24 Single-Source Shortest Paths

24.1 The Bellman-Ford algorithm

24.1-1

> Run the Bellman-Ford algorithm on the directed graph of Figure 24.4, using vertex z as the source. In each pass, relax edges in the same order as in the figure, and show the d and π values after each pass. Now, change the weight of edge (z, x) to 4 and run the algorithm again, using s as the source.

- Using vertex z as the source:
    - d values:

s	t	x	y	z
∞	∞	∞	∞	0
2	∞	7	∞	0
2	5	7	9	0
2	5	6	9	0
2	4	6	9	0

    - π values:

s	t	x	y	z
NIL	NIL	NIL	NIL	NIL
z	NIL	z	NIL	NIL
z	x	z	s	NIL
z	x	y	s	NIL
z	x	y	s	NIL

- Changing the weight of edge (z, x) to 4:
    - d values:

s	t	x	y	z
0	∞	∞	∞	∞
0	6	∞	7	∞
0	6	4	7	2
0	2	4	7	2
0	2	4	7	-2

    - π values:

s	t	x	y	z
NIL	NIL	NIL	NIL	NIL
NIL	s	NIL	s	NIL
NIL	s	y	s	t
NIL	x	y	s	t
NIL	x	y	s	t

Consider edge (z, x), it'll return FALSE since $x.d = 4 > z.d + w(z, x) = -2 + 4$.

24.1-2

> Prove Corollary 24.3.

Suppose there is a path from s to v. Then there must be a shortest such path of length $\delta(s, v)$. It must have finite length since it contains at most $|V| - 1$ edges and each edge has finite length. By Lemma 24.2, $v.d = \delta(s, v) < \infty$ upon termination.

On the other hand, suppose $v.d < \infty$ when BELLMAN-FORD terminates. Recall that $v.d$ is monotonically decreasing throughout the algorithm, and RELAX will update $v.d$ only if $u.d + w(u, v) < v.d$ for some u adjacent to v. Moreover, we update $v.\pi = u$ at this point, so v has an ancestor in the predecessor subgraph. Since this is a tree rooted at s, there must be a path from s to v in this tree. Every edge in the tree is also an edge in G, so there is also a path in G from s to v.

24.1-3

> Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v. (Here, the shortest path is by weight, not the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

By the upper bound theory, we know that after m iterations, no d values will ever change. Therefore, no d values will change in the $(m + 1)$-th iteration. However, we do not know the exact m value in advance, we cannot make the algorithm iterate exactly m times and then terminate. If we try to make the algorithm stop when every d values do not change anymore, then it will stop after $m + 1$ iterations.

24.1-4

> Modify the Bellman-Ford algorithm so that it sets $v.d$ to $-\infty$ for all vertices v for which there is a negative-weight cycle on some path from the source to v.

```
BELLMAN-FORD'(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for i = 1 to |G.V| - 1
        for each edge (u, v) ∈ G.E
            RELAX(u, v, w)
    for each edge(u, v) ∈ G.E
        if v.d > u.d + w(u, v)
            mark v
    for each vertex u ∈ marked vertices
        DFS-MARK(u)
```

```
DFS-MARK(u)
    if u ≠ NIL and u.d ≠ -∞
        u.d = -∞
        for each v in G.Adj[u]
            DFS-MARK(v)
```

After running $\text{BELLMAN-FORD}'$, run DFS with all vertices on negative-weight cycles as source vertices. All the vertices that can be reached from these vertices should have their d attributes set to $-\infty$.

24.1-5 ⋆

> Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$. Give an $O(VE)$-time algorithm to find, for each vertex $v \in V$, the value $\delta^*(v) = \min_{u \in V} \{\delta(u, v)\}$.

```
RELAX(u, v, w)
    if v.d > min(w(u, v), w(u, v) + u.d)
        v.d = min(w(u, v), w(u, v) + u.d)
        v.π = u.π
```

24.1-6 ⋆

> Suppose that a weighted, directed graph $G = (V, E)$ has a negative-weight cycle. Give an efficient algorithm to list the vertices of one such cycle. Prove that your algorithm is correct.

Based on exercise 24.1-4, DFS from a vertex u that $u.d = -\infty$, if the weight sum on the search path is negative and the next vertex is BLACK, then the search path forms a negative-weight cycle.

24.2 Single-source shortest paths in directed acyclic graphs

24.2-1

> Run $\text{DAG-SHORTEST-PATHS}$ on the directed graph of Figure 24.5, using vertex r as the source.

- d values:

r	s	t	x	y	z
0	∞	∞	∞	∞	∞
0	5	3	∞	∞	∞
0	5	3	11	∞	∞
0	5	3	10	7	5
0	5	3	10	7	5
0	5	3	10	7	5

- π values:

r	s	t	x	y	z
NIL	NIL	NIL	NIL	NIL	NIL
NIL	r	r	NIL	NIL	NIL
NIL	r	r	s	NIL	NIL
NIL	r	r	t	t	t
NIL	r	r	t	t	t
NIL	r	r	t	t	t

24.2-2

> Suppose we change line 3 of $\text{DAG-SHORTEST-PATHS}$ to read
>
> ```
> 3  for the first |V| - 1 vertices, taken in topologically sorted order
> ```
>
> Show that the procedure would remain correct.

When we reach vertex v, the last vertex in the topological sort, it must have $\text{out-degree}\ 0$. Otherwise there would be an edge pointing from a later vertex to an earlier vertex in the ordering, a contradiction. Thus, the body of the for-loop of line 4 is never entered for this final vertex, so we may as well not consider it.

24.2-3

> The PERT chart formulation given above is somewhat unnatural. In a more natural structure, vertices would represent jobs and edges would represent sequencing constraints; that is, edge (u, v) would indicate that job u must be performed before job v. We would then assign weights to vertices, not edges. Modify the

> DAG-SHORTEST-PATHS procedure so that it finds a longest path in a directed acyclic graph with weighted vertices in linear time.

