a list of distance-1 nodes to visit. After visiting these nodes and looking at their outgoing edges, it can determine the set of nodes that are at

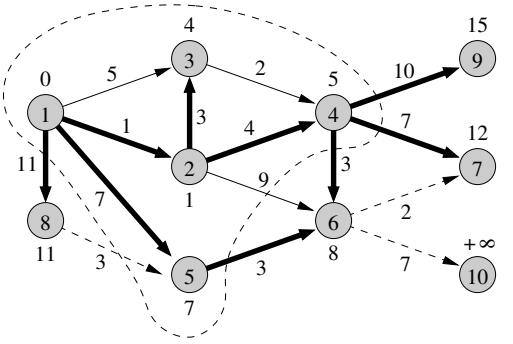

FIGURE 16.5: A snapshot in the middle of the execution of Dijkstra's algorithm, where only a subset of nodes have been visited. The source node is $s = 1$, and the remaining nodes are numbered in the order they will be visited. The shortest path tree computed thus far is highlighted, and dotted edges represent edges that have not yet been visited/tightened. The cost label of every node is shown, and the set of nodes visited so far has been outlined (we can think of this as having visited all nodes within "cost horizon" of 7; all yet-unvisited nodes will end up having shortest path cost at least this large). Node 6, with $c[8] = 9$, will be the next node to be visited, after which edges $(6, 7)$ and $(6, 10)$ will be tightened.

a distance of 2 from the source, and so on. Dijkstra's algorithm does roughly the same thing, visiting nodes "greedily" in order of shortest path cost from the source. In every iteration, Dijkstra's algorithm visits the yet-unvisited node with minimum cost label (say, node i) and tries to tighten i's outgoing edges.

Due to nonnegativity of edge costs, we will find that at the point in time node i is visited, its $c[i]$ cost label must represent the true cost of a shortest path to i. This means that we visit each node i only once, at which point we declare $c[i]$ to be correct. A snapshot of Dijkstra's algorithm in progress therefore looks like Figure 16.5: there is some "cost horizon" within which we have already visited all nodes and determined their correct shortest paths. The nodes outside have yet to be visited, and the cost label of every such node reflects the cost of the best path discovered to it so far through the set of previously-visited nodes. As the algorithm progresses, the cost horizon expands from initially just the source node until it encompasses the entire graph. Since Dijkstra's algorithm visits each node exactly once, it is sometimes called a *label setting* algorithm, in contrast to a label-correcting algorithm, which might visit a node and tighten its outgoing edges multiple times. [Correctness of Dijkstra's algorithm]

Pseudocode for Dijkstra's algorithm appears in Figure 16.4(d). We use a priority queue to maintain the set of unvisited nodes, keyed on their cost labels (you may want to review Chapter 5 for more detail on priority queues). Every iteration we call *remove-min* to extract the node in the queue with minimum cost label, and we tighten its outgoing edges. Tightening an edge (i, j) may potentially reduce the cost label $c[j]$, requiring that we call *decrease-key* to reduce j's key in the priority queue. We call *remove-min* n times, once for each node in sequence, and *decrease-key* at most m times, since we attempt to tighten each edge at most once (when we visit the node from which it emanates). Therefore, the running time of Dijkstra's

Priority Queue	*Remove-Min* ($O(n)$ times)	*Decrease-Key* ($O(m)$ times)	Total Runtime
Unsorted Array	$O(n)$	$O(1)$	$O(n^2)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$
B-Heap, $B = m/n$	$O(\frac{m}{n} \log_{m/n} n)$	$O(\log_{m/n} n)$	$O(m \log_{m/n} n)$
Fibonacci Heap	$O(\log n)$	$O(1)$ am.	$O(m + n \log n)$

FIGURE 16.6: Running time of Dijkstra's algorithm with different types of comparison-based priority queues. The first two columns give the running time for individual *remove-min* and *decrease-key* operations, and the last column shows the resulting runtime for Dijkstra's algorithm. The unsorted array is superior to a binary heap for dense graphs, while the binary heap is substantially better in sparse graphs. The B-heap with $B = m/n$ (see problem 80), scales gracefully between both of these extremes, since $\log_{m/n} n$ behaves like $\log n$ in a sparse graph and like n in a dense graph. The Fibonacci heap is the strongest choice, however, as it matches or outperforms all of these alternatives regardless of density.

algorithm is essentially determined by the running times of our priority queue operations. Several common choices for a priority queue, and the resulting running times for Dijkstra's algorithm, are listed in Figure 16.6. If you recall in Chapter 5 that we spent quite a bit of work with Fibonacci heaps to achieve $O(1)$ amortized running time of the *decrease-key* operation, Dijkstra's algorithm was one of the primary motivating factors for this work. As you will notice, the $O(m+n \log n)$ running time of Dijkstra's algorithm with a Fibonacci heap is almost linear except for the $O(n \log n)$ term. In the real RAM model, no implementation of Dijkstra's algorithm will be able to circumvent this particular term, since the algorithm visits its nodes in sorted order of cost from the source and therefore is a victim of the $\Omega(n \log n)$ lower bound for comparison-based sorting.

Problem 282 (Dijkstra's Algorithm on a Random Graph). In practice, a simple priority queue such as a binary heap or a b-heap is typically sufficient to ensure reasonable performance for Dijkstra's algorithm. These priority queues give asymptotic performance guarantees equivalent to Fibonacci heaps on sparse graphs (with $m = O(n)$), and in practice most very large graphs tend to be quite sparse. On dense graphs, although the Fibonacci heap is supposedly much better in theory, it can be surprisingly difficult to find instances of graphs for which the Fibonacci heap wins in practice. One of the reasons for this is that many large dense graphs tend to be sufficiently "well behaved" that they do not end up generating many calls to *decrease-key* during edge tightening, and this is the only place where Fibonacci heaps have the potential to give us any advantage in terms of added efficiency. To see a good example of this phenomenon, suppose we take any n-node graph and choose the cost of each edge independently from a common probability distribution. Please show that this input causes Dijkstra's algorithm to perform only $O(n \log n)$ *decrease-key* operations with high probability (so the running time with even a plain binary heap will be only $O(m + n \log^2 n)$ time with high probability, which is very close to the running time we get with a Fibonacci heap). This result should be carefully considered when conducting experimental comparisons between different shortest path algorithms. [Solution]

