10/10

6.046 Dan Ports

Problem 5-1

No collab Recitation #8

Part a:

Algorithm First, we convert the given currencies and rates of exchange into a graph whose vertices represent currencies and whose edges have weights given by $w(i,j) = -\lg(r_{i,j})$. Building this graph requires time $\Theta(V+E)$, where $V = n$ is the number of vertices/currencies and E is the number of edges in the graph ($E = O(n^2)$). Next we apply the Bellman-Ford algorithm to find the shortest path from vertex representing the starting currency to the vertex representing the desired currency. We return the sequence of currency exchanges that corresponds to this path. The Bellman-Ford algorithm requires time $O(VE) = O(n^3)$, so the algorithm runs in time $O(n^3)$.

Proof of correctness We will justify correctness by arguing a bijection between the currency exchanges and our graph. For a sequence of exchanges between currencies $x_1, x_2, ..., x_n$, the overall exchange rate is the product of the individual exchange rates: $\prod_{i=1}^{n-1} r_{x_i,x_{i+1}}$. This equals $2^{\lg \prod_{i=1}^{n-1} r_{x_i,x_{i+1}}} = 2^{\sum_{i=1}^{n-1} \lg r_{x_i,i+1}} = 2^{-\sum_{i=1}^{n-1} w(x_i,x_{i+1})}$, applying the properties of logarithms and the definition of $w(i,j)$.

Therefore, to maximize the currency we end up with, we need to maximize the rate $2^{-\sum_{i=1}^{n-1} w(x_i,x_{i+1})}$. To do this, we minimize $\sum_{i=1}^{n-1} w(x_i,x_{i+1})$, which is the total cost of a path $x_1, x_2, ...x_n$ leading from the zlotys vertex to the dollars vertex in our graph. We assume the Bellman-Ford algorithm is correct and provides us the lowest-cost path from the starting vertex to the target vertex. The only necessary condition is that there are no negative-weight cycles in our graph. We know that this is the case. By contradiction: suppose there were a cycle with weight $-x$ ($x > 0$). Then, by the above weight-rate bijection, the net exchange rate for following that cycle of currency exchanges is $2^{-(-x)} = 2^x > 1$. This means that following this directed trades makes a profit, which violates one of our given assumptions. So there can be no negative-weight cycles and the Bellman-Ford algorithm gives the correct result.

10/10

6.046	Dan Ports
Problem 5-2	
No collab	Recitation #8

Part a:

We first show that if a graph has an Euler tour, then in-degree(v)=out-degree(v) for every vertex $v \in V$. A Euler tour, by definition, passes through all edges, and thus through all vertices since we are given that the graph is connected. Choosing an arbitrary starting vertex x, the tour starts at x, passes through every other vertex (and possibly also x again) at least once, and returns to x. Each time the tour passes through one vertex, it must enter the vertex and then exit it; there is also one edge that leaves x at the beginning and one that arrives at x at the end of the tour. So the Euler tour contains one edge entering each vertex for each edge leaving the vertex; since every edge in the graph must be part of the Euler tour, we must have in-degree(v)=out-degree(v) for every $v \in V$.

To show that any graph with in-degree(v)=out-degree(v) has an Euler tour, we use the algorithm in part b. This algorithm finds a Euler tour, and it requires only that in-degree(v)=out-degree(v), so it suffices as a proof.

Part b:

Algorithm Assume we are given a graph as a list of vertices V and a list of directed edges E, and it satisfies in-degree(v)=out-degree(v) $\forall v \in V$.

Next we use a modified depth-first search to generate a list of cycles $\{C_1, C_2, ...\}$ in this graph. We modify the given DFS algorithm (CLRS, p. 541) so that it can mark edges as either visited or unvisited. We initialize each edge (in $\Theta(|E|)$ time) to be unvisited. We also ignore the "coloring" process done by the given DFS algorithm. We modify the DFS-Visit routine so that, when we DFS-Visit a node u, we select the first unmarked edge (call it e) from u to some other vertex (call it v), mark that edge e, and add e to our current cycle C_i. We then DFS-Visit the vertex v, and continue this process until we reach a vertex that has no remaining unmarked edges. We will show (below) that this only happens when we complete a cycle.

To use this modified DFS-Visit to find cycles, we start by initializing $i \leftarrow 1$ and the list of found cycles to be empty. We begin by choosing some arbitrary unmarked edge e and apply DFS-Visit to its source vertex. When this terminates, C_1 will contain a cycle beginning with edge e, and every edge in the cycle will be marked. If there are no remaining unmarked edges, then the cycle we just found contains every edge, and it is an Euler tour, so we are done. Otherwise we increment $i \leftarrow i+1$, choose another unmarked edge e, and DFS-Visit the source vertex of e. We repeat this process until there are no remaining unmarked edges. We then have a set of edge-disjoint cycles $\{C_1, C_2, ...C_n\}$ that contain every edge of the graph.

