Chapter 4 – Solutions

4.1. The shortest path tree is shown in Figure 1.

	Iteration							
Node	0	1	2	3	4	5	6	7
A	0	0	0	0	0	0	0	0
B	∞	1	1	1	1	1	1	1
C	∞	∞	3	3	3	3	3	3
D	∞	∞	∞	4	4	4	4	4
E	∞	4	4	4	4	4	4	4
F	∞	8	7	7	7	7	6	6
G	∞	∞	7	5	5	5	5	5
H	∞	∞	∞	∞	8	8	6	6

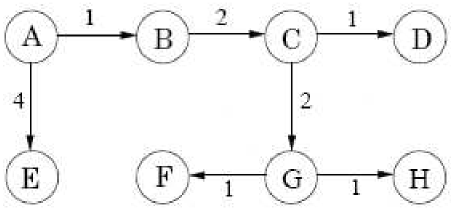

Figure 1: Shortest-path tree for 4.1.

4.2. The shortest path tree is shown in Figure 2.

	Iteration						
Node	0	1	2	3	4	5	6
S	0	0	0	0	0	0	0
A	∞	7	7	7	7	7	7
B	∞	∞	11	11	11	11	11
C	∞	6	5	5	5	5	5
D	∞	∞	8	7	7	7	7
E	∞	6	6	6	6	6	6
F	∞	5	4	4	4	4	4
G	∞	∞	∞	9	8	8	8
H	∞	∞	9	7	7	7	7
I	∞	∞	∞	∞	8	7	7

4.3. Suppose the input graph G is given as an adjacency matrix. Notice that G contains a square if and only if there are two vertices u and v that share more than one neighbor. For any u, v we can check this in time $O(|V|)$ by comparing the row of u and the row of v in the adjacency matrix of G. Because we need to repeat this process $O(|V|^2)$ to iterate over all u and v, this algorithm has running time $O(|V|^3)$. We can do better by noticing that, when comparing the rows of the adjacency matrix a_u and a_v, we are actually checking if $a_u \cdot a_v$ is greater than 1, i.e. if $[A(G)^2]_{uv} > 1$. It suffices then to

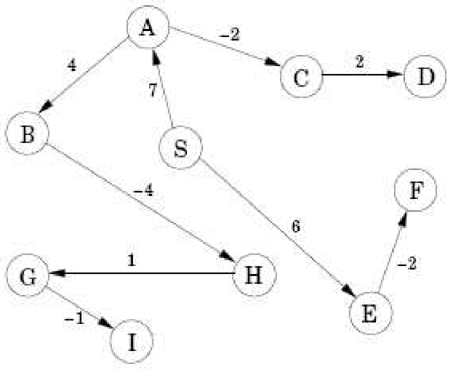

Figure 2: Shortest-path tree for 4.2.

compute $A(G)$, which we can do in time $O(|V|^{2.71}$ using our matrix multiplication algorithm and check all non-diagonal entries to see if we find one larger than 1.

4.4. The graph in Figure 3 is a counterexample: vertices are labelled with their level in the DFS tree, back edges are dashed. The shortest cycle consists of vertices $1-4-5$, but the cycle found by the algorithm is $1-2-3-4$. In general, the strategy will fail if the shortest cycle contains more than one back edge.

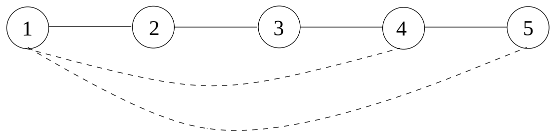

Figure 3: Counterexample for 4.4.