Priority Queue	*Remove-Min* ($O(n)$ times)	*Decrease-Key* ($O(m)$ times)	Total Runtime
Bucket List	$O(C)$	$O(1)$	$O(m + nC)$
Radix Tree	$O(\log C)$	$O(\log C)$	$O(m \log C)$
Radix Heap	$O(\log C)$	$O(1)$ am.	$O(m + n \log C)$
Improved Radix Heap	$O(\sqrt{\log C})$ am.	$O(1)$ am.	$O(m + n\sqrt{\log C})$
Y-fast tree (and (its relatives)	$O(\log \log C)$ am.	$O(\log \log C)$ am.	$O(m \log \log C)$

FIGURE 16.7: Running time of Dijkstra's algorithm with different types of integer priority queues. The "Bucket List" priority queue refers to the simple one-level bucket scheme described in Section 5.5.2, and the "Improved Radix Heap" refers to the result of problem 82. The bounds for the Y-fast tree and its relatives (e.g. stratified/van Emde Boas trees) are all in expectation. These structures are described in Section 7.4.2.

16.2.4 Nonnegative Integer Edge Costs

What if edge costs are nonnegative and also integral? By moving from the real RAM model to the more powerful RAM model of computation, we obtain potentially more "efficient" single-source shortest path algorithms with input-sensitive running times depending on a parameter C describing the magnitude of edge costs.

If edge costs are integers in the range $0 \ldots C$, we can now use our full host of integer priority queues (again, see Chapter 5 for more details) in combination with Dijkstra's algorithm. This gives us a wide variety of possible running times, as summarized in Figure 16.7. Observe that our running time is linear as long as $C = O(1)$, even using simple priority queue structures. If $C = O(n^k)$ with $k = O(1)$, then the improved radix heap gives us a running time of $O(m + n\sqrt{\log n})$, which improves on the $O(m + n \log n)$ running time we get with Fibonacci heaps.

Some of the fastest priority queues included in our summary are the monotone priority queues which we discussed at the end of Chapter 5. These priority queues assume that elements being removed from the queue form a monotonically nondecreasing sequence, and that no key within the queue will ever drop below that of an element already removed from the queue. Note that Dijkstra's algorithm does in fact satisfy this monotonicity property. Additionally, note that the running times in Figure 16.7 take into account the fact that priority queues used with Dijkstra's algorithm will be *range-bounded* (see problem 84). Although individual cost labels can take values as high as $(n-1)C$ (excluding the special case of infinity), the range of cost label values over the set of yet-unvisited nodes will never exceed C during the entire course of the algorithm, since these nodes lie within a cost of C from the cost horizon. As a result, the running times of our priority queue operations depend on C rather than nC.

Problem 283 (Shortest Paths with Lower-Bounded Integral Edge Costs). Consider the single-source shortest path problem with nonnegative edge costs. If we apply

Dijkstra's algorithm to solve this problem, we can achieve a variety of running times listed in Figure 16.7, depending on the type of integer priority queue we use. All of these running times depend somehow on C, the magnitude of the largest edge cost. If we know a lower bound C_{min} on edge costs, it turns out we can improve these running times. Please show how to reduce the running time of Dijkstra's algorithm so that it depends on C/C_{min} rather than C, irrespective of the priority queue we use. [Solution]

Problem 284 (Cost-Scaling Shortest Path Algorithms). Many problems with integer solutions admit solution algorithms known as "bit scaling algorithms", which build up solutions to successively larger problems by shifting in one bit at a time of each piece of problem data (in this case edge costs).

(a) Suppose we have solved a single-source shortest path problem with integer edge costs and computed the optimal cost label $c[i]$ for every node i. Now suppose that the costs of some edges in the graph increase by one. Show how to modify our solution in linear time so it remains optimal. [Solution]

(b) Building on your solution to (a), please design an $O(m \log C)$ bit-scaling algorithm for the single-source shortest path problem. Start with all edge costs equal to zero, and in every iteration, shift in a single bit of every edge cost, modifying the solution appropriately. [Solution]

16.2.5 Negative Integer Edge Costs

The preceding problem on cost-scaling algorithms gives a glimpse into another popular technique for designing shortest path algorithms. Scaling algorithms have also been successfully adapted to solve the single-source problem with integer and potentially negative edge costs. The strongest such algorithm yet known is *Goldberg's algorithm*, which runs in $O(m\sqrt{n} \log C^-)$ time, where C^- denotes the magnitude of the largest negative edge cost. This can be competitive with the Bellman-Ford algorithm if negative edge costs have reasonable magnitude. We will discuss the details of this algorithm in a moment in Section 16.4.2, once we learn about the concept of reduced edge costs.

16.3 All-Pairs Shortest Paths

In this section we switch gears and consider the all-pairs shortest path problem. The goal of this problem is to compute the shortest path cost $c[i, j]$ for every pair of nodes i and j in our graph. In order to be able to reconstruct these paths quickly, we may also want to compute for every (i, j) pair a backpointer $b[i, j]$ giving the penultimate node on the $i \rightsquigarrow j$ shortest path. For simplicity, let us focus only on path costs for now.

It will sometimes be convenient to think of the output of an all-pairs shortest path algorithm as just an $n \times n$ matrix C whose entries contain the shortest path costs $c[i, j]$ for every pair of nodes. Since our algorithms must always spend $\Theta(n^2)$ work filling in this matrix, there is usually no harm in assuming that our graph is represented in its "adjacency matrix" format — that is, in terms of an $n \times n$ matrix A whose (i, j) entry gives the cost of the edge from i to j, or infinity if there is no such edge in our graph (we put zeros on the diagonal, so there is an implicit zero-length edge from every node to itself). With the single-source problem, this


```
Floyd-Warshall:
1.  For k ← 1...n
2.      For i ← 1...n
3.          For j ← 1...n
4.              If c[i,k] + c[k,j] < c[i,j],
5.                  c[i,j] ← c[i,k] + c[k,j]
```

FIGURE 16.8: The remarkably simple Floyd-Warshall algorithm.

can be a terrible way to represent a graph if it is sparse, since it forces an algorithm to spend $\Theta(n^2)$ time just reading its input. However, for the all-pairs problem, it gives us a very clean way to formulate things: our algorithm will take as input the "edge cost" matrix A and produce as output the "shortest path cost" matrix C.