There are two ways to transform a PERT chart $G = (V, E)$ with weights on the vertices to a PERT chart $G' = (V', E')$ with weights on edges. Both ways satisfy $|V'| \le 2|V|$ and $|E'| \le |V| + |E|$, so we can scan G' using the same algorithm to find the longest path through a directed acyclic graph.

In the first way, we transform each vertex $v \in V$ into two vertices v' and v'' in V'. All edges in E that enters V will also enter V' in E', and all edges in E that leaves V will leave V'' in E'. Thus, if $(u, v) \in E$, then $(u'', v') \in E'$. All such edges have weight 0, so we can put edges (v', v'') into E' for all vertices $v \in V$, and these edges are given the weight of the corresponding vertex v in G. Finally, we get $|V'| \le 2|V|$ and $|E'| \le |V| + |E|$, and the edge weight of each path in G' equals the vertex weight of the corresponding path in G.

In the second way, we leave vertices in V, but try to add one new source vertex s to V', given that $V' = V \cup \{s\}$. All edges of E are in E', and E' also includes an edge (s, v) for every vertex $v \in V$ that has in-degree 0 in G. Thus, the only vertex with in-degree 0 in G' is the new source s. The weight of edge $(u, v) \in E'$ is the weight of vertex v in G. We have the weight of each entering edge in G' is the weight of the vertex it enters in G.

24.2-4

> Give an efficient algorithm to count the total number of paths in a directed acyclic graph. Analyze your algorithm.

We will compute the total number of paths by counting the number of paths whose start point is at each vertex v, which will be stored in an attribute $v.\text{paths}$. Assume that initial we have $v.\text{paths} = 0$ for all $v \in V$. Since all vertices adjacent to u occur later in the topological sort and the final vertex has no neighbors, line 4 is well-defined. Topological sort takes $O(V + E)$ and the nested for-loops take $O(V + E)$ so the total runtime is $O(V + E)$.

```
PATHS(G)
    topologically sort the vertices of G
    for each vertex u, taken in topologically sorted order
        for each v ∈ G.Adj[u]
            v.paths = u.paths + 1 + v.paths
    return the sum of all paths attributes
```

24.3 Dijkstra's algorithm

24.3-1

> Run Dijkstra's algorithm on the directed graph of Figure 24.2, first using vertex s as the source and then using vertex z as the source. In the style of Figure 24.6, show the d and π values and the vertices in set S after each iteration of the **while** loop.

- s as the source:
  - d values:

s	t	x	y	z
0	3	∞	5	∞
0	3	9	5	∞
0	3	9	5	11
0	3	9	5	11
0	3	9	5	11

  - π values:

s	t	x	y	z
NIL	s	NIL	NIL	NIL
NIL	s	t	s	NIL
NIL	s	t	s	y
NIL	s	t	s	y
NIL	s	t	s	y

- z as the source:
    - d values:

s	t	x	y	z
3	∞	7	∞	0
3	6	7	8	0
3	6	7	8	0
3	6	7	8	0
3	6	7	8	0

    - π values:

s	t	x	y	z
z	NIL	z	NIL	NIL
z	s	z	s	NIL
z	s	z	s	NIL
z	s	z	s	NIL
z	s	z	s	NIL

24.3-2

> Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers. Why doesn't the proof of Theorem 24.6 go through when negative-weight edges are allowed?

Consider any graph with a negative cycle. RELAX is called a finite number of times but the distance to any vertex on the cycle is $-\infty$, so Dijkstra's algorithm cannot possibly be correct here. The proof of theorem 24.6 doesn't go through because we can no longer guarantee that

$$\delta(s, y) \le \delta(s, u).$$

24.3-3

> Suppose we change line 4 of Dijkstra's algorithm to the following.
>
> ```
> 4  while |Q| > 1
> ```
>
> This change causes the **while** loop to execute $|V| - 1$ times instead of $|V|$ times. Is this proposed algorithm correct?

Yes, the algorithm is correct. Let u be the leftover vertex that does not get extracted from the priority queue Q. If u is not reachable from s, then

$$u.d = \delta(s, u) = \infty.$$

If u is reachable from s, then there is a shortest path

$$p = s \to x \to u.$$

When the node x was extracted,

$$x.d = \delta(s, x)$$

and then the edge (x, u) was relaxed; thus,

$$u.d = \delta(s, u).$$

24.3-4

> Professor Gaedel has written a program that he claims implements Dijkstra's algorithm. The program produces $v.d$ and $v.\pi$ for each vertex $v \in V$. Give an $O(V + E)$-time algorithm to check the output of the professor's program. It should determine whether the d and π attributes match those of some shortest-paths tree. You may assume that all edge weights are nonnegative.

(Removed)

24.3-5

> Professor Newman thinks that he has worked out a simpler proof of correctness for Dijkstra's algorithm. He claims that Dijkstra's algorithm relaxes the edges of every shortest path in the graph in the order in which they appear on the path, and therefore the path-relaxation property applies to every vertex reachable from the source. Show that the professor is mistaken by constructing a directed graph for which Dijkstra's algorithm could relax the edges of a shortest path out of order.

(Removed)

24.3-6

> We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \le r(u, v) \le 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

(Removed)

24.3-7

> Let $G = (V, E)$ be a weighted, directed graph with positive weight function $w : E \to \{1, 2, \dots, W\}$ for some positive integer W, and assume that no two vertices have the same shortest-path weights from source vertex s. Now suppose that we define an unweighted, directed graph $G' = (V \cup V', E')$ by replacing each edge $(u, v) \in E$ with $w(u, v)$ unit-weight edges in series. How many vertices does G' have? Now suppose that we run a breadth-first search on G'. Show that the order in which the breadth-first search of G' colors vertices in V black is the same as the order in which Dijkstra's algorithm extracts the vertices of V from the priority queue when it runs on G.

$V + \sum_{(u,v) \in E} w(u, v) - E$.

24.3-8

> Let $G = (V, E)$ be a weighted, directed graph with nonnegative weight function $w : E \to \{0, 1, \dots, W\}$ for some nonnegative integer W. Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time.

(Removed)

24.3-9

> Modify your algorithm from Exercise 24.3-8 to run in $O((V + E) \lg W)$ time. (*Hint:* How many distinct shortest-path estimates can there be in $V - S$ at any point in time?)