4.5 We perform a BFS on the graph starting from u, and create a variable num_paths(x) for the number of paths from u to x, for all vertices x. If $x_1, x_2, \ldots x_k$ are vertices at depth l in the BFS tree and x is a vertex at depth $l+1$ such that $(x_1, x), \ldots, (x_k, x) \in E$ then we want to set num_paths(x) = num_paths(x_1)+…+num_paths(x_k). The easiest way to do this is to start with $\mathtt{num}_\mathtt{p}$aths(x) $= 0$ for all vertices $x \neq u$ and num_paths(u) $= 1$. We then update num_paths(y) $=$ num_paths(y) $+$ num_paths(x), for each edge (x, y) that goes down one level in the tree. Since, we only modify BFS to do one extra operation per edge, this takes linear time. The pseudocode is as follows

```
function count_paths($G, u, v$)
    for all $x \in V$:
        dist($x$) = $\infty$
        num_paths($x$) = 0

    dist($u$) = 0
    num_paths($u$) = 1
    $Q$ = [$u$]
    while $Q$ is not empty:
        $x$ = eject($Q$)
        for all edges $(x, y) \in E$
            if dist($y$) = dist($x$) + 1:
```

```
        num_paths($y$) = num_paths($y$) + num_paths($x$)
    if dist($y$) = $\infty$:
        inject($Q$,$y$)
        dist($y$) = dist($x$) + 1
        num_paths($y$) = num_paths($x$)
```

4.6. This is true as long as every node $u \in V$ is reachable from s, the node from which Dijkstra's algorithm was run. In this case, every node is connected to s through a shortest path consisting of edges of the type $\{u, \texttt{prev}(u)\}$ and the graph formed by all these edges is connected. Moreover, such graph cannot have cycles, because $\{u, \texttt{prev}(u)\}$ can be an edge only if $\texttt{prev}(u)$ was deleted off the heap before u, so that a cycle would lead to a contradiction.

4.7. Let w_e be the weight of edge e. Let $d_T(u, v)$ be the distance between u and v along the edges of T. This can be computed in linear time by either BFS or DFS. For any edge $(u, v) = e \in E - E'$, i.e. every e not belonging to the tree, check that

$$w_{(u,v)} + d_T(s, u) \geq d_T(s, v)$$

This will succeed for all $e \in E - E'$ if and only if T is a correct shortest path tree from s. We proceed to prove this.

If T is a correct shortest path tree $d_T(s, v) = d_G(s, v)$ and hence $w_{(u,v)} + d_T(s, u) \geq d_T(s, v)$ or the path through (u, v) would be shorter. If T is not a correct shortest path, consider a run of the Bellman Ford algorithm for shortest paths starting at s. Consider the first update $\texttt{update}(u, v)$ causing the distance of v to drop below $d_T(s, v)$. Let d_u be the distance label of u at the moment of that update. As this was the first update contradicting d_T, it must be the case that $d_T(s, u) \leq d_u$. Then, we have $d_T(s, u) + w_{(u,v)} \leq d_u + w_{(u,v)} < d_T(s, v)$, by construction. Hence, our algorithm will detect such edge (u, v) and correctly recognize T as an invalid shortest path tree from s.

4.8. The weighted graph in Figure 4 is a counterexample: According to the algorithm proposed by Professor

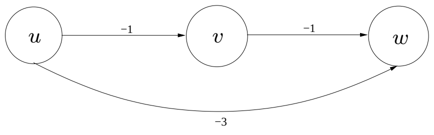

Figure 4: Counterexample for 4.8.

Lake we should add +4 to the weight of each edge. Then, the shortest path between u and w would be the edge (u, w) of weight 1. However, the shortest path in the original graph was $u - v - w$.