We discuss three methods for addressing the all-pairs problem in this book. Once we begin our discussion of matrices and linear algebra in Chapter 24, we will describe an elegant $O(n^3 \log n)$ algorithm based on a modified notion of matrix multiplication. This algorithm is not the fastest approach available for solving the all-pairs problem, but it highlights a useful general approach, and it also tends to parallelize better than our remaining approaches. In the next section, we describe the Floyd-Warshall algorithm, a breathtakingly simple $O(n^3)$ algorithm for solving the all-pairs problem in only 5 lines of code. After this, we introduce the notion of reduced edge costs, which leads us to Johnson's algorithm, running in $O(mn + n^2 \log n)$ time, which the best running time we currently know for the all-pairs problem in the real RAM model. All three algorithms accommodate negative edge costs, and can detect the presence of negative-cost directed cycles.

16.3.1 The Floyd-Warshall Algorithm

The *Floyd-Warshall* algorithm is so easy to remember and implement that the author often uses it to solve single-source shortest path problems on reasonably small graphs. Both its simplicity and $O(n^3)$ running time are evident from the pseudocode in Figure 16.8. As input to the algorithm, we set $C = A$; that is, we set $c[i, j]$ equal to the (i, j) entry in the edge cost matrix of our graph. After termination, the matrix C of $c[i, j]$ values will be the shortest path cost matrix. The Floyd-Warshall algorithm has no problem dealing with negative-cost edges, and can also detect negative-cost cycles: our graph has a negative cycle if a diagonal entry in C ever becomes negative, indicating a negative-cost path from a node back to itself[4]. Correctness of the algorithm is easily argued by interpreting it as a dynamic program. [Correctness argument]

Problem 285 (Arbitrage). This is a classic problem that is sure to appear in any textbook chapter on all-pairs shortest paths. Consider a set of n different currencies. You are given an $n \times n$ matrix containing the exchange rates between every pair of currencies.

[4]We should monitor the diagonal elements of C during the entire algorithm, breaking any time we find a negative value. If, instead, we wait until the end of the algorithm to check for a negative diagonal entry in C, it is possible for the magnitude of these negative entries to grow exponentially large, causing overflow problems in practice.

If r_{ij} denotes the exchange rate between currencies i and j, please do not assume necessarily that $r_{ij} = 1/r_{ji}$ (this is realistic, since banks often charge a small commission on currency transfers). Your goal is to determine whether you can make money by *arbitrage*: exchanging currencies until you eventually end up with more of your original currency than you started with. How can you formulate this as a shortest path problem? [Solution]

Problem 286 (Transitive Closure). The *transitive closure* of a directed graph G is a new graph G' such that there is an edge (i, j) in G' if and only if there is an $i \rightsquigarrow j$ path in G. Please describe a simple $O(mn)$ algorithm for solving this problem. In problem 482, we develop an alternative algorithm based on linear algebra that runs slightly faster in dense graphs. [Solution]

Problem 287 (Transitive Reduction). The *transitive reduction* of a directed graph G is a graph G' having the same transitive closure as G that has the smallest possible number of edges. Sometimes transitive reduction is defined so as to require G' to be a subgraph of G, but we will not impose this restriction since it makes some the problem NP-hard in graphs that are not DAGs. (see Problem 321).

(a) Prove that the transitive reduction of a DAG is unique, and that it must also be a subgraph of the original DAG. Give an $O(mn)$ algorithm to find the transitive reduction of a DAG. Hint: first compute the transitive closure. [Solution]

(b) Extend the algorithm from part (b) to compute the transitive reduction of a general directed graph in $O(mn)$ time. In this case, there may not be a unique transitive reduction, so your goal is to compute any transitive reduction. As a hint, think of the graph in terms of its strongly-connected components. [Solution]

16.4 Reduced Edge Costs

As we have seen, single-source shortest paths can be computed much more efficiently if all edges have nonnegative costs. Ideally, we would like to be able to take a problem involving negative-cost edges and transform it into one in which edge costs are all nonnegative. Many algorithms students have at one point in their careers been tempted to use the following trick: prior to computing shortest paths, simply increase all edges in a graph uniformly by a sufficiently large amount so as to ensure that all edge costs become nonnegative. It only takes a few moments of thought to convince oneself that this does not work at all! Whereas multiplicative scaling of edge costs preserves shortest paths, adding the same value to every edge cost can wreak havoc on the structure of shortest paths, since it penalizes paths according to the number of edges they contain[5].

Even though adding a constant to all edge costs does not accomplish what we want, it turns out that there is a way to re-weight the edges of a graph so that shortest paths are preserved and so that all edge costs become nonnegative. To employ this technique, we first solve a single-source shortest path problem for some arbitrary source node. This yields a set of shortest path cost labels that satisfy the triangle inequality, so for every edge (i, j) we have

$$c[i] + cost(i, j) \geq c[j].$$

[5]Note, however, that the addition of a constant to every edge cost is still useful for certain algorithms, such as the minimum average cost cycle problem we discussed back in Section 16.2.2.

We can rewrite this as

$$cost(i, j) + c[i] - c[j] \geq 0.$$

The quantity $cost(i, j)+c[i]-c[j]$ is called the *reduced cost* of the edge (i, j), and since the reduced cost of every edge is nonnegative, this is indeed a transformation that removes negative edge costs. There is one small subtlety that we must however address: we want to ensure that all nodes are reachable from the source node, otherwise some cost labels will be infinite, resulting in ill-defined reduced costs on some edges. To overcome this difficulty, we typically introduce a new dummy source node and connect it to every node with a zero-cost directed edge — this ensures that the cost label of every node is finite (in fact, at most zero), and it avoids introducing any unwanted negative cycles.