(Removed)

24.3-10

> Suppose that we are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from s in this graph.

The proof of correctness, Theorem 24.6, goes through exactly as stated in the text. The key fact was that $\delta(s, y) \le \delta(s, u)$. It is claimed that this holds because there are no negative edge weights, but in fact that is stronger than is needed. This always holds if y occurs on a shortest path from s to u and $y \ne s$ because all edges on the path from y to u have nonnegative weight. If any had negative weight, this would imply that we had "gone back" to an edge incident with s, which implies that a cycle is involved in the path, which would only be the case if it were a negative-weight cycle. However, these are still forbidden.

24.4 Difference constraints and shortest paths

24.4-1

> Find a feasible solution or determine that no feasible solution exists for the following system of difference constraints:
>
> $$\begin{aligned} x_1 - x_2 &\le 1, \\ x_1 - x_4 &\le -4, \\ x_2 - x_3 &\le 2, \\ x_2 - x_5 &\le 7, \\ x_2 - x_6 &\le 5, \\ x_3 - x_6 &\le 10, \\ x_4 - x_2 &\le 2, \\ x_5 - x_1 &\le -1, \\ x_5 - x_4 &\le 3, \\ x_6 - x_3 &\le 8 \end{aligned}$$

Our vertices of the constraint graph will be

$$\{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}.$$

The edges will be

$$(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_0, v_4), (v_0, v_5), (v_0, v_6), (v_2, v_1), (v_4, v_1), (v_3, v_2), (v_5, v_2), (v_6, v_2), (v_6, v_3),$$

with edge weights

$$0, 0, 0, 0, 0, 0, 1, -4, 2, 7, 5, 10, 2, -1, 3, -8$$

respectively. Then, computing

$$(\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \delta(v_0, v_4), \delta(v_0, v_5), \delta(v_0, v_6)),$$

we get

$$(-5, -3, 0, -1, -6, -8),$$

which is a feasible solution by Theorem 24.9.

24.4-2

> Find a feasible solution or determine that no feasible solution exists for the following system of difference constraints:

> $$\begin{aligned} x_1 - x_2 &\le 4, \\ x_1 - x_5 &\le 5, \\ x_2 - x_4 &\le -6, \\ x_3 - x_2 &\le 1, \\ x_4 - x_1 &\le 3, \\ x_4 - x_3 &\le 5, \\ x_4 - x_5 &\le 10, \\ x_5 - x_3 &\le -4, \\ x_5 - x_4 &\le -8. \end{aligned}$$

There is no feasible solution because the constraint graph contains a negative-weight cycle: $(v_1, v_4, v_2, v_3, v_5, v_1)$ has weight -1.

24.4-3

> Can any shortest-path weight from the new vertex v_0 in a constraint graph be positive? Explain.

No, it cannot be positive. This is because for every vertex $v \ne v_0$, there is an edge (v_0, v) with weight zero. So, there is some path from the new vertex to every other of weight zero. Since $\delta(v_0, v)$ is a minimum weight of all paths, it cannot be greater than the weight of this weight zero path that consists of a single edge.

24.4-4

> Express the single-pair shortest-path problem as a linear program.

(Removed)

24.4-5

> Show how to modify the Bellman-Ford algorithm slightly so that when we use it to solve a system of difference constraints with m inequalities on n unknowns, the running time is $O(nm)$.

We can follow the advice of problem 14.4-7 and solve the system of constraints on a modified constraint graph in which there is no new vertex v_0. This is simply done by initializing all of the vertices to have a d value of 0 before running the iterated relaxations of Bellman Ford. Since we don't add a new vertex and the n edges going from it to to vertex corresponding to each variable, we are just running Bellman Ford on a graph with n vertices and m edges, and so it will have a runtime of $O(mn)$.

24.4-6

> Suppose that in addition to a system of difference constraints, we want to handle ***equality constraints*** of the form $x_i = x_j + b_k$. Show how to adapt the Bellman-Ford algorithm to solve this variety of constraint system.

To obtain the equality constraint $x_i = x_j + b_k$ we simply use the inequalities $x_i - x_j \le b_k$ and $x_j - x_i \le -bk$, then solve the problem as usual.

24.4-7

> Show how to solve a system of difference constraints by a Bellman-Ford-like algorithm that runs on a constraint graph without the extra vertex v_0.

(Removed)

24.4-8 ⋆

> Let $Ax \le b$ be a system of m difference constraints in n unknowns. Show that the Bellman-Ford algorithm, when run on the corresponding constraint graph, maximizes $\sum_{i=1}^n x_i$ subject to $Ax \le b$ and $x_i \le 0$ for all x_i.

Bellman-Ford correctly solves the system of difference constraints so $Ax \le b$ is always satisfied. We also have that $x_i = \delta(v_0, v_i) \le w(v_0, v_i) = 0$ so $x_i \le 0$ for all i. To show that $\sum x_i$ is maximized, we'll show that for any feasible solution

$(y_1, y_2, \dots, y_n)$ which satisfies the constraints we have $yi \le \delta(v_0, v_i) = x_i$. Let $v_0, v_{i_1}, \dots, v_{i_k}$ be a shortest path from v_0 to v_i in the constraint graph. Then we must have the constraints $y_{i_2} - y_{i_1} \le w(v_{i_1}, v_{i_2}), \dots, y_{i_k} - y_{i_{k-1}} \le w(v_{i_{k-1}}, v_{i_k})$. Summing these up we have

$$y_i \le y_i - y_1 \le \sum_{m=2}^{k} w(v_{i_m}, v_{i_{m-1}}) = \delta(v_0, v_i) = x_i .$$

24.4-9 ⋆

> Show that the Bellman-Ford algorithm, when run on the constraint graph for a system $Ax \le b$ of difference constraints, minimizes the quantity $(\max\{x_i\} - \min\{x_i\})$ subject to $Ax \le b$. Explain how this fact might come in handy if the algorithm is used to schedule construction jobs.

