Chapter 24

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 24.1-1

If we change our source to z and use the same ordering of edges to decide what to relax, the d values after successive iterations of relaxation are:

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

The π values are:

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

Now, if we change the weight of edge (z, x) to 4 and rerun with s as the source, we have that the d values after successive iterations of relaxation are:

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

The π values are:

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

Note that these values are exactly the same as in the worked example. The difference that changing this edge will cause is that there is now a negative weight cycle, which will be detected when it considers the edge (z, x) in the for loop on line 5. Since $x.d = 4 > -2 + 4 = z.d + w(z, x)$, it will return false on line 7.

Exercise 24.1-2

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.

Exercise 24.1-3

Before each iteration of the for loop on line 2, we make a backup copy of the current d values for all the vertices. Then, after each iteration, we check to see if any of the d values changed. If none did, then we immediately terminate the for loop. This clearly works because if one iteration didn't change the values of d, nothing will of changed on later iterations, and so they would all proceed to not change any of the d values.

Exercise 24.1-4

If there is a negative weight cycle on some path from s to v such that u immediately preceeds v on the path, then $v.d$ will strictly decrease every time RELAX(u, v, w) is called. If there is no negative weight cycle, then $v.d$ can never decrease after lines 1 through 4 are executed. Thus, we just update all vertices v which satisfy the if-condition of line 6. In particular, replace line 7 with $v.d = -\infty$.

Exercise 24.1-5

Initially, we will make each vertex have a D value of 0, which corresponds to taking a path of length zero starting at that vertex. Then, we relax along each edge exactly $V - 1$ times. Then, we do one final round of relaxation, which if any thing changes, indicated the existence of a negative weight cycle. The code for this algorithm is identical to that for Bellman ford, except instead of initializing the values to be infinity except at the source which is zero, we initialize every d value to be infinity. We can even recover the path of minimum length for each

vertex by looking at their π values.

Note that this solution assumes that paths of length zero are acceptable. If they are not to your liking then just initialize each vertex to have a d value equal to the minimum weight edge that they have adjacent to them.

Exercise 24.1-6

Begin by calling a slightly modified version of DFS, where we maintain the attribute $v.d$ at each vertex which gives the weight of the unique simple path from s to v in the DFS tree. However, once $v.d$ is set for the first time we will never modify it. It is easy to update DFS to keep track of this without changing its runtime. At first sight of a back edge (u, v), if $v.d > u.d + w(u, v)$ then we must have a negative-weight cycle because $u.d + w(u, v) - v.d$ represents the weight of the cycle which the back edge completes in the DFS tree. To print out the vertices print v, u, $u.\pi$, $u.\pi.\pi$, and so on until v is reached. This has runtime $O(V + E)$.

Exercise 24.2-1

If we run the procedure on the DAG given in figure 24.5, but start at vertex r, we have that the d values after successive iterations of relaxation are:

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

The π values are:

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

Exercise 24.2-2

When we reach vertex v, the last vertex in the topological sort, it must have 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.

Exercise 24.2-3

Introduce two new dummy tasks, with cost zero. The first one having edges going to every task that has no in edges. The second having an edge going to it from every task that has no out edges. Now, construct a new directed graph in which each e gets mapped to a vertex v_e and there is an edge $(v_e, v_{e'})$ with cost w if and only if edge e goes to the same vertex that e' comes from, and that vertex has weight w. Then, every path through this dual graph corresponds to a path through the original graph. So, we just look for the most expensive path in this DAG which has weighted edges using the algorithm from this section.

Exercise 24.2-4

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 $v.paths$. Assume that initial we have $v.paths = 0$ 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 $O(V + E)$.

Algorithm 1 PATHS(G)

```
1: topologically sort the vertices of G
2: for each vertex u, taken in reverse topologically sorted order do
3:     for each vertex v ∈ G.Adj[u] do
4:         u.paths = u.paths + 1 + v.paths
5:     end for
6: end for
```

Exercise 24.3-1

We first have s as the source, in this case, the sequence of extractions from the priority queue are: s, t, y,x,z. The d values after each iteration are:

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

The π values are:

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

Now, if we repeat the procedure, except having z as the source, we have that the d values are

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

The π values are:

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

Exercise 24.3-2

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

Exercise 24.3-3

It does work correctly to modify the algorithm like that. Once we are at the point of considering the last vertex, we know that it's current d value is at at least as large as the largest of the other vertices. Since none of the edge weights are negative, its d value plus the weight of any edge coming out of it will be at least as large as the d values of all the other vertices. This means that the relaxations that occur will not change any of the d values of any vertices, and so not change their π values.