If we now measure the cost of paths and cycles in our graph using reduced costs, some very pleasant things happen. By adding up the reduced costs of all edges along a directed cycle, we obtain nothing more than the original cost of the cycle (since the $c[i]$'s and $c[j]$'s all cancel). Similarly, if we add the reduced costs of all edges along an $s \rightsquigarrow t$ path, we obtain the original cost of the path plus $c[s] - c[t]$ [Very short proofs]. However, since all $s \rightsquigarrow t$ paths are offset by the same additive quantity, the shortest $s \rightsquigarrow t$ path remains unchanged if we look at reduced edge costs instead of original edge costs. Therefore, if we compute shortest paths with respect to reduced costs, we will find the same paths as if we had used the original edge costs. The only difference is that the reduced cost of a shortest path from s to t will be offset from its actual value by $c[s] - c[t]$.

There are several nice intuitive ways to interpret reduced costs. If you recall the physical ball and string model of a graph where edges are represented by strings and shortest paths contain taught strings, reduced cost in this case measures the amount of "slack" on an edge. Tight edges (satisfying the triangle inequality with equality; these are the edges along shortest paths) will have zero reduced cost, and all other edges will have positive reduced cost. Along the same lines, reduced edge costs tell us the "sensitivity" of our shortest path costs to small changes in edge cost: if we lower the cost of an tight edge of reduced cost zero, this has an immediate impact on the cost of some shortest path. However, if we start with an edge of reduced cost 17, then we can lower its cost by 17 units before this has any impact on the cost of a shortest path from s to any other node. If you prefer an economic interpretation, think of $c[i]$ as the price for some commodity at node i, and think of the cost of an edge (i, j) as the cost for transporting that commodity from i to j. In this case, the reduced cost of (i, j) tells you the total cost of buying one unit of the commodity at i, transporting it to j, and selling it. The fact that all reduced costs are nonnegative means that commodity prices are in a sort of stable equilibrium — there is no incentive to engage in trade between nodes. This relates very closely to problem 285 on arbitrage.

16.4.1 Johnson's Algorithm for the All-Pairs Problem

For the all-pairs shortest path problem in which edge costs could potentially be negative, the Floyd-Warshall algorithm is our current champion, running in $O(n^3)$ time. Our next-best alternative, n invocations of the Bellman-Ford algorithm, would run in $O(n^2m)$ time, and this is much worse for dense graphs. Armed with reduced

costs, we can do much better. This technique, known as *Johnson's algorithm*, is relatively straightforward: run the Bellman-Ford algorithm for one source node[6], compute reduced costs, and then (since reduced costs are nonnegative) run Dijkstra's algorithm from every other source node. If we use the variant of Dijkstra's algorithm with a Fibonacci heap, the entire computation takes only $O(mn+n^2 \log n)$ time, which is always as good or better than the running time of the Floyd-Warshall algorithm for both sparse and dense graphs.

We will encounter similar techniques when we study algorithms for minimum cost network flow problems in Chapter 18. The "successive shortest path" algorithm for the minimum cost circulation problem performs a sequence of single-source shortest path computations, and we can speed these up by maintaining reduced costs so we can use Dijkstra's algorithm.

Problem 288 (Reoptimizing Shortest Paths). Suppose we have solved an instance of the single source shortest path problem (with potentially negative edge costs) and computed the shortest path cost label $c[i]$ for every node i. If there is a small change in some of the edge costs, we would like to *reoptimize* our solution (i.e. recompute shortest paths) more efficiently than by solving the problem again from scratch.

(a) If edge costs do not decrease, give an algorithm to reoptimize our solution in only $O(m + n \log n)$ time. [Solution]

(b) If at most one edge cost decreases, give an algorithm to reoptimize our solution in only $O(m + n \log n)$ time. If we have a single-source shortest path problem with only k negative-cost edges, we can use this approach to solve it in $O(km + kn \log n)$ time, outperforming the Bellman-Ford algorithm if k is sufficiently small. [Solution]

16.4.2 Goldberg's Algorithm

Let us return briefly to the subject of "scaling" algorithms and in particular Goldberg's algorithm. Recall that Goldberg's algorithm solves the single-source shortest path problem with potentially negative integral edge costs, and its running time is $O(m\sqrt{n} \log C^-)$ time, where C^- denotes the magnitude of the largest negative edge cost. As we learned in problem 284, a typical scaling algorithm operates in phases, where in each phase we shift in a single bit of every edge cost (thereby causing the cost of certain edges to increase by one), and then reoptimize our solution. In the case of Goldberg's algorithm, we perform scaling on just the negative-cost edges, so in each scaling phase we may potentially lower the cost of certain edges by one. In terms of reduced costs, we start at the beginning of each scaling phase with a set of cost labels with respect to which reduced edge costs are nonnegative. When we lower the cost of some edges by one unit, some of these edges may potentially acquire a reduced cost of -1.

The key subroutine in Goldberg's algorithm is an $O(m\sqrt{n})$ procedure for reoptimizing shortest paths in the event that edge costs are either nonnegative or -1. After $\log C^-$ scaling phases the algorithm consumes all of the "negativity" in the edge costs, leaving only a problem with nonnegative reduced edge costs that can be solved by Dijkstra's algorithm to complete the solution. Goldberg's algorithm is a

[6] As we mentioned before, we typically use an artificial source node connected to every other node by a zero-cost directed edge.

bit more involved than all of the shortest path algorithms we have seen so far, but the interested reader is encouraged to listen to its [details].

16.5 Point-to-Point Shortest Paths

In the event that we want to compute the shortest path from a single source s to a single destination t, we do not know of any algorithm whose worst-case running time is better than that of Dijkstra's more general single-source algorithm. However, there are several heuristics that tend to be quite effective for this variant. For starters, if we start Dijkstra's algorithm at s, then we can terminate it as soon as it labels t, thereby saving a large amount of work if s and t are close to each-other. We can also run a *bi-directional* variant of Dijkstra's algorithm where we simultaneously scan outward from s and t, stopping when these two procedures meet [Additional details]. As we see in Figure 16.9(a), we can think of Dijkstra's algorithm as searching radially outward from s and visiting a "ball" of nodes in our graph. Since two balls of radius $r/2$ cover half the area as one ball of radius r, we roughly expect the bi-directional variant (Figure 16.9(b)) to save us half the work of the unidirectional variant.