4.9. As stated the answer is negative, because there could be a negative cycle involving s. However, it is more interesting to ask what happens in no such cycle exists. In this case, Dijkstra's algorithm works. Consider the proof of Dijkstra's algorithm. The proof depended on the fact that if we know the shortest paths for a subset $S \subseteq V$ of vertices, and if (u, v) is an edge going out of S such that v has the minimum estimate of distance from s among the vertices in $V \setminus S$, then the shortest path to v consists of the (known) path to u and the edge (u, v). We can argue that this still holds even if the edges going out of the vertex s are allowed to negative. Let (u, v) be the edge out of S as described above. For the sake of contradiction, assume that the path claimed above is not the shortest path to v. Then there must be some other path from s to v which is shorter. Since $s \in S$ and $v \notin S$, there must be some edge (i, j) in this path such that $i \in S$ and $j \notin S$. But then, the distance from s to j along this path must be greater than that the estimate of v, since v had the minimum estimate. Also, the edges on the path between j and v must all have non-negative weights since the only negative edges are the ones out of s. Hence, the distance along this path from s to v must be greater than the the estimate of v, which leads to a contradiction.

4.10. Perform k rounds of the `update` procedure on all edges.

4.11. Define matrix D so that D_{ij} is the length of the shortest path from vertex i to vertex j in the input graph. Row i of the matrix can be computed by a run of Dijkstra's algorithm in time $O(|V|^2)$. So we can calculate all of D in time $O(|V|^3)$. For any pair of vertices u, v we know that there is a cycle of length $D_{uv}+D_{vu}$ consisting of the two shortest paths between u and v and that this cycle is the shortest among cycles containing u and v. This shows that it suffices to compute the minimum $D_{uv} + D_{vu}$ over all pairs of vertices u, v to find the length of the shortest cycle. This last operation takes time $O(|V|^2)$, so the overall running time is $O(|V|^3)$.

4.12. Let u and v be the end vertices of e. Using Dijkstra's algorithm, compute shortest path lengths from u and v to all other vertices in $G - e$, the graph obtained removing edge e from G. Let $d_u(x)$ and $d_v(x)$ respectively denote the shortest path length from u and v to node x in this graph. For any node x, there exists then a cycle of size at most $d_u(x) + d_v(x) + 1$ containing e in G: this cycle consists of the non-overlapping parts of the shortest paths from u and v to x and of edge e. Moreover, if x belongs to the shortest cycle C containing e, $d_u(x) + d_v(x) + 1$ must be the length of C or a shorter cycle will exist. This shows that the length of C is the minimum over all x of $d_u(x) + d_v(x) + 1$, which can be calculated in time $O(|V|)$. Hence, the overall running time is $O(|V|^2)$ given by the two initial runs of Dijkstra's algorithm.

4.13. a) This can be done by performing DFS from s ignoring edges of weight larger than L.

b) This can be achieved by a simple modification of Dijkstra's algorithm. We redefine the distance from s to t to be the minimum over all paths p from s to u of the maximum length edge over all edges of p. Compare this with the original definition of distance, i.e. the minimum over all p of the sum of lengths of edges in p. This comparison suggests that by modifying the way distances are updated in Dijkstra we can produce a new version of the algorithm for the modified problem. It is sufficient to change the final loop to:

```
while $H$ is not empty:
    $u$ = deletemin($H$)
    for all edges $(u,v) \in E$:
        if dist(v) > max(dist($u$), $l(u,v)$)
            dist(v) = max(dist($u$), $l(u,v)$)
            prev(v) = u
            decreasekey(H,v)
```

When implemented with a binary heap, this algorithm achieves the required running time.

4.14. Let P be shortest path from vertex u to v passing through v_0. Note that, between v_0 and v, P must necessarily follow the shortest path from v_0 to v. By the same reasoning, between u and v_0, P must follow the shortest path from v_0 and u in the reverse graph. Both these paths are guaranteed to exist as the graph is strongly connected. Hence, the shortest path from u to v through v_0 can be computed for all pairs u, v by performing two runs on Dijkstra's algorithm from v_0, one on the input graph G and the other on the reverse of G. The running time is dominated by looking up all the $O(|V^2|)$ pairs of distances.