We show that we can combine this set of cycles into an Euler tour. Because the graph is connected, we can find two cycles C_i and C_j that share a vertex (call it v). If this were not true, then some vertex would not be reachable from some other vertex, contradicting the connectedness of the graph. Choosing arbitrary starting vertices $i \in C_i$ and $j \in C_j$, we can write the two cycles as paths $i \rightarrow v \rightarrow i$ and $j \rightarrow v \rightarrow j$. We can then replace C_i and C_j with a new cycle that follows the path

$i \to v \to j \to v \to i$. This new cycle contains every edge in $C_i \cup C_j$. We can repeat this process, combining cycles until we have only one remaining cycle that contains all edges in the graph. This cycle is an Euler tour, so we return it.

Proof of correctness We must show that, when we call DFS-Visit on some vertex u, it recursively follows a chain of edges (marking them and adding them to the current cycle), and can only terminate when it has returned to the vertex u. We show this first for the initial case, when all edges are unmarked. We know that in-degree(v) = out-degree(v) $\forall v \in V$, and so unmarked-in-degree(v) = unmarked-out-degree(v) (because all edges are unmarked). When we start by DFS-Visiting a vertex u, we choose an edge e from u to v (we know one exists because we chose u accordingly), add it to the cycle, and mark it. This decreases unmarked-out-degree(u) and unmarked-in-degree(v) by 1. For every other vertex v we encounter along this path, we decrement unmarked-in-degree(v) when we mark the edge into v and decrement unmarked-out-degree(v) when we mark the edge out of v. Thus we maintain the invariant that, for all $v \neq u$ unmarked-in-degree(v) = unmarked-out-degree(v). Assuming $v \neq u$, we know there will be an unmarked edge out of v because unmarked-in-degree(v) has to be non-zero for there to be an unmarked edge into v, and unmarked-in-degree(v) = unmarked-out-degree(v). The only time this is not true is when $v = u$, in which case unmarked-out-degree(u) = unmarked-in-degree(u)-1 because we already marked one edge out of u when we started the traversal. Since the DFS-Visit recursion only terminates when there are no remaining unmarked edges, the recursion can only terminate when it returns to the starting vertex u, thus finding a cycle. We have also shown that the unmarked-in-degree(v) = unmarked-out-degree(v) $\forall v \in V$ invariant is maintained, so the same proof applies for later calls to DFS-Visit.

We have just shown that repeatedly calling DFS-Visit as specified in the algorithm produces cycles. Each application of DFS-Visit will identify and mark a cycle, so it will mark some elements on each iteration. Thus the number of unmarked edges is strictly decreasing, so it will eventually reach zero and every edge will be marked. Since we continue to call DFS-Visit until there are no remaining unmarked edges, when we finish, the set $\{C_1, C_2, ...C_n\}$ contains all edges in V. Also, since we can never add a marked edge to a new cycle, each of the cycles C_i are edge-disjoint. We showed a process for combining these cycles into one large cycle two paragraphs above, and argued its correctness. The resulting path is a cycle that contains every edge, so it is an Euler tour, which is the correct result.

Runtime analysis We will construct several additional data structures to make this algorithm efficient. First, we construct an adjacency list A of unmarked edges, which is an array of doubly-linked lists, one for each vertex (noting that each edge is initially unmarked). For each vertex $v \in V$, A_v is a list of pointers to all unmarked edges in E whose source endpoint is v. Generating these adjacency lists takes time $\Theta(|E|)$. Given a vertex v, this allows us to find some unmarked edge that starts at v in constant time, and allows us to unmark an edge in constant time (assuming we have a pointer to the edge's entry in the adjacency list, which we accomplish by including with each edge a pointer to its location in the adjacency list).

We will represent a cycle as a data structure containing a doubly-linked list of pointers to edges and a size-$|V|$ array that allows us to quickly check whether a vertex is in the cycle. Specifically, the vth element of the array is null if v is not in the cycle, and a pointer to an edge in the cycle beginning at v if it is. It takes constant time to initialize this cycle, since we assume that we can initialize an array to zero in constant time, and initializing a linked list is trivial. This structure allows us to add an edge to the cycle in constant time (add it to the end of the list, and set the corresponding pointer in the array). We can also check (using the array) whether a given vertex is included in the cycle, in constant time.

These data structures allow us to perform the DFS-Visit cycle-identification procedure in time $\Theta(|E|)$. Each call to DFS-Visit requires choosing some arbitrary unmarked edge, marking that edge, and adding it to the current cycle, all of which we have shown above can be done in $\Theta(1)$ time. Each call to DFS-Visit marks an unmarked edge, and there are only $|E|$ unmarked edges to begin with, so DFS-Visit is called a total of $\Theta(|E|)$ times at constant runtime per call, so the overall runtime of the cycle-identification procedure is $\Theta(|E|)$.

