CMSC 441: Unit 3 Homework Solutions

November 1, 2017

(15.1-1) We will prove formula (15.4) using induction with base $T(0) = 1$. Since $2^0 = 1$, we see that the formula holds when $n = 0$. For our inductive hypothesis, suppose that $T(k) = 2^k$ for $0 \le k < n$; we will show that $T(n) = 2^n$.

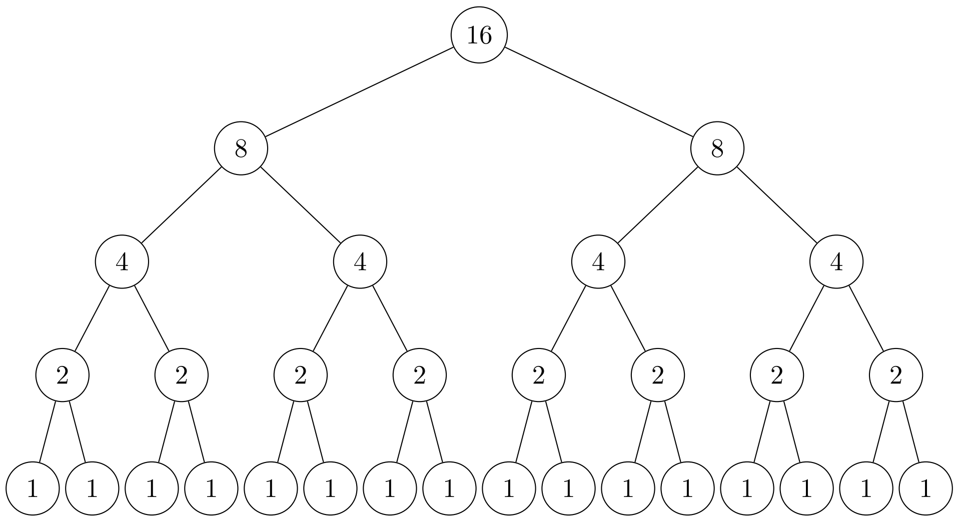
$$\begin{aligned} T(n) &= T(0) + \sum_{j=0}^{n-1} T(j) \\ &= T(0) + \sum_{j=0}^{n-2} T(j) + T(n-1) \\ &= T(n-1) + T(n-1) \\ &= 2 \cdot T(n-1) \\ &= 2^n \end{aligned}$$

The step from line 2 to 3 follows from the definition of $T(n-1)$, and the step from line 4 to 5 follows from the inductive hypothesis.

(15.1-3) Use the algorithm Bottom-Up-Cut-Rod from the textbook as a starting point. If a cut has cost c, we must be able to identify when a cut is required and when it is not, and subtract c from the revenue when appropriate. Looking at lines 5-6, we see that for $1 \le i < j$, we are assuming a cut at position i, but when $i = j$, this corresponds to no cut, and, in fact, the subproblem in this case is $r[0]$. We can re-organize this loop as follows:

```
for $i = 1$ to $j - 1$
    $q = \max(q, p[i] + r[j-i] - c)$
$q = \max(q, p[j])$
```

(15.3-2) A recursion tree for Merge-Sort on 16 elements is given below. Since in most cases each of the nodes is processing different data, there is no gain from memoization. The problem lacks *overlapping subproblems.*



(15.3-5) The two sub-problems can not be solved independently. For example, if I know at most $l_1 > 0$ cuts of length one may be made, there is no way to ensure that my two sub-problems produce a *total* number of cuts of length one that is less than l_1 unless the two sub-problems are able to coordinate their solutions.