16.5.1 The A^* Heuristic

The so-called A^* heuristic improves the running time of Dijkstra's algorithm in practice by using reduced edge costs to penalize edges that point "away" from t and favor edges that point "toward" t. For example, if each node i is located at some point p_i in the two-dimensional plane where $d_i = ||p_i - p_t||$ is the distance from node i to node t, we would use $||p_i - p_j|| - d_i + d_j$ as the "reduced cost" of edge (i, j). If (i, j) points towards t, then its cost is reduced by this transformation, and if (i, j) points away from t, its cost is increased. Moreover, the geometric triangle inequality ensures that our reduced costs must be nonnegative, so Dijkstra's algorithm still applies. As shown in Figure 16.9(c), our reduced costs introduce a strong bias in Dijkstra's algorithm that can substantially reduce the total number of nodes examined.

The key to making the A^* heuristic work is the selection of a set of good "node potentials" (the d_i's, in this case) with respect to which we will obtain nonnegative reduced costs, that also cause our search to be appropriately biased. A good rule of thumb here is that we will obtain a feasible set of node potentials as long as we set the potential of node i to a quantity that is a lower bound on the cost of a shortest $i \rightsquigarrow t$ path. Note that this approach is more or less the same as the approach we discussed back in Section 13.3, where we used a lower bound on the distance to an optimal solution to prune our search space when implementing an exhaustive search. We can also run the A^* heuristic in a bi-directed fashion, although there are a few subtleties to address in this case when choosing node potentials. [Further mathematical details].

Instead of using just the destination node as a "reference point" when computing reduced costs for the A^* algorithm, we can often improve performance substantially in practice by using a small collection of designated reference nodes (sometimes

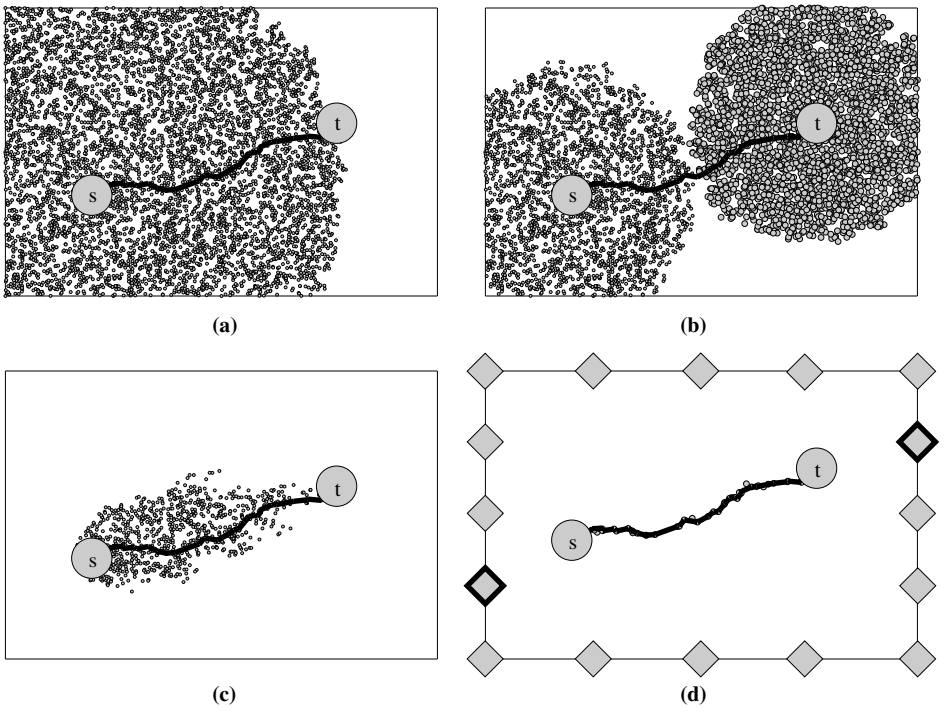

FIGURE 16.9: For this example, we have generated a random graph by placing 8000 nodes uniformly throughout a rectangle, randomly connecting pairs of nodes that are close together (with connection probability inverse exponential in distance) so that each node has between one and four neighbors. Edge costs are given by their Euclidean lengths. In each of the examples above, we compute a shortest s-t path, which is highlighted in bold, and we display only the nodes visited by Dijkstra's algorithm. In (a), Dijkstra's algorithm starts at s and terminates once it labels t, (b) depicts the bi-directional variant of Dijkstra's algorithm, (c) shows the result of the A^* heuristic starting from s and using Euclidean distance to t as a lower bound, and (d) shows the result of using multiple landmarks to guide the A^* search; there are 16 landmarks placed around the edge of the graph, and the two we have used are highlighted.

known as "landmarks" or "beacons") spaced out regularly throughout the graph, from which we have already precomputed shortest paths to all other nodes. Figure 16.9(d) shows an example where we have 16 beacons spread around the boundary of a planar graph. By using only two of these, a bi-directional A^* algorithm ends up examining less than 1% of the nodes in our graph! We could use more beacons if we so desired, but the two highlighted beacons in this case are sufficient since they lie more or less "in line" with the source and destination, so shortest paths in the directions of these beacons will be quite close to the shortest paths we seek. [Further details]

16.6 Relatives of the Shortest Path Problem

All of our shortest path algorithms rely on two arithmetic operations: one that is used to select which of two path costs is better (usually minimum), and one that is used to combine the costs of edges along a path (usually addition). By replacing these two operations with meaningful alternatives, we can trivially adapt any of the shortest path algorithms described earlier in this chapter (with no change in running time) to solve a host of related path-finding problems! For example, suppose we replace the (min, +) operations with:

- **(min, max).** In this case, we will end up solving the *bottleneck* shortest path problem, where a "shortest" path is a path whose maximum-cost edge is as small as possible. Remarkably, a (min, max) shortest path algorithm also allows us to solve the famous *minimum spanning tree* problem, as we will discover in problem 308 in the next chapter. In fact, the prominent minimum spanning tree algorithm known as Jarník's algorithm (and also as Prim's algorithm) is nothing more than Dijktra's algorithm modified for the (max, min) case. We will also see in problem 308 how to solve the all-pairs bottleneck shortest path problem in an undirected graph by computing a single minimum spanning tree. See also problem 296 for more on the bottleneck shortest path problem.
- **(max, min).** This case is symmetric to the bottleneck shortest path problem, and it allows us to find a path whose minimum-cost edge is as large as possible (and the same algorithmic comments above apply here, except using maximum spanning trees instead of minimum spanning trees). We will find this problem to be rather important when we study network flows, since by interpreting costs as capacities, this problem allows us to find a path through which one can pump the greatest amount of steady-state flow.
- **(max, $\times$).** Here, we end up solving the *maximum-reliability path problem*, introduced in problem 274, where we interpret the cost of an edge as a real number in $[0, 1]$ that gives its reliability probability (one minus its failure probability), and we seek paths with minimum failure probability.
- **(+, max).** In this case, we obtain an algorithm for computing a *maximum adjacency* (MA) ordering, which has applications in computing perfect elimination orderings for chordal graphs (Section 15.6.7) and also in "contraction"-based algorithms for computing global minimum cuts (Section 19.3.1). Note that the process of tightening edge (i, j) here *always* updates the cost label of node j, and that this cost label reflects the sum of the edge costs incoming to j from nodes we have previously processed; each iteration then selects a new node to process with maximum label.

Remember that at the heart of any shortest path algorithm is the quest to satisfy the triangle inequality: $c[i, j] + c[j, k] \geq c[i, k]$ for all triples (i, j, k) of nodes. By replacing addition and minimum with other alternatives, we are essentially building a shortest path algorithm that follows a different notion of "triangle inequality". For example, the triangle inequality for the bottleneck shortest path would require that $\max(c[i, j], c[j, k]) \geq c[i, k]$. [Precise mathematical conditions required to successfully modify a shortest path algorithm as described above]

16.7 Closing Remarks and Additional Problems

For the benefit of readers who are still interested in seeing more algorithms for shortest path problems, we will see in Chapter 22 that shortest paths in planar graphs with nonnegative edge costs can be computed in only $O(n)$ time, and we will investigate other types of "Euclidean" shortest path problems involving minimum-length paths through and around geometric objects. In addition, in Chapter 20 we will finally learn how to compute shortest paths in undirected graphs in the presence of negative edge costs.

For those who have read Chapter 12 on linear programming, we wish to mention a few brief remarks on some nice connections between shortest paths and linear programming. It may not come as much surprise that the shortest path problem can be written as a linear program. In fact, the generic label-correcting algorithm is nothing more than the simplex algorithm viewed in the context of solving this special type of linear program. Furthermore, our notion of the reduced cost of an edge corresponds exactly to the notion of the reduced cost of a variable in a linear program. In Chapter 12, we observed that a solution to a linear program is optimal only if all its variables have nonnegative reduced costs. Equivalently, the solution to a shortest path problem is optimal of all edges have nonnegative reduced costs (meaning that they all satisfy the triangle inequality). [More detailed discussion]

Problem 289 (Diameter of a Graph). The *diameter* of a graph is the maximum shortest path cost over all pairs of nodes in the graph. If we interpret shortest path costs as distances, graph diameter therefore gives the distance between the two nodes that are the farthest from each-other.

(a) Consider the following simple $O(n)$ algorithm for a tree T with edge costs: start at some arbitrary node x, compute the farthest node y from x, then compute the farthest node z from y. Please argue that the cost (distance) from y to z gives us the diameter of T. [Solution]

(b) Let us define the *edge unreliability* between nodes x and y in a graph as the number of edges e such that deletion of e would result in x and y belonging to different connected components. Similarly, we define the *node unreliability* between x and y as the number of nodes having this property. Building on your results from the previous question, please show how to find two nodes x and y having maximum edge or node unreliability in only linear time. [Solution]

(c) The diameter of a general graph can be computed by solving the all-pairs shortest path problem and taking the maximum over all path costs. However, this approach can be computationally expensive for large graphs. Show how to approximate the diameter of a graph to within a factor of 2 using only a small number of single-source shortest path computations. Try to design approaches that work for both undirected and directed graphs. You may want to consider first the undirected case, as it is slightly easier. Note that by definition, the diameter of a disconnected graph (or a non-strongly-connected graph, in the directed case) is infinite. [Solution]

Problem 290 (Centers and Medians of a Graph). The *eccentricity* of a node i in an undirected graph is the shortest path distance to the node farthest from i. A node having minimum eccentricity in a graph is known as a *center* (a graph might have several center nodes), and its eccentricity is known as the *radius* of the graph. We can find all center nodes and compute the radius of a graph exactly by solving the all-pairs shortest path problem.

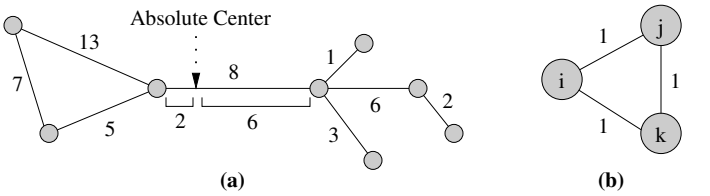

FIGURE 16.10: The absolute center of a graph is shown in (a). In (b), the edge eccentricity of node i is 1.5, since the furthest point from i is the middle of the edge from j to k.

(a) Devise an algorithm that finds the eccentricity of every node (and hence all centers and the radius) of a tree in linear time. Please also argue that an unweighted tree can have at most 2 centers. [Solution]

(b) The *absolute center* of a graph is a point of minimum eccentricity that is not constrained to be a node, and which might possibly lie "in the middle" of an edge, as shown in Figure 16.10(a). Can you find all absolute centers in a graph in $O(mn + n^2 \log n)$ time? [Solution]

(c) Let us define the *edge eccentricity* of a point in a graph (either a node, or a point in the "middle" of an edge (as in the absolute center problem) as the shortest distance to the farthest point on some edge in the graph. An example is shown in Figure 16.10(b). We can now ask for a center or absolute center of a graph with respect to edge eccentricity. Please discuss how to find all such locations in graph in polynomial time. Location problems involving sets of points along edges (not just their endpoint nodes), such as absolute center problems and problems involving edge eccentricities, are often called *continuous* location problems. [Solution]

(d) Please argue that the absolute center of a tree is unique, and that you can find it in $O(n)$ time. [Solution]

(e) Another common graph centrality measure is the *median* of a graph, a node from which the sum of shortest path distances to all other nodes is minimized. One can also generalize this notion to allow for consideration of points in the middle of edges in the same manner as we consider absolute centers and edge eccentricities above. Please show how to compute all medians of a graph in polynomial time, and all medians of a tree in linear time. [Solution]

When we study clustering algorithms in Chapter 19, we will revisit the notions of centers and medians in graphs, and further generalize these to obtain the k-center and k-median problems. The goal of the k-center problem is to partition a graph into k clusters so as to minimize the maximum cluster radius, and the goal of the k-median problem is to partition a graph into k clusters, each with an associated median node so as to minimize the sum of distances from all nodes to their respective cluster medians.