These data structures also allow us to merge the cycles in $\Theta(|E|)$ time. To merge a cycle C_j into a cycle C_i, we iterate through each edge e in C_j's list, checking at each point whether the endpoint of e is a vertex in C_i. If it is, we use C_i's array to find the edge f that begins at the endpoint of e. We then continue to iterate through C_j, splicing each edge in before f and setting the appropriate pointer in C_i's array. Once we have iterated through C_j, we have updated the cycle C_i so that it contains every edge that was formerly in $C_i \cup C_j$, and we can discard C_j. Each iteration of this merge procedure takes constant time, so the total runtime of merging C_j into C_i is $\Theta(|C_j|)$. Thus, the total runtime required to merge all cycles $C_1, C_2, ...C_n$ is $\Theta(\sum_{i=1}^{n} |C_i|) = \Theta(|E|)$

Each of the two major parts of the algorithm has runtime $\Theta(|E|)$, so the algorithm's overall runtime is $\Theta(|E|)$.

10/10

6.046	Dan Ports
Problem 5-3	
No collab	Recitation #8

Algorithm We will determine if a graph is bipartite by attempting to two-color it. We take as input a set V of vertices and a set E of edges. Assume w/o loss of generality that the graph is connected (if it is not, we can divide the graph into its connected components in $\Theta(|E|)$ time and test whether each of the components is bipartite).

First, we construct an adjacency list A, which is an array of linked lists, one for each vertex. For each vertex $v \in V$, A_v is a list of pointers to all edges in E whose source endpoint is v. Generating these adjacency lists takes time $\Theta(|E|)$, and it allows us to obtain the list A_v of edges leading from v in time $\Theta(|A_v|)$.

We will augment the list of vertices to associate a color field with each vertex. Each vertex can be assigned one of two colors – which we will arbitrarily choose to be red and blue – or be uncolored. We first select some arbitrary vertex u and arbitrarily color it red. We then perform a modified breadth-first search. We modify the algorithm slightly so that whenever we dequeue a vertex u, we check the color of each vertex v in the adjacency list A_u. If v is uncolored, then it has not been previously visited, and we color it the opposite of u (and add it to the queue, as the normal BFS algorithm does). If v is already colored, then we compare the colors of u and v. If they are identical, we immediately declare the graph non-bipartite and return. Otherwise, we continue the BFS. When the BFS finishes, if the algorithm has not already terminated (by finding the graph non-bipartite), we declare the graph to be bipartite and return.

Proof of correctness We will show that for either result, the algorithm is correct. Suppose the algorithm returns bipartite. Then it has constructed a two-coloring for the graph. We know this is true, because at every [redacted] of the BFS we verified that any two adjacent vertices have different colors. Thus, the graph is indeed bipartite.

If the algorithm returns non-bipartite, we know that it found two adjacent nodes u and v that were colored the same color. We will show that this means there is no possible two-coloring of the graph. Because both u and v were already colored, they were previously dequeued and colored by the algorithm. Call the starting node s. Because u and v have the same color, the paths $s \to u$ and $s \to v$ must either both have even length or both have odd length. So the path $u \to s \to v$ has length equal to their sum, which will be even. Then, since u and v are adjacent, the path $u \to s \to v \to u$ has length one greater, and thus has odd length. This means that the graph contains an odd cycle. From 6.042, we know that there is no way to two-color a graph that contains an odd cycle (trivial proof omitted). So if the algorithm returns non-bipartite, the graph is non-bipartite. Thus the algorithm always returns the correct result.

Runtime analysis Generating the adjacency list requires time $\Theta(|E|)$. Our modifications to the BFS procedure do not change the runtime. Each adjacency list is scanned at most once, so scanning the adjacency lists requires $O(\sum_{v \in V} |A_v|) = O(|E|)$, and the overhead for initializing the vertices is $O(|V|)$. So the total runtime of the algorithm is $O(|V| + |E|) = O(|E|)$, since we are given the assumption that $|E| = \Omega(|V|)$.

10/10

6.046	Dan Ports
Problem 5-4	
No collab	Recitation #8

Algorithm Given a graph $G = (V, E)$ and a source node s, we will identify the upwards and downwards critical edges by computing for each node the nodes that can be predecessors in a shortest path. We will show (in the proof of correctness below) that an edge is downwards critical iff it appears in the shortest-path-predecessor list for the edge's destination vertex, and that it is upwards critical iff it is the only shortest-path-predecessor for the destination vertex.