We can see that the Bellman-Ford algorithm run on the graph whose construction is described in this section causes the quantity $\max\{x_i\} - \min\{x_i\}$ to be minimized. We know that the largest value assigned to any of the vertices in the constraint graph is a 0. It is clear that it won't be greater than zero, since just the single edge path to each of the vertices has cost zero. We also know that we cannot have every vertex having a shortest path with negative weight. To see this, notice that this would mean that the pointer for each vertex has it's p value going to some other vertex that is not the source. This means that if we follow the procedure for reconstructing the shortest path for any of the vertices, we have that it can never get back to the source, a contradiction to the fact that it is a shortest path from the source to that vertex.

Next, we note that when we run Bellman-Ford, we are maximizing $\min\{x_i\}$. The shortest distance in the constraint graphs is the bare minimum of what is required in order to have all the constraints satisfied, if we were to increase any of the values we would be violating a constraint.

This could be in handy when scheduling construction jobs because the quantity $\max\{x_i\} - \min\{x_i\}$ is equal to the difference in time between the last task and the first task. Therefore, it means that minimizing it would mean that the total time that all the jobs takes is also minimized. And, most people want the entire process of construction to take as short of a time as possible.

24.4-10

> Suppose that every row in the matrix A of a linear program $Ax \le b$ corresponds to a difference constraint, a single-variable constraint of the form $x_i \le b_k$, or a singlevariable constraint of the form $-x_i \le b_k$. Show how to adapt the Bellman-Ford algorithm to solve this variety of constraint system.

(Removed)

24.4-11

> Give an efficient algorithm to solve a system $Ax \le b$ of difference constraints when all of the elements of b are real-valued and all of the unknowns x_i must be integers.

To do this, just take the floor of (largest integer that is less than or equal to) each of the b values and solve the resulting integer difference problem. These modified constraints will be admitting exactly the same set of assignments since we required that the solution have integer values assigned to the variables. This is because since the variables are integers, all of their differences will also be integers. For an integer to be less than or equal to a real number, it is necessary and sufficient for it to be less than or equal to the floor of that real number.

24.4-12 ⋆

> Give an efficient algorithm to solve a system $Ax \le b$ of difference constraints when all of the elements of b are real-valued and a specified subset of some, but not necessarily all, of the unknowns x_i must be integers.

To solve the problem of $Ax \le b$ where the elements of b are real-valued we carry out the same procedure as before, running Bellman-Ford, but allowing our edge weights to be real-valued. To impose the integer condition on the x_i's, we modify the RELAX procedure. Suppose we call $\text{RELAX}(v_i, v_j, w)$ where v_j is required to be integral valued. If $v_j.d > \lfloor v_i.d + w(v_i, v_j) \rfloor$, set $v_j.d = \lfloor v_i.d + w(v_i, v_j) \rfloor$. This guarantees that the condition that

$v_j.d - v_i.d \le w(v_i, v_j)$ as desired. It also ensures that v_j is integer valued. Since the triangle inequality still holds, $x = (v_1.d, v_2.d, \dots, v_n.d)$ is a feasible solution for the system, provided that G contains no negative weight cycles.

24.5 Proofs of shortest-paths properties

24.5-1

> Give two shortest-paths trees for the directed graph of Figure 24.2 (on page 648) other than the two shown.

Since the induced shortest path trees on $\{s, t, y\}$ and on $\{t, x, y, z\}$ are independent and have to possible configurations each, there are four total arising from that. So, we have the two not shown in the figure are the one consisting of the edges $\{(s, t), (s, y), (y, x), (x, z)\}$ and the one consisting of the edges $\{(s, t), (t, y), (t, x), (y, z)\}$.

24.5-2

> Give an example of a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ and source vertex s such that G satisfies the following property: For every edge $(u, v) \in E$, there is a shortest-paths tree rooted at s that contains (u, v) and another shortest-paths tree rooted at s that does not contain (u, v).

Let G have 3 vertices s, x, and y. Let the edges be (s, x), (s, y), and (x, y) with weights 1, 1, and 0 respectively. There are 3 possible trees on these vertices rooted at s, and each is a shortest paths tree which gives $\delta(s, x) = \delta(s, y) = 1$.

24.5-3

> Embellish the proof of Lemma 24.10 to handle cases in which shortest-path weights are ∞ or $-\infty$.

To modify Lemma 24.10 to allow for possible shortest path weights of ∞ and $-\infty$, we need to define our addition as $\infty + c = \infty$, and $-\infty + c = -\infty$. This will make the statement behave correctly, that is, we can take the shortest path from s to u and tack on the edge (u, v) to the end. That is, if there is a negative weight cycle on your way to u and there is an edge from u to v, there is a negative weight cycle on our way to v. Similarly, if we cannot reach v and there is an edge from u to v, we cannot reach u.

24.5-4

> Let $G = (V, E)$ be a weighted, directed graph with source vertex s, and let G be initialized by $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$. Prove that if a sequence of relaxation steps sets $s.\pi$ to a non-NIL value, then G contains a negative-weight cycle.

(Removed)

24.5-5

> Let $G = (V, E)$ be a weighted, directed graph with no negative-weight edges. Let $s \in V$ be the source vertex, and suppose that we allow $v.\pi$ to be the predecessor of v on *any* shortest path to v from source s if $v \in V - \{s\}$ is reachable from s, and NIL otherwise. Give an example of such a graph G and an assignment of π values that produces a cycle in G_π. (By Lemma 24.16, such an assignment cannot be produced by a sequence of relaxation steps.)

Suppose that we have a grap hon three vertices $\{s, u, v\}$ and containing edges $(s, u), (s, v), (u, v), (v, u)$ all with weight 0. Then, there is a shortest path from s to v of s, u, v and a shortest path from s to u of s v, u. Based off of these, we could set $v.\pi = u$ and $u.\pi = v$. This then means that there is a cycle consisting of u, v in G_π.

24.5-6

> Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$ and no negative-weight cycles. Let $s \in V$ be the source vertex, and let G be initialized by $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$. Prove that for every vertex $v \in V_\pi$, there exists a path from s to v in G_π and that this property is maintained as an invariant over any sequence of relaxations.