4.15. This can be done by slightly modifying Dijkstra's algorithm in Figure 4.8. The array `usp[·]` is initialized to `true` in the initialization loop. The main loop is modified as follows:

```
while $H$ is not empty:
    $u$ = deletemin($H$)
    for all edges $(u,v) \in E$:
```

```
if dist($v$) $>$ dist($u$) + $l(u,v)$:
    dist($v$) $=$ dist($u$) + $l(u,v)$
    usp($v$) = usp($u$)
    decreasekey($H, v$)
if dist($v$) $=$ dist($u$) + $l(u,v)$:
    usp($v$) = false
```

This will run in the required time when the heap is implemented as a binary heap.

4.16. a) If the node at position j is the ith node on the kth level of the binary tree we have $j = 2^k + i - 1$. Its parent will then be the $\lceil \frac{i}{2} \rceil$th node on the $(k-1)$th level of the tree and will be found in position $2^{k-1} + \lceil \frac{i}{2} \rceil - 1 = 2^{k-1} + \lfloor \frac{i-1}{2} \rfloor = \lfloor \frac{j}{2} \rfloor$. Its children will be the $2i-1$th and $2i$th nodes on the $(k+1)$th level of the tree. Hence, they will be stored in positions $2^{k+1} + 2i - 2 = 2j$ and $2^{k+1} + 2i - 1 = 2j + 1$.

b) By the same reasoning as above, the parent of the node at position j will be at position $\lceil \frac{j-1}{d} \rceil$, while the children will be at position $dj + 1, dj, dj - 1, \cdots, dj - (d-2)$.

c) The procedure `siftdown` places element x at position i of h and rearranges the heap by letting x "'sift down"' until both its children have values greater than x. This is done by iteratively swapping x with the minimum of its children and takes at most time proportional to the height of the subtree rooted at i. The height of the whole tree is $\log n$, as there are n nodes in the heap, and the depth of node i is $\log i$, so that the subtree rooted at i has depth $\log n - \log i = \log \left(\frac{n}{i}\right)$. Because `makeheap` calls `siftdown` at all nodes in the tree, `makeheap` will take time:

$$\sum_{i=1}^{n} \log \left(\frac{n}{i}\right) = \log \frac{n^n}{n!}$$

By Stirling's formula (see page 53), $n! \geq \left(\frac{n}{e}\right)^n$, so:

$$\log \left(\frac{n^n}{n!}\right) \leq \log(e^n) = O(n)$$

d) In procedure `bubbleup`, p must be assigned to the index of the parent node of i, for which we gave the formula in b). In `minchild`, the minimum must be taken over all children of node i, the indices of which we gave in b).

4.17. a) Because every edge has length $\{0, \cdots, W\}$, all `dist` values will be in the range $\{0, 1, \cdots, W(|V|-1), \infty\}$, as any shortest path contains at most $|V|-1$ edges. Hence, we can implement the heap on the `dist` values by mantaining an array of size $W(|V|-1)+2$ indexed by all possible values of `dist`, where each entry i is a pointer to a linked list of elements having `dist` value equal to i. With this implementation, we can perform `insert` operations in constant time, simply by appending the element at the beginning of the linked list corresponding to its value. So, `makeheap` will take time $O(|V|)$. Because during a run of Dijkstra's algorithm the minimum value on the heap is increasing, when we perform a `deletemin` operation we do not need to scan the whole array, but can start looking for the new minimum at the previous minimum value. This implies that, in scanning for the minimum element, we look at each array entry at most once, so that all the `deletemin` operations take time $O(W|V|)$. Finally, the `decreasekey` operation can be implemented by inserting a new copy of the element into the list corresponding to its new value, without removing the previous copies. This means that, when performing `deletemin` operations, we need to check whether the current minimum is a copy of an element already processed and, in that case, ignore it. But, because there at most $|E|$ `decreasekey` operations, there are at most $|E|$ copies we need to ignore and we only pay a penalty of time $O(|E|)$. Moreover, in this way, each `decreasekey` takes time $O(1)$, so that all `decreasekey` operations take time $O(|E|)$. This shows that the running time of Dijkstra's algorithm with this implementation is $O(|V|) + O(W|V|) + O(|E|) = O(W|V| + |E|)$.