Problem 291 (Girth of a Graph). The *girth* of an undirected graph is the length of its shortest cycle. How can we compute the girth of a graph by solving an all-pairs shortest path problem? Next, suppose we are dealing with an unweighted graph, so the length of a cycle is just the number of edges in the cycle. For this case, show how to compute the girth of a graph exactly in $O(mn)$ time and to within an additive error of one in only $O(n^2)$ time. As a hint for the $O(n^2)$ algorithm, recall how depth-first search can detect cycles more efficiently in undirected than directed graphs. [Solution]

Problem 292 (Betweenness). *Betweenness* is a popular centrality measure that

captures the importance of an edge or node in terms of the extent to which it lies on shortest paths. That is, an edge or node with a large betweenness score will be a crucial member of many shortest paths in a graph, so it will generally lie in an important "central" location. Let n_{st} denote the total number of different shortest paths from s to t, and let $n_{st}(x)$ denote the number of shortest paths passing through x (where x can be either an edge or a node). Setting $\delta_{st}(x) = n_{st}(x)/n_{st}$, the betweenness of x is given by the sum of $\delta_{st}(x)$ over all pairs of nodes (s,t) (for the case where x is a node, we do not include pairs for which $s = x$ or $t = x$). Please show how to compute the betweenness of every node in a graph in $O(n^3)$ time, and of every edge in $O(mn^2)$ time. [Solution]

Problem 293 (Computing Shortcut Values). The *shortcut value* of an edge e joining nodes i and j is the cost of a shortest path from i to j when e is temporarily removed from the graph. Given a graph with nonnegative edge costs, it is easy to compute the shortcut values of all edges by running Dijsktra's algorithm m times, once with each edge removed. Show how, with a bit of cleverness, you can compute the shortcut values for all edges in running time proportional to only n times that of Dijkstra's algorithm. For simplicity, you may wish to assume all edges have strictly positive costs (although this assumption is not necessary to solve the problem). [Solution]

Problem 294 (Properties of Label-Correcting Algorithms). Earlier in this chapter we described the generic "label-correcting" algorithm for the single source shortest path problem: initialize the cost label of every node to infinity (except the source node, initialized to zero), and repeatedly locate and tighten any edge that violates the triangle inequality until no such edges exist and we have reached optimality. This is perhaps the simplest possible algorithm for the single-source shortest path problem, but its running time can be exponential in the worst case, as we showed earlier.

(a) If our graph has no negative cycles, the generic label-correcting algorithm must always terminate after a finite number of tightening operations. Just for fun, can you prove this fact? Moreover, can you prove that it will perform at most 2^n tightening operations on a graph with n nodes? [Solution]

(b) At the termination of a single-source shortest path algorithm, the predecessor pointers of our nodes form a shortest path tree directed into the source. Normally, we will never find a directed cycle amongst these predecessor pointers at any point during the execution of any label-correcting algorithm. Please argue that if we ever notice a directed cycle in the predecessor pointers when we run a label-correcting algorithm, then our graph must necessarily have a negative cycle. Must the cycle among the predecessor pointers be such a negative cycle? [Solution]

(c) Please argue that if we apply the generic label-correcting algorithm to a graph having a negative cycle, then after a finite number of tightening operations (in fact, at most 2^n if them) we must reach a state where the predecessor pointers contain a directed cycle. [Solution]

Problem 295 (Closest Pair of Designated Nodes). You are given an undirected graph with nonnegative edge costs, along with a subset of the nodes that are designated as "special". Show how to compute the closest pair of special nodes (in terms of shortest path cost) using only the same asymptotic running time as one invocation of Dijkstra's algorithm. [Solution]

Problem 296 (The Bottleneck Shortest Path Problem). The *bottleneck shortest path problem*, also known as the *fattest* or *thickest* path problem, involves finding paths emanating from a single source node that minimize the maximum-cost edge along the path (i.e. the bottleneck edge). Recall that we can solve the single-source bottleneck shortest path problem in $O(m + n \log n)$ time using a (min, max) variant of Dijkstra's algorithm, as discussed in Section 16.6. In fact, we will show in the next chapter (problem

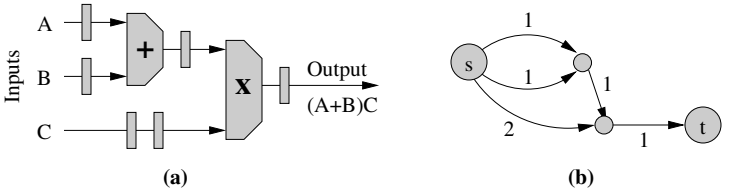

FIGURE 16.11: (a) A simple pipelined circuit with three inputs A, B, and C, that computes $(A + B)C$. In (b), the circuit is shown as a DAG where the label on each edge corresponds to the number of registers on a wire.