We will prove this by induction on the number of relaxations performed. For the base-case, we have just called $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$. The only vertex in V_π is s, and there is trivially a path from s to itself. Now suppose that after any sequence of n relaxations, for every vertex $v \in V_\pi$ there exists a path from s to v in G_π. Consider the $(n+1)$th relaxation. Suppose it is such that $v.d > u.d + w(u, v)$. When we relax v, we update $v.\pi = u.\pi$. By the induction hypothesis, there was a path from s to u in G_π. Now v is in V_π, and the path from s to u, followed by the edge $(u, v) = (v.\pi, v)$ is a path from s to v in G_π, so the claim holds.

24.5-7

> Let $G = (V, E)$ be a weighted, directed graph that contains no negative-weight cycles. Let $s \in V$ be the source vertex, and let G be initialized by $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$. Prove that there exists a sequence of $|V| - 1$ relaxation steps that produces $v.d = \delta(s, v)$ for all $v \in V$.

(Removed)

24.5-8

> Let G be an arbitrary weighted, directed graph with a negative-weight cycle reachable from the source vertex s. Show how to construct an infinite sequence of relaxations of the edges of G such that every relaxation causes a shortest-path estimate to change.

(Removed)

Problem 24-1 Yen's improvement to Bellman-Ford

> Suppose that we order the edge relaxations in each pass of the Bellman-Ford algorithm as follows. Before the first pass, we assign an arbitrary linear order $v_1, v_2, \ldots, v_{|V|}$ to the vertices of the input graph $G = (V, E)$. Then, we partition the edge set E into $E_f \cup E_b$, where $E_f = \{(v_i, v_j) \in E : i < j\}$ and $E_b = \{(v_i, v_j) \in E : i > j\}$. (Assume that G contains no self-loops, so that every edge is in either E_f or E_b.) Define $G_f = (V, E_f)$ and $G_b = (V, E_b)$.
>
> **a.** Prove that G_f is acyclic with topological sort $\langle v_1, v_2, \ldots, v_{|V|} \rangle$ and that G_b is acyclic with topological sort $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1 \rangle$.
>
> Suppose that we implement each pass of the Bellman-Ford algorithm in the following way. We visit each vertex in the order $v_1, v_2, \ldots, v_{|V|}$, relaxing edges of E_f that leave the vertex. We then visit each vertex in the order $v_{|V|}, v_{|V|-1}, \ldots, v_1$, relaxing edges of E_b that leave the vertex.
>
> **b.** Prove that with this scheme, if G contains no negative-weight cycles that are reachable from the source vertex s, then after only $\lceil |V|/2 \rceil$ passes over the edges, $v.d = \delta(s, v)$ for all vertices $v \in V$.
>
> **c.** Does this scheme improve the asymptotic running time of the Bellman-Ford algorithm?

a. Since in G_f edges only go from vertices with smaller index to vertices with greater index, there is no way that we could pick a vertex, and keep increasing it's index, and get back to having the index equal to what we started with. This means that G_f is acyclic. Similarly, there is no way to pick an index, keep decreasing it, and get back to the same vertex index. By these definitions, since G_f only has vertices going from lower indices to higher indices, $(v_1, \ldots, v_{|V|})$ is a topological ordering of the vertices. Similarly, for G_b, $(v_{|V|}, \ldots, v_1)$ is a topological ordering of the vertices.

b. Suppose that we are trying to find the shortest path from s to v. Then, list out the vertices of this shortest path $v_{k_1}, v_{k_2}, \ldots, v_{k_m}$. Then, we have that the number of times that the sequence $\{k_i\}_i$ goes from increasing to decreasing or from decreasing to increasing is the number of passes over the edges that are necessary to notice this path. This is because any increasing sequence of vertices will be captured in a pass through E_f and any decreasing sequence will be captured in a pass through E_b. Any sequence of integers of length $|V|$ can only change direction at most $\lfloor |V|/2 \rfloor$ times. However, we need to add one more in to account for the case that the source appears later in the ordering of the vertices than v_{k_2}, as it is in a sense initially expecting increasing vertex indices, as it runs through E_f before E_b.

c. It does not improve the asymptotic runtime of Bellman ford, it just drops the runtime from having a leading coefficient of 1 to a leading coefficient of $\frac{1}{2}$. Both in the original and in the modified version, the runtime is $O(EV)$.

Problem 24-2 Nesting boxes

A d-dimensional box with dimensions $(x_1, x_2, \ldots, x_d)$ ***nests*** within another box with dimensions $(y_1, y_2, \ldots, y_d)$ if there exists a permutation π on $\{1, 2, \ldots, d\}$ such that $x_{\pi(1)} < y_1$, $x_{\pi(2)} < y_2$, ..., $x_{\pi(d)} < y_d$.

a. Argue that the nesting relation is transitive.

b. Describe an efficient method to determine whether or not one d-dimensional box nests inside another.

c. Suppose that you are given a set of n d-dimensional boxes $\{B_1, B_2, \ldots, B_n\}$. Give an efficient algorithm to find the longest sequence $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k} \rangle$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1, 2, \ldots, k-1$. Express the running time of your algorithm in terms of n and d.

a. Suppose that box $x = (x_1, \ldots, x_d)$ nests with box $y = (y_1, \ldots, y_d)$ and box y nests with box $z = (z_1, \ldots, z_d)$. Then there exist permutations π and σ such that $x_{\pi(1)} < y_1, \ldots, x_{\pi(d)} < y_d$ and $y_{\sigma(1)} < z_1, \ldots, y_{\sigma(d)} < z_d$. This implies $x_{\pi(\sigma(1))} < z_1, \ldots, x_{\pi(\sigma(d))} < z_d$, so x nests with z and the nesting relation is transitive.

b. Box x nests inside box y if and only if the increasing sequence of dimensions of x is component-wise strictly less than the increasing sequence of dimensions of y. Thus, it will suffice to sort both sequences of dimensions and compare them. Sorting both length d sequences is done in $O(d \lg d)$, and comparing their elements is done in $O(d)$, so the total time is $O(d \lg d)$.

c. We will create a nesting-graph G with vertices $B_1, \ldots, B_n$ as follows. For each pair of boxes B_i, B_j, we decide if one nests inside the other. If B_i nests in B_j, draw an arrow from B_i to B_j. If B_j nests in B_i, draw an arrow from B_j to B_i. If neither nests, draw no arrow. To determine the arrows efficiently, after sorting each list of dimensions in $O(nd \lg d)$ we compair all pairs of boxes using the algorithm from part (b) in $O(n^2 d)$. By part (a), the resulted graph is acyclic, which allows us to easily find the longest chain in it in $O(n^2)$ in a bottom-up manner. This chain is our answer. Thus, the total time is $O(nd \max(\lg d, n))$.