Exercise 24.3-4

Check that $s.d = 0$. Then for each vertex in $V \backslash \{s\}$, examine all edges coming into V. Check that $v.\pi$ is the minimum of $u.d + w(u, v)$ for all vertices u for

which there is an edge (u, v), and that $v.d = v.\pi.d + w(v.\pi, v)$. If this is ever false, return false. Otherwise, return true. This takes $O(V + E)$ time. Now we must check that this correctly checks whether or not the d and π attributes match those of some shortest-paths tree. Suppose that this is not true. Let v be the vertex of smallest $v.d$ value which is incorrect. We may assume that $v \neq s$ since we check correctness of $s.d$ explicitly. Since all edge weights are nonnegative, v must be preceded by a vertex of smaller estimated distance, which we know to be correct since v has the smallest incorrect estimated distance. By verifying that $v.\pi$ is in fact the vertex which minimizes the distance of v, we have ensured that $v.\pi$ is correct, and by checking that $v.d = v.\pi.d + w(v.\pi, v)$ we ensure that the computation of $v.d$ is accurate. Thus, if there is a vertex which has the wrong estimated distance or parent, we will find it.

Exercise 24.3-5

Consider the graph on 5 vertices $\{a, b, c, d, e\}$, and with edges $(a, b), (b, c), (c, d), (a, e), (e, c)$ all with weight 0. Then, we could pull vertices off of the queue in the ordera, e, c, b, d. This would mean that we relax (c, d) before (b, c). However, a shortest pat to d is $(a, b), (b, c), (c, d)$. So, we would be relaxing an edge that appears later on this shortest path before an edge that appears earlier.

Exercise 24.3-6

We now view the weight of a path as the reliability of a path, and it is computed by taking the product of the reliabilities of the edges on the path. Our algorithm will be similar to that of DIJKSTRA, and have the same runtime, but we now wish to maximize weight, and RELAX will be done inline by checking products instead of sums, and switching the inequality since we want to maximize reliability. Finally, we track that path from y back to x and print the vertices as we go.

Exercise 24.3-7

Each edge is replaced with a number of edges equal to its weight, and one less than that many vertices. That is, $|V'| = \sum_{(v,u)\in E} w(v, u) - 1$. Similarly, $|E'| = \sum_{(v,u)\in E} w(v, u)$. Since we can bound each of these weights by W, we can say that $|V'| \leq W|E| - |E|$ so there are at most $W|E| - |E| + |V|$ vertices in G'. A breadth first search considers vertices in an order so that u and v satisfy $u.d < v.d$ it considers u before v. Similarly, since each iteration of the while loop in Dijkstra's algorithm considers the vertex with lowest d value in the queue, we will also be considering vertices with smaller d values first. So, the two order of considering vertices coincide.

Exercise 24.3-8

Algorithm 2 RELIABILITY(G, r, x, y)

```
1: INITIALIZE-SINGLE-SOURCE(G, x)
2: S = ∅
3: Q = G.V
4: while Q ≠ ∅ do
5:     u = EXTRACT-MIN(Q)
6:     S = S ∪ {u}
7:     for each vertex v ∈ G.Adj[u] do
8:         if v.d < u.d · r(u, v) then
9:             v.d = u.d · r(u, v)
10:            v.π = u
11:        end if
12:    end for
13: end while
14: while y ≠ x do
15:    Print y
16:    y = y.π
17: end while
18: Print x
```

We will modify Dijkstra's algorithm to run on this graph by changing the way the priority queue works, taking advantage of the fact that its members will have keys in the range $[0, WV] \cup \{\infty\}$, since in the worst case we have to compute the distance from one end of a chain to the other, with $|V|$ vertices each connected by an edge of weight W. In particular, we will create an array A such that $A[i]$ holds a linked list of all vertices whose estimated distance from s is i. We'll also need the attribute $u.list$ for each vertex u, which points to u's spot in the list stored at $A[u.d]$. Since the minimum distance is always increasing, k will be at most VW, so the algorithm spends $O(VW)$ time in the while loop on line 9, over all iterations of the for-loop of line 8. We spend only $O(V + E)$ time executing the for-loop of line 14 because we look through each adjacency list only once. All other operations take $O(1)$ time, so the total runtime of the algorithm is $O(VW) + O(V + E) = O(VW + E)$.

Exercise 24.3-9

We can modify the construction given in the previous exercise to avoid having to do a linear search through the array of lists A. To do this, we can just keep a set of the indices of A that contain a non-empty list. Then, we can just maintain this as we move the vertices between lists, and replace the while loop in the previous algorithm with just getting the minimum element in the set.