We can find the predecessor nodes by modifying Dijkstra's algorithm. Rather than maintaining a single predecessor pointer $\pi[v]$ for every vertex $v \in V$, we maintain a predecessor list that contains a pointer to each node that could be a predecessor in a shortest path to v. To do this, we modify the Relax procedure slightly. If $d[v] > d[u] + w(u, v)$, then in addition to updating the distance $d[v]$ as Dijkstra's algorithm normally does, we also replace v's predecessor list with u: $\pi[v] \leftarrow \{u\}$. We also check whether $d[v] = d[u] + w(u, v)$. In this case, we append u to v's predecessor list: $\pi[v] \leftarrow \pi[v] \cup \{u\}$. If $d[v] < d[u] + w(u, v)$, we still do nothing.

Once our modified Dijkstra's algorithm terminates, we use the predecessor lists to identify which edges are upwards and downwards critical edges. We iterate through every edge $e \in E$, and let v be the destination vertex of e. We check whether $e \in \pi[v]$; if so, then we designate e as downwards-critical if $w(e) > 0$. We also check whether e is the only element in $\pi[v]$ and designate e as upwards-critical if so.

Runtime analysis Our modifications to Dijkstra's algorithm do not affect the runtime. Appending to and/or replacing the list $\pi[v]$ adds only a constant amount of time to each call to Relax. So, if we implement Dijkstra's algorithm using a binary min-heap, it can run in time $O(|E| \lg |V|)$. Scanning the predecessor lists requires $\Theta(|E|)$ time for loop overhead and to find the lists $\pi[v]$, since we iterate over each $e \in E$. Checking whether each edge e is in its appropriate $\pi[v]$ requires $\Theta(|\pi[v]|)$ time, for the appropriate v, so the amortized cost of all the tests is $\Theta(\sum_{v \in V} |\pi[v]|) = \Theta(|E|)$ since there are $|E|$ edges and each one appears in only one set $\pi[v]$ since it only has one destination vertex. So the total cost of the algorithm is $O(|E| \lg |V|)$

Proof of correctness We will first show that the algorithm correctly generates the list of predecessor nodes. We base this on the correctness of Dijkstra's algorithm, which we can assume since the graph contains no negative weights. First, we show that every element $u \in \pi[v]$ is in fact a shortest-path-predecessor of v. Note that, when Relax(u, v, w) is called, u will only be added to $\pi[v]$ if the cost of the path through u to v is less than that of all previously discovered paths. If a path through some other node x is later discovered that is shorter, then Relax(x, v, w) will replace $\pi[v]$ with $\{x\}$, removing u. By the time the algorithm examines the adjacency list of v, it will have found the correct shortest path to v, and so it will have called Relax on every shorter path, so every element in $\pi[v]$ is a predecessor node.

We next show that there are no predecessor nodes that are not in $\pi[v]$. Suppose there were some u that should be in $\pi[v]$ but is not. This would require that the distance of the shortest path to u is less than the shortest path distance to v by $w(u, v)$, and there can not be any shorter path to v. But this would mean that Dijkstra's algorithm would visit u before it visits v, and when it

calls Relax(u, v, w), it would find that $d[v]$ is either greater than or equal to $d[u] + w(u, v)$, and add u to $\pi[v]$. So there is no way $u \notin \pi[v]$. Thus, we have shown that $\pi[v]$ contains all shortest-path predecessor nodes of v.

To show the algorithm is correct, we show that an edge e is downwards critical if and only if it appears in its destination node's predecessor list (and e's weight is non-zero), and that e is upwards critical if and only if it is the only predecessor node of it's destination node. Consider an edge e such that $w(e) > 0$ and call its source node u and its destination node v. If e is a shortest-path-predecessor of v, then no other path from s to v has length less than the path $s \to u \to v$ has length. So if we lower $w(e)$, then we lower the distance of the shortest path to v, and so e is downwards-critical. If e is not a predecessor of v, then it is not downwards-critical. If it were, then lowering the weight of e would lower the distance of the path to some node p. Since e is an edge from u to v, this means that the path $s \to u \to v \to p$ would be shortened. This means the path $s \to u \to v$ must be the shortest path to v (and so $e \in \pi[v]$), because otherwise there would be a shorter path from s to v and thus a shorter path $s \to v \to p$.

Next we show that e is upwards-critical if and only if it is the only predecessor of v. If e is the only predecessor of v, then $s \to u \to v$ is shorter than all other paths to v, so increasing $w(e)$ increases the shortest distance to v. If e is not a predecessor, then there is some shorter path from s to v which does not include e, and it will continue to be shorter if $w(e)$ is raised. If e is not the only predecessor, then there is some path from s to v that does not include e that has the same length as the one that includes e. The former path will not increase in length if $w(e)$ is raised, so the shortest path from s to v is unchanged. Thus the shortest path to any vertex beyond v is not increased unless e is the only predecessor of v.

Since we have shown that our algorithm correctly computes the predecessors of each node, and that the predecessors of each node correctly give the upwards and downwards critical edges, our algorithm gives the correct result.