308) how to solve the all-pairs bottleneck shortest path problem in *undirected* graphs in linear expected time by computing a minimum spanning tree (since the path connecting two nodes through a minimum spanning tree is a bottleneck shortest path). In this exercise, please show how to find a bottleneck shortest path between two designated nodes s and t in an undirected graph in linear worst-case time. Hint: binary search for the smallest edge cost threshold that allows for connectivity between s and t. [Solution]

Problem 297 (Re-Timing a Parallel Network). In parallel computation, we often build up a circuit from wires (that carry values), *combinatorial logic* (boxes that add, multiply, or otherwise manipulate values on a set of input wires to produce a value on an output wire), and *registers*. All registers in the circuit will be aware of a global clock (for example on a 1 gigahertz machine, the global clock emits a tick every nanosecond). When the clock ticks, a register takes its current input value and passes it along to its output, holding this output value fixed until the next clock tick, when it passes along its next input value. It takes some amount of time for a signal to propagate through each piece of combinatorial logic, so the clock frequency must be long enough to allow for values to propagate between the longest register-to-register path in the circuit. Circuits are often *pipelined*, or divided up into stages so each stage can be put to use working on an independent problem. For example, in Figure 16.11(b) our circuit contains three stages, one which receives inputs A, B, and C, one which adds, and one which multiplies. At every clock tick the registers propagate values forward from each stage to the successive stage.

We can indicate the structure of a circuit as a directed graph, as shown in Figure 16.11, where nodes represent pieces of combinatorial logic and where each edge is labeled with the number of registers on the edge. We have also drawn in a fictitious source node s (from which inputs originate) and destination node t (which accepts outputs). Notices that as a consequence of pipelining, the length of every $s \rightsquigarrow i$ path must be the same, for every node i. Hence, in this graph the shortest path problem is easy to solve.

Circuits can often be easier to design if we don't need to worry about registers — that is, we only care about the layout of wires and combinatorial logic. After designing a circuit, however, we would like to place registers on wires so that (i) for every node i, every path $s \rightsquigarrow i$ contains the same number of registers, (ii) the number of registers on $s \rightsquigarrow t$ paths is as small as possible (so the computation takes a minimal number of phases), and (iii) each wire receives at least one register (so we can avoid long propagation delays through multiple pieces of combinatorial logic and run the clock a high frequency). How can we solve this register-placement problem in $O(m)$ time? [Solution]

Problem 298 (Shortest Paths Through a Time-Varying Network). Many networks exhibit time-varying characteristics, for example automobile networks around "rush hour". Suppose along with every edge (i, j) we are told an *arrival time function*,

$a_{ij}(t)$, such that if we enter the edge at node i time t, we will arrive at the other side at node j at time $a_{ij}(t)$. We assume that arrival time functions are strictly increasing — this is known as the *FIFO condition* since it requires that commodities exit from an edge in the same order as they enter.

(a) In the *single-source earliest arrival time problem*, we consider paths that depart from a given source node s at time $t = 0$, and we would like to compute for every other node i the earliest arrival time possible at node i, which we denote $A[i]$. Show how to modify the triangle inequality to obtain a set of optimality conditions for this problem. [Solution]

(b) Show how to modify Dijkstra's algorithm, without changing its asymptotic running time, to solve the earliest arrival time problem. [Solution]

(c) In the *single-destination latest departure time problem*, we are given a destination node d and we wish to know the latest possible time $D[i]$ we can depart from each node i so it is still possible to arrive at the destination by time $t = 0$. Show how to transform an instance of this problem into an equivalent instance of the earliest arrival time problem. [Solution]

Problem 299 (Finding a 3-Spanner of a Graph). A *k-spanner* of an undirected graph is a subgraph of its edges such that the shortest path cost between any two nodes in the subgraph is at most k times the shortest path cost between the same nodes in the original graph. A 1-spanner of a graph is just the graph itself, and in the worst case, a 2-spanner of a graph must also be the graph itself — this is true for any bipartite graph, since the next-shortest path between any two adjacent nodes has length 3. We therefore focus on k-spanners for $k \geq 3$, and in particular on 3-spanners for this problem. It is known that a 3-spanner of a graph must in the worst case contain $\Omega(n^{3/2})$ edges. This bound is actually achievable as well — it turns out that a simple randomized algorithm with linear expected running time can be used to compute a 3-spanner containing $O(n^{3/2})$ edges. Using this algorithm, we can take a dense graph and "sparsify" it to obtain fast approximation algorithms for the all-pairs shortest path problem. Whereas exact all-pairs shortest path algorithms run in $O(n^3)$ time on a dense graph, this approach gives an $O(n^{5/2})$ algorithm that computes shortest path costs between all pairs of nodes that are within a factor of 3 of optimal.

Here is how the algorithm works. We start by randomly sampling a set of $\sqrt{n}$ nodes. Keep all edges incident to these nodes. For every remaining non-sampled node i, we look at the sorted ordering of edges incident to i in terms of their costs, and keep only those from the minimum-cost edge up through the edge joining i to the earliest sampled node found in this ordering. If no sampled nodes appear in the ordering, we keep all of the edges. The other edges are marked for deletion. Since every node will remain connected to at most one sampled node, we can think of each sampled node accumulating a "cluster" of non-sampled nodes during this process. Finally, for every non-sampled node in the graph, we look at its incident edges and make sure that the minimum-cost edges joining it to each of these $\sqrt{n}$ clusters is not marked for deletion. We then delete all marked edges, and the remaining subgraph is our 3-spanner. Please argue that we can implement this algorithm to run in linear expected time, and that it produces a spanner having $O(n^{3/2})$ edges. [Solution]

Problem 300 (The Quickest Path Problem). An electronic communication network can be modeled by a directed graph where each edge has a latency and a capacity. Suppose we wish to send a message of size M across such a network from node s to node t. If we choose path P for this transmission, the total time required is the latency of P (the sum of latencies of all edges in P) plus M/b, where b is the bottleneck (i.e. minimum) edge capacity among all edges in P (which governs the rate at which packets can be transmitted

through P as a whole). Give an efficient algorithm that selects a path P for which total transmission time is minimized. [Solution]

Problem 301 (The Constrained Shortest Path Problem). In the *constrained shortest path problem*, edges in a graph carry two associated parameters: cost and travel time. Our goal is to compute a minimum-cost path subject to a constraint that the total travel time of the path must not exceed T. This problem is NP-hard, but we can approximate its solution to some extent. Let C^* denote cost of an optimal path. Show how to compute a path of cost at most C^* and travel time at most $2T$ in polynomial time. Can you improve your solution to find a path of travel time at most $(1 + \varepsilon)T$ for any constant $\varepsilon > 0$ in polynomial time? [Solution]