Problem 24-3 Arbitrage

Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 49 Indian rupees, 1 Indian rupee buys 2 Japanese yen, and 1 Japanese yen buys 0.0107 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $49 \times 2 \times 0.0107 = 1.0486$ U.S. dollars, thus turning a profit of 4.86 percent.

Suppose that we are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j.

a. Give an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$

Analyze the running time of your algorithm.

b. Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

a. To do this we take the negative of the natural log (or any other base will also work) of all the values c_i that are on the edges between the currencies. Then, we detect the presence or absence of a negative weight cycle by applying Bellman Ford. To see that the existence of an arbitrage situation is equivalent to there being a negative weight cycle in the original graph, consider the following sequence of steps:

$$\begin{aligned} R[i_1, i_2] \cdot R[i_2, i_3] \cdot \cdots \cdot R[i_k, i_1] &> 1 \\ \ln(R[i_1, i_2]) + \ln(R[i_2, i_3]) + \cdots + \ln(R[i_k, i_1]) &> 0 \\ -\ln(R[i_1, i_2]) - \ln(R[i_2, i_3]) - \cdots - \ln(R[i_k, i_1]) &< 0. \end{aligned}$$

b. To do this, we first perform the same modification of all the edge weights as done in part (a) of this problem. Then, we wish to detect the negative weight cycle. To do this, we relax all the edges $|V| - 1$ many times, as in BellmanFord algorithm. Then, we record all of the d values of the vertices. Then, we relax all the edges $|V|$ more times. Then, we check to see which vertices had their d value decrease since we recorded them. All of these vertices must lie on some (possibly disjoint) set of negative weight cycles. Call S this set of vertices. To find one of these cycles in particular, we can pick any vertex in S and greedily keep picking any vertex that it has an edge to that is also in S. Then, we just keep an eye out for a repeat. This finds us our cycle. We know that we will never get to a dead end in this process because the set S consists of vertices that are in some union of cycles, and so every vertex has out degree at least 1.

Problem 24-4 Gabow's scaling algorithm for single-source shortest paths

A ***scaling*** algorithm solves a problem by initially considering only the highestorder bit of each relevant input value (such as an edge weight). It then refines the initial solution by looking at the two highest-order bits. It progressively looks at more and more high-order bits, refining the solution each time, until it has examined all bits and computed the correct solution.

In this problem, we examine an algorithm for computing the shortest paths from a single source by scaling edge weights. We are given a directed graph $G = (V, E)$ with nonnegative integer edge weights w. Let $W = \max_{(u,v)\in E}\{w(u,v)\}$. Our goal is to develop an algorithm that runs in $O(E \lg W)$ time. We assume that all vertices are reachable from the source.

The algorithm uncovers the bits in the binary representation of the edge weights one at a time, from the most significant bit to the least significant bit. Specifically, let $k = \lceil \lg(W+1) \rceil$ be the number of bits in the binary representation of W, and for $i = 1, 2, \ldots, k$, let $w_i(u,v) = \lfloor w(u,v)/2^{k-i} \rfloor$. That is, $w_i(u,v)$ is the "scaled-down" version of $w(u,v)$ given by the i most significant bits of $w(u,v)$. (Thus, $w_k(u,v) = w(u,v)$ for all $(u,v) \in E$.) For example, if $k = 5$ and $w(u,v) = 25$, which has the binary representation $\langle 11001 \rangle$, then $w_3(u,v) = \langle 110 \rangle = 6$. As another example with $k = 5$, if $w(u,v) = \langle 00100 \rangle = 4$, then $w_3(u,v) = \langle 001 \rangle = 1$. Let us define $\delta_i(u,v)$ as the shortest-path weight from vertex u to vertex v using weight function w_i. Thus, $\delta_k(u,v) = \delta(u,v)$ for all $u, v \in V$. For a given source vertex s, the scaling algorithm first computes the shortest-path weights $\delta_1(s,v)$ for all $v \in V$, then computes $\delta_2(s,v)$ for all $v \in V$, and so on, until it computes $\delta_k(s,v)$ for all $v \in V$. We assume throughout that $|E| \ge |V| - 1$, and we shall see that computing δ_i from δ_{i-1} takes $O(E)$ time, so that the entire algorithm takes $O(kE) = O(E \lg W)$ time.

a. Suppose that for all vertices $v \in V$, we have $\delta(s,v) \le |E|$. Show that we can compute $\delta(s,v)$ for all $v \in V$ in $O(E)$ time.

b. Show that we can compute $\delta_1(s,v)$ for all $v \in V$ in $O(E)$ time.