b) The key observation is that there are at most $W + 2$ distinct `dist` values in the heap at any one time, namely ∞ or any integer between *Min* (the current smallest value in the heap) and *Min*+W. We can now implement the heap using a binary heap with at most $W + 2$ leaf nodes, where keys are possible `dist` value (including ∞), and the value associated with the key is a linked list that contains all nodes with that `dist` value. We need to make a few modifications to Dijkstra's algorithm. When we update a node's `dist` value to j, we simply append this node to the linked list at the key value j (again, we do not remove the node from the linked list corresponding to the old `dist` value). If we take out a node v during `deleteMin` that has been processed, we simply ignore the node and move on. We can do now `deletemin` and `insert` in $O(\log W)$ time, since the heap has at most $W + 2$ children at any one time. We also perform at most $|V| + |E|$ `deletemin`'s and at most $|V| + |E|$ `insert`'s, hence the total running time of the algorithm is $O((|V| + |E|) \log W)$.

4.18. This is another simple variation of Dijkstra's algorithm of Figure 4.8. In the initialization loop, `best`(s) is set to 0 and all other entries of `best` are set to ∞. The main loop is modified as follows:

```
while $H$ is not empty:
    $u =$ deletemin($H$)
    for all edges $(u, v) \in E$:
        if dist($v$) $>$ dist($u$) + $l(u, v)$:
            dist($v$) $=$ dist($u$) + $l(u, v)$
            best($v$) = best($u$) $+ 1$
            decreasekey($H, v$)
        if dist($v$) $=$ dist($u$) + $l(u, v)$:
            if best($v$) $<$ best($u$) $+ 1$:
                best($v$) $=$ best($u$) $+ 1$
```

This has the same asymptotic running time as the original Dijkstra's algorithm, as the additional operations in the loop take constant time.

4.19. There are two approaches: one is a *reduction*; the other is a direct modification of Dijkstra's algorithm.

Method I: The idea is to use a *reduction*: on input (G, l, c, s), we construct a graph $G' = (V', E')$ where G only has edge weights (no node weights), so that the shortest path from s to t in G is essentially the same as that in G', with some minor modifications. We can then compute shortest paths in G' using Dijkstra's algorithm.

The reduction works by taking every vertex v of G and splitting it into two vertices v_i and v_o. All edges coming into v now come into v_i, while all edges going out of v now go out of v_o. Finally, we add an edge from v_i to v_o of weight $c(v)$.

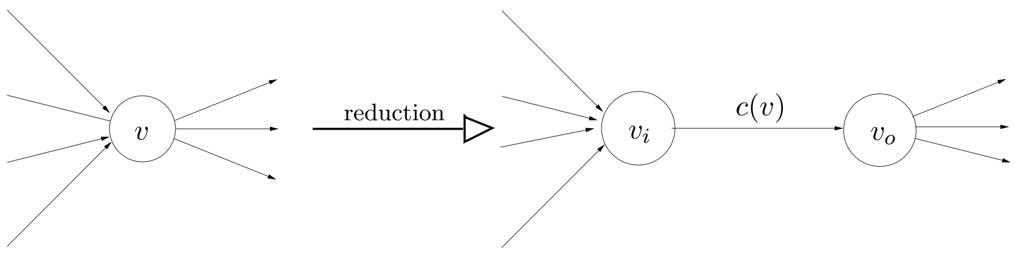

Figure 5: Reduction in 4.19.