One way of implementing this set structure is with a self-balancing binary search tree. Since the set consists entirely of indices of A which has length W, the size of the tree is at most W. We can find the minimum element, delete an element and insert in element all in time $O(\lg(W))$ in a self balancing binary

Algorithm 3 MODIFIED-DIJKSTRA(G,w,s)

1: **for** each $v \in G.V$ **do** $v.d = VW + 1$ $v.\pi = NIL$
2: **end for**
3: $s.d = 0$
4: Initialize an array A of length $VW + 2$
5: $A[0].insert(s)$
6: Set $A[VW + 1]$ equal to a linked list containing every vertex except s
7: $k = 0$
8: **for** $i = 1$ to $|V|$ **do**
9: **while** $A[k] = NIL$ **do**
10: $k = k + 1$
11: **end while**
12: $u = A[k].head$
13: $A[k].delete(u)$
14: **for** each vertex $v \in G.Adj[u]$ **do**
15: **if** $v.d > u.d + w(u, v)$ **then**
16: $A[v.d].delete(v.list)$
17: $v.d = u.d + w(u, v)$
18: $v.\pi = u$
19: $A[v.d].insert(v)$
20: $v.list = A[v.d].head$
21: **end if**
22: **end for**
23: **end for**

serach tree.

Another way of doing this, since we know that the set is of integers drawn from the set $\{1, \ldots, W\}$, is by using a vEB tree. This allows the insertion, deletion, and find minimum to run in time $O(\lg(\lg(W))$.

Since for every edge, we potentially move a vertex from one list to another, we also need to possibly perform a deletion and insertion in the set. Also, at the beginning of the outermost for loop, we need to perform a get min operation, if removing this element would cause the list in that index of A to become empty, we have to delete it from our set of indices. So, we need to perform a set operation for each vertex, and also for each edge. There is only a constant amount of extra work that has to be done for each, so the total runtime is $O((V + E) \lg \lg(W))$ which is also $O((V + E) \lg(W))$.

Exercise 24.3-10

The proof of correctness, Theorem 24.6, goes through exactly as stated in the text. The key fact was that $\delta(s, y) \leq \delta(s, u)$. 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 $y \neq s$ 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.

Exercise 24.4-1

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.

Exercise 24.4-2

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.

Exercise 24.4-3

No, it cannot be positive. This is because for every vertex $v \neq v_0$, 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.

Exercise 24.4-4

To solve the single source shortest path problem we must have that for each edge (v_i, v_j), $\delta(s, v_j) \leq \delta(s, v_i) + w(v_i, v_j)$, and $\delta(s, s) = 0$. We we will use these as our inequalities.

Exercise 24.4-5

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 $O(mn)$.

Exercise 24.4-6

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

Exercise 24.4-7

We could avoid adding in the additional vertex by instead initializing the d value for each vertex to be 0, and then running the bellman ford algorithm. These modified initial conditions are what would result from looking at the vertex v_0 and relaxing all of the edges coming off of it. After we would of processed the edges coming off of v_0, we can never consider it again because there are no edges going to it. So, we can just initialize the vertices to what they would be after relaxing the edges coming off of v_0.

Exercise 24.4-8

Bellman-Ford correctly solves the system of difference constraints so $Ax \leq b$ is always satisfied. We also have that $x_i = \delta(v_0, v_i) \leq w(v_0, v_i) = 0$ so $x_i \leq 0$ for all i. To show that $\sum x_i$ is maximized, we'll show that for any feasible solution $(y_1, y_2, \ldots, y_n)$ which satisfies the constraints we have $y_i \leq \delta(v_0, v_i) = x_i$. Let $v_0, v_{i_1}, \ldots, v_{i_k}$ be a shortest path from v_0 to v_i in the constraint graph. Then we must have the constraints $y_{i_2} - y_{i_1} \leq w(v_{i_1}, v_{i_2}), \ldots, y_{i_k} - y_{i_{k-1}} \leq w(v_{i_{k-1}}, v_{i_k})$. Summing these up we have

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

Exercise 24.4-9

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.

Exercise 24.4-10

Consider introducing the dummy variable x. Let $y_i = x_i + x$. Then $(y_1, \ldots, y_n)$ is a solution to a system of difference constraints if and only if $(x_1, \ldots, x_n)$ is. Moreover, we have $x_i \leq b_k$ if and only if $y_i - x \leq b_k$ and $x_i \geq b_k$ if and only if $y_i - x \geq b_k$. Finally, $x_i - x_j \leq b$ if and only if $y_i - y_j \leq b$. Thus, we construct our constraint graph as follows: Let the vertices be $v_0, v_1, v_2, \ldots, v_n, v$. Draw the usual edges among the v_i's, and weight every edge from v_0 to another vertex by 0. Then for each constraint of the form $x_i \leq b_k$, create edge (x, y_i) with weight b_k. For every constraint of the form $x_i \geq b_k$, create edge (y_i, x) with weight $-b_k$. Now use Bellman-Ford to solve the problem as usual. Take whatever weight is assigned to vertex x, and subtract it from the weights of every other vertex to obtain the desired solution.