Let us now focus on computing δ_i from δ_{i-1}.

c. Prove that for $i = 2, 3, \ldots, k$, we have either $w_i(u,v) = 2w_{i-1}(u,v)$ or $w_i(u,v) = 2w_{i-1}(u,v) + 1$. Then, prove that

$$2\delta_{i-1}(s,v) \le \delta_i(s,v) \le 2\delta_{i-1}(s,v) + |V| - 1$$

for all $v \in V$.

d. Define for $i = 2, 3, \ldots, k$ and all $(u,v) \in E$,

$$\hat{w_i} = w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v).$$

Prove that for $i = 2, 3, \ldots, k$ and all $u, v \in V$, the "reweighted" value $\hat{w_i}(u,v)$ of edge (u,v) is a nonnegative integer.

e. Now, define $\hat{\delta_i}(s,v)$ as the shortest-path weight from s to v using the weight function $\hat{w_i}$. Prove that for $i = 2, 3, \ldots, k$ and all $v \in V$,

$$\delta_i(s,v) = \hat{\delta_i}(s,v) + 2\delta_{i-1}(s,v)$$

and that $\hat{\delta}_i(s, v) \leq |E|$.

f. Show how to compute $\delta_i(s, v)$ from $\delta_{i-1}(s, v)$ for all $v \in V$ in $O(E)$ time, and conclude that we can compute $\delta(s, v)$ for all $v \in V$ in $O(E \lg W)$ time.

a. We can do this in $O(E)$ by the algorithm described in exercise 24.3-8 since our "priority queue" takes on only integer values and is bounded in size by E.

b. We can do this in $O(E)$ by the algorithm described in exercise 24.3-8 since w takes values in $\{0, 1\}$ and $V = O(E)$.

c. If the ith digit, read from left to right, of $w(u, v)$ is 0, then $w_i(u, v) = 2w_{i-1}(u, v)$. If it is a 1, then $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Now let $s = v_0, v_1, \ldots, v_n = v$ be a shortest path from s to v under w_i. Note that any shortest path under w_i is necessarily also a shortest path under w_{i-1}. Then we have

$$\begin{aligned}
\delta_i(s, v) &= \sum_{m=1}^{n} w_i(v_{m-1}, v_m) \\
&\leq \sum_{m=1}^{n} [2w_{i-1}(u, v) + 1] \\
&\leq \sum_{m=1}^{n} w_{i-1}(u, v) + n \\
&\leq 2\delta_{i-1}(s, v) + |V| - 1.
\end{aligned}$$

On the other hand, we also have

$$\begin{aligned}
\delta_i(s, v) &= \sum_{m=1}^{n} w_i(v_{m-1}, v_m) \\
&\geq \sum_{m=1}^{n} 2w_{i-1}(v_{m-1}, v_m) \\
&\geq 2\delta_{i-1}(s, v).
\end{aligned}$$

d. Note that every quantity in the definition of $\hat{w}_i$ is an integer, so $\hat{w}_i$ is clearly an integer. Since $w_i(u, v) \geq 2w_{i-1}(u, v)$, it will suffice to show that $w_{i-1}(u, v) + \delta_{i-1}(s, u) \geq \delta_{i-1}(s, v)$ to prove nonnegativity. This follows immediately from the triangle inequality.

e. First note that $s = v_0, v_1, \ldots, v_n = v$ is a shortest path from s to v with respect to \hatw if and only if it is a shortest path with respect to w. Then we have

$$\begin{aligned}
\hat{\delta}_i(s, v) &= \sum_{m=1}^{n} w_i(v_{m-1}, v_m) + 2\delta_{i-1}(s, v_{m-1}) - 2\delta_{i-1}(s, v_m) \\
&= \sum_{m=1}^{n} w_i(v_{m-1}, v_m) - 2\delta_{i-1}(s, v_n) \\
&= \delta_i(s, v) - 2\delta_{i-1}(s, v).
\end{aligned}$$

f. By part (a) we can compute $\hat{\delta}_i(s, v)$ for all $v \in V$ in $O(E)$ time. If we have already computed $\delta_i - 1$ then we can compute δ_i in $O(E)$ time. Since we can compute δ_1 in $O(E)$ by part b, we can compute δ_i from scratch in $O(iE)$ time. Thus, we can compute $\delta = \delta_k$ in $O(Ek) = O(E \lg W)$ time.

Problem 24-5 Karp's minimum mean-weight cycle algorithm

Let $G = (V, E)$ be a directed graph with weight function $w : E \rightarrow \mathbb{R}$, and let $n = |V|$. We define the ***mean weight*** of a cycle $c = \langle e_1, e_2, \ldots, e_k \rangle$ of edges in E to be

$$\mu(c) = \frac{1}{k} \sum_{i=1}^{k} w(e_i).$$

Let $\mu^* = \min_c \mu(c)$, where c ranges over all directed cycles in G. We call a cycle c for which $\mu(c) = \mu^*$ a ***minimum mean-weight cycle***. This problem investigates an efficient algorithm for computing μ^*.

Assume without loss of generality that every vertex $v \in V$ is reachable from a source vertex $s \in V$. Let $\delta(s, v)$ be the weight of a shortest path from s to v, and let $\delta_k(s, v)$ be the weight of a shortest path from s to v consisting of *exactly* k edges. If there is no path from s to v with exactly k edges, then $\delta_k(s, v) = \infty$.

a. Show that if $\mu^* = 0$, then G contains no negative-weight cycles and $\delta(s, v) = \min_{0 \le k \le n-1} \delta_k(s, v)$ for all vertices $v \in V$.

b. Show that if $\mu^* = 0$, then

$$\max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \ge 0$$

for all vertices $v \in V$. $(\text{Hint:}$ Use both properties from part (a).)

c. Let c be a 0-weight cycle, and let u and v be any two vertices on c. Suppose that $\mu^* = 0$ and that the weight of the simple path from u to v along the cycle is x. Prove that $\delta(s, v) = \delta(s, u) + x$. $(\text{Hint:}$ The weight of the simple path from v to u along the cycle is $-x$.)

d. Show that if $\mu^* = 0$, then on each minimum mean-weight cycle there exists a vertex v such that

$$\max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

$(\text{Hint:}$ Show how to extend a shortest path to any vertex on a minimum meanweight cycle along the cycle to make a shortest path to the next vertex on the cycle.)

e. Show that if $\mu^* = 0$, then

$$\min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

f. Show that if we add a constant t to the weight of each edge of G, then μ^* increases by t. Use this fact to show that

$$\mu^* = \min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k}.$$

g. Give an $O(VE)$-time algorithm to compute μ^*.

a. If $\mu^* = 0$, then we have that the lowest that $\frac{1}{k}\sum_{i=1}^{k} w(e_i)$ can be zero. This means that the lowest $\sum_{i=1}^{k} w(e_i)$ can be 0. This means that no cycle can have negative weight. Also, we know that for any path from s to v, we can make it simple by removing any cycles that occur. This means that it had a weight equal to some path that has at most $n - 1$ edges in it. Since we take the minimum over all possible number of edges, we have the minimum over all paths.