Consider now any path in G and notice that it can be converted to an edge-weighted path of the same weight in G' by replacing the visit to vertex v with the traversal of edge (v_o, v_i). Conversely, consider a path in G': every other edge visited is of the form (v_i, v_o) and corresponds to a vertex v of G. Replacing these edges with the corresponding vertices we obtain a path in G of the same weight as the path in G'. The time required to perform this reduction is $O(|V| + |E|)$. G' has $|V| + |E|$ edges and $2|V|$ vertices, so running Dijkstra takes time $O(|V|^2)$ and the total running time is $O(|V|^2)$.

Method II: We make the following modifications to Dijkstra's algorithm to take into account node weights:

– In the initialization phase, $\texttt{dist(s)} = \texttt{w(s)}$.
– In the update phase, we use $\texttt{dist(u)} + \texttt{l(u,v)} + \texttt{w(v)}$ instead of $\texttt{dist(u)} + \texttt{l(u,v)}$.

Analysis of correctness and running time are exactly the same as in Dijkstra's algorithm.

4.20. G is an undirected graph with edge weights l_e. If the distance between s and t decreases with the addition of $e' = (u, v)$, the new shortest path from s to t will be the concatenation of the shortest path from s to u, the edge (u, v) and the shortest path from v to t. But we can then compute the length of this path by running Dijkstra's algorithm once from s and once from t in G. With all the shortest path distances from s and t in G, we can compute in constant time the length of the shortest path from s to t going through e' for any $e' \in E'$. The shortest of these paths will give us the best edge to add and its length will tell us what improvement the addition brings, if any. The running time of this algorithm is $O(|V|^2 + |E'|)$.

4.21. a) Represent the currencies as the vertex set V of a complete directed graph G. To find the most advantageous ways to converts c_s into c_t, you need to find the path $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ maximizing the product $r_{i_1,i_2} r_{i_2,i_3} \cdots \cdot r_{i_{k-1},i_k}$. This is equivalent to minimizing the sum $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}})$. Hence, it is sufficient to find a shortest path in the graph G with weights $w_{ij} = -\log r_{ij}$. Because these weights can be negative, we apply the Bellman-Ford algorithm for shortest paths to the graph , taking s as origin.

b) Just iterate the updating procedure once more after $|E||V|$ rounds. If any distance is updated, a negative cycle is guaranteed to exist, i.e. a cycle with $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}}) < 0$, which implies $\prod_{j=1}^{k-1} r_{i_j,i_{j+1}} > 1$, as required.

4.22. a) Let C be the negative cycle, $E(C)$ the set of edges of C, $V(C)$ the set of vertices. Then:

$$\sum_{e \in E(C)} w_e = r \sum_{e \in E(C)} c_e - \sum_{v \in V(C)} p_v < 0$$

This shows that $\frac{\sum_{v \in V(C)} p_v}{\sum_{e \in E(C)} c_e} > r$, so that $r < r^*$.

b) The same argument as in a) yields that all cycles have ratio less than r, so that $r > r^*$.

c) We can use Bellman-Ford to detect negative cycles, so, for any r, we can check in time $O(|V||E|)$ whether r is smaller or greater than r^*. We can then perform a binary search for r^* on the interval $[0, R]$. After $\log\left(\frac{R}{\epsilon}\right)$ rounds of binary search, our lower bound r' on r^* will be at most ϵ smaller than r^*, i.e. $r' \geq r^* - \epsilon$. Consider now the weighted graph $G_{r'}$ obtained by setting the weights as above with $r = r'$. Because $r' < r^*$, the optimal cycle C^* with ratio r^* will appear as a negative cycle in $G_{r'}$. Hence, when we run Bellman-Ford on $G_{r'}$, it will detect some negative cycle C (notice C is not necessarily equal to C^*). Then, by a), the profit-cost ratio of C will be greater than r', i.e. $r(C) > r' \geq r^* - \epsilon$. This algorithm requires $\log\left(\frac{R}{\epsilon}\right) + 1$ runs on Bellman-Ford on different weighted versions of G. Its total running time is then $O(\log\left(\frac{R}{\epsilon}\right)|V||E|)$.