Exercise 24.4-11

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.

Exercise 24.4-12

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

Exercise 24.5-1

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 $\{(s, t), (t, y), (t, x), (y, z)\}$.

Exercise 24.5-2

Let G have 3 vertices s, x, and y. Let the edges be $(s, x), (s, y), (x, y)$ 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 $\delta(s, x) = \delta(s, y) = 1$.

Exercise 24.5-3

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.

Exercise 24.5-4

Suppose u is the vertex which first caused $s.\pi$ to be set to a non-NIL value. Then we must have $0 = s.d > u.d + w(u, s)$. Let p be the path from s to u in the shortest paths tree so far, and C be the cycle obtained by following that path from s to u, then taking the edge (u, s). Then we have $w(C) = w(p) + w(u, s) = u.d + w(u, s) < 0$, so we have a negative-weight cycle.

Exercise 24.5-5

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.

Exercise 24.5-6

We will prove this by induction on the number of relaxations performed. For the base-case, we have just called 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 $(n+1)^{st}$ 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.

Exercise 24.5-7

We know by 24.16 that a G_π forms a tree after a sequence of relaxation steps. Suppose that T is the tree formed after performing all the relaxation steps of the Bellman Ford algorithm. While finding this tree would take many more than $V-1$ relaxations, we just want to say that there is some sequence of relaxations that gets us our answer quickly, not necessarily proscribe what those relaxations are. So, our sequence of relaxations will be all the edges of T in an order so that we never relax an edge that is below an unrelaxed edge in the tree(a topological ordering). This guarantees that G_π will be the same as was obtained through the slow, proven correct, Bellman-Ford algorithm. Since any tree on V vertices has $V-1$ edges, we are only relaxing $V-1$ edges.

Exercise 24.5-8

Since the negative-weight cycle is reachable from s, let v be the first vertex on the cycle reachable from s (in terms of number of edges required to reach v) and $s = v_0, v_1, \ldots, v_k = v$ be a simple path from s to v. Start by performing the relaxations to v. Since the path is simple, every vertex on this path is encountered for the first time, so its shortest path estimate will always decrease from infinity. Next, follow the path around from v back to v. Since v was the first vertex reached on the cycle, every other vertex will have shortest-path estimate set to ∞ until it is relaxed, so we will change these for every relaxation around the cycle. We now create the infinite sequence of relaxations by continuing to relax vertices around the cycle indefinitely. To see why this always causes the shortest-path estimate to change, suppose we have just reached vertex x_i, and the shortest-path estimates have been changed for every prior relaxation. Let $x_1, x_2, \ldots, x_n$ be the vertices on the cycle. Then we have $x_{i-1}.d + w(x_{i-1}, x) = x_{i-2}.d + w(x_{i-2}, x_{i-1}) + w(x_{i-1}, w_i) = \ldots = x_i.d + \sum_{j=1}^{n} w(x_j) < w.d$ since the

cycle has negative weight. Thus, we must update the shortest-path estimate of x_i.

Problem 24-1

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

1. 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.

2. 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 $O(d \lg d)$.

3. 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 can sort all boxes' sorted dimensions lexicographically in $O(dn \lg n)$ using radix sort. By transitivity, it will suffice to test adjacent nesting relations. Thus, the total time to build this graph is $O(nd \max \lg d, \lg n)$. Next, we need to find the longest chain in the graph.

Problem 24-3

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 Bellman-Ford 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

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 i^{th} 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

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

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 $\hat{w}$ 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 δ_{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

a. If $\mu^* = 0$, then we have that the lowest that $\frac{1}{k}\sum_{i=1}^{k} w(e_i)$ can be is zero. This means that the lowest $\sum_{i=1}^{k} w(e_i)$ can be is 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 b says that for all v the inequality holds, so, we have

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

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

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

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 $\mu(c) = \frac{1}{k}\sum_{i=1}^{k}(w(e_i)+t) = \frac{1}{k}\left(\sum_i^k w(e_i)\right) + \frac{kt}{k} = \frac{1}{k}\left(\sum_i^k w(e_i)\right) + t$. 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 $k \in \{1,\ldots,n\}$ 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

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 three times (for a total of six 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 6 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.