b. To show that

$$\max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \ge 0,$$

we need to show that

$$\max_{0 \le k \le n-1} \delta_n(s, v) - \delta_k(s, v) \ge 0.$$

Since we have that $\mu^* = 0$, there aren't any negative weight cycles. This means that we can't have the minimum cost of a path decrease as we increase the possible length of the path past $n - 1$. This means that there will be a path that at least ties for cheapest when we restrict to the path being less than length n. Note that there may also be cheapest path of longer length since we necessarily do have zero cost cycles. However, this isn't guaranteed since the zero cost cycle may not lie along a cheapest path from s to v.

c. Since the total cost of the cycle is 0, and one part of it has cost x, in order to balance that out, the weight of the rest of the cycle has to be $-x$. So, suppose we have some shortest length path from s to u, then, we could traverse the path from u to v along the cycle to get a path from s to u that has length $\delta(s,u)+x$. This gets us that $\delta(s,v)\le\delta(s,u)+x$.

To see the converse inequality, suppose that we have some shortest length path from s to v. Then, we can traverse the cycle going from v to u. We already said that this part of the cycle had total cost $-x$. This gets us that $\delta(s,u)\le\delta(s,v)-x$. Or, rearranging, we have $\delta(s,u)+x\le\delta(s,v)$. Since we have inequalities both ways, we must have equality.

d. To see this, we find a vertex v and natural number $k\le n-1$ so that $\delta_n(s,v)-\delta_k(s,v)=0$. To do this, we will first take any shortest length, smallest number of edges path from s to any vertex on the cycle. Then, we will just keep on walking around the cycle until we've walked along n edges. Whatever vertex we end up on at that point will be our v. Since we did not change the d value of v after looking at length n paths, by part (a), we know that there was some length of this path, say k, which had the same cost. That is, we have $\delta_n(s,v)=\delta_k(s,v)$.

e. This is an immediate result of the previous problem and part (b). Part (a) says that the inequality holds for all v, so, we have

$$\min_{v\in V}\max_{0\le k\le n-1}\frac{\delta_n(s,v)-\delta(s,v)}{n-k}\ge 0.$$

The previous part says that there is some v on each minimum weight cycle so that

$$\max_{0\le k\le n-1}\frac{\delta_n(s,v)-\delta(s,v)}{n-k}=0,$$

which means that

$$\min_{v\in V}\max_{0\le k\le n-1}\frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}\le 0.$$

Putting the two inequalities together, we have the desired equality.

f. If we add t to the weight of each edge, the mean weight of any cycle becomes

$$\mu(c)=\frac{1}{k}\sum_{i=1}^{k}(w(e_i)+t)=\frac{1}{k}\Big(\sum_{i}^{k}w(e_i)\Big)+\frac{kt}{k}=\frac{1}{k}\Big(\sum_{i}^{k}w(e_i)\Big)+t.$$

This is the original, unmodified mean weight cycle, plus t. Since this is how the mean weight of every cycle is changed, the lowest mean weight cycle stays the lowest mean weight cycle. This means that μ^* will increase by t. Suppose that we first compute μ^*. Then, we subtract from every edge weight the value μ^*. This will make the new μ^* equal zero, which by part (e) means that

$$\min_{v\in V}\max_{0\le k\le n-1}\frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}=0.$$

Since they are both equal to zero, they are both equal to each other.

g. By the previous part, it suffices to compute the expression on the previous line. We will start by creating a table that lists $\delta_k(s,v)$ for every $k\in\{1,\dots,n\}$ and $v\in V$. This can be done in time $O(V(E+V))$ by creating a $|V|$ by $|V|$ table, where the kth row and vth column represent $\delta_k(s,v)$ when wanting to compute a particular entry, we need look at a number of entries in the previous row equal to the in degree of the vertex we want to compute.

So, summing over the computation required for each row, we need $O(E+V)$. Note that this total runtime can be bumped down to $O(VE)$ by not including in the table any isolated vertices, this will ensure that $E\in\Omega(V)$. So, $O(V(E+V))$ becomes $O(VE)$. Once we have this table of values computed, it is simple to just replace each row with the last row minus what it was, and divide each entry by $n-k$, then, find the min column in each row, and take the max of those numbers.

Problem 24-6 Bitonic shortest paths

> A sequence is ***bitonic*** if it monotonically increases and then monotonically decreases, or if by a circular shift it monotonically increases and then monotonically decreases. For example the sequences $\langle 1, 4, 6, 8, 3, -2 \rangle$, $\langle 9, 2, -4, -10, -5 \rangle$, and $\langle 1, 2, 3, 4 \rangle$ are bitonic, but $\langle 1, 3, 12, 4, 2, 10 \rangle$ is not bitonic. (See Problem 15-3 for the bitonic euclidean traveling-salesman problem.)
>
> Suppose that we are given a directed graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$, where all edge weights are unique, and we wish to find single-source shortest paths from a source vertex s. We are given one additional piece of information: for each vertex $v \in V$, the weights of the edges along any shortest path from s to v form a bitonic sequence.
>
> Give the most efficient algorithm you can to solve this problem, and analyze its running time.

We'll use the Bellman-Ford algorithm, but with a careful choice of the order in which we relax the edges in order to perform a smaller number of RELAX operations. In any bitonic path there can be at most two distinct increasing sequences of edge weights, and similarly at most two distinct decreasing sequences of edge weights. Thus, by the path-relaxation property, if we relax the edges in order of increasing weight then decreasing weight twice (for a total of four times relaxing every edge) the we are guaranteed that $v.d$ will equal $\delta(s, v)$ for all $v \in V$. Sorting the edges takes $O(E \lg E)$. We relax every edge 4 times, taking $O(E)$. Thus the total runtime is $O(E \lg E) + O(E) = O(E \lg E)$, which is asymptotically faster than the usual $O(VE)$ runtime of Bellman-Ford.