(15.4-2) To see that the following algorithm is $O(m+n)$ consider the number of iterations of the while-loop in the worst case. We always decrement *row* or *col* or both, so there can never be more than $m+n$ iterations before the loop terminates.

```
LCS-RECONSTRUCT($C, x, y$)
row = x.length
col = y.length
lcs-length = k = C[row, col]
lcs is a new array of length lcs-length
while row > 0 and col > 0
    if x[row] == y[col]
        lcs[k] = x[row]
        k = k - 1
        row = row - 1
        col = col - 1
    elseif C[row - 1, col] >= C[row, col - 1]
        row = row - 1
    else col = col - 1
return lcs
```

(15-4.4) Since the roles of x and y are interchangeable, assume that the shorter of the two sequences is written across the horizontal axis so that the matrix c has $\min(m,n)$ columns. Recall that the computation proceeds by iterating over a row of c and that the computation of a given row only depends on the previous row and the previous elements in the current row. Once a row has been computed, we may discard the previous row. Thus, the computation can be done using at most $2\min(m,n)$ storage in the c array. $O(1)$ additional storage is needed for loop and temporary variables.

We can reduce the storage requirements further by noting that if we have just computed $c[i,j]$, the remaining elements in the row $(c[i,j+1\,..\,\min(m,n)])$ only require the previous elements in the current row and the elements $c[i-1,j\,..\,\min(m,n)]$ from the previous row. That is a total of j elements from the current row and $\min(m,n)-j$ elements from the previous row, giving a total of $\min(m,n)$ elements of the c array that must be stored.

(16.1-1) The tricky part is determining which activities are in the set S_{ij} . If activity k is in S_{ij}, then we must have $i<k<j$, which means that $j-i\geq 2$, but we must also have that $f_i\leq s_k$ and $f_k\leq s_j$. If we start k at $j-1$ and decrement k, we can stop once k reaches i, but we can also stop once we find that $f_k\leq f_i$, since then activities $i+1$ through k cannot be compatible with activity i.

We create two fictitious activities, a_0 with $f_0=0$ and a_{n+1} with $s_{n+1}=\infty$. We are interested in a maximum-size set $A_{0,n+1}$ of mutually compatible activities in $S_{0,n+1}$. Use tables $c[0\,..\,n+1,0\,..\,n+1]$, as in recurrence (16.2) (so that $c[i,j]=|A_{ij}|$), and $act[0\,..\,n+1,0\,..\,n+1]$, where $act[i,j]$ is the activity k that we choose to put into A_{ij}.

Fill the tables according to increasing difference $j-i$, which we denote by l. Since $S_{ij}=\emptyset$ if $j-i<2$, we initialize $c[i,i]=0$ for all i and $c[i,i+1]=0$ for $0\leq i\leq n$. As in RECURSIVE-ACTIVITY-SELECTOR and GREEDY-ACTIVITY-SELECTOR, the start and finish times are given as arrays s and f, where we assume that the arrays already include the two fictitious activities and that the activities are sorted by monotonically increasing finish time.

(16.1-2) Assume the events have been sorted by finish time. Let S_k be the subproblem consisting of all events that start before event a_k and are compatible with a_k. We must prove the greedy choice property, that there is a solution to the subproblem that includes the greedy choice. Let A_k be an optimal solution for S_k and b_k the event in A_k with the latest start time. Let m_k be the event in S_k with the latest start time. If $b_k=m_k$, we are done, since this means A_k includes the greedy choice. Suppose $b_k\neq m_k$. Then m_k starts later than b_k, and since b_k is compatible with all the other events in A_k, m_k is also compatible will all the other events in A_k. Let $A'_k=A_k-\{b_k\}\cup\{m_k\}$. Then $|A'_k|=|A_k|-1+1=|A_k|$ and A'_k consists of mutually compatible events in S_k, so A'_k is an optimal solution for the subproblem that includes the greedy choice.

Here is a sample implementation in Python:

```
#!/usr/bin/env python

def dynamic_activity_selector(s, f, n):
    c =   [ [ 0 for x in range(n+2) ] for x in range(n+2) ]
    act = [ [ 0 for x in range(n+2) ] for x in range(n+2) ]

    for l in range(2, n+2):
        for i in range(0, n - l + 2):
            j = i + l
            c[i][j] = 0
            k = j - 1
            while f[i] < f[k]:
                if ( f[i] <= s[k] and f[k] <= s[j]
                     and c[i][k] + c[k][j] + 1 > c[i][j] ):
                    c[i][j] = c[i][k] + c[k][j] + 1
                    act[i][j] = k
                k = k - 1
    return c, act

def print_activities(c, act, i, j):
    if c[i][j] > 0:
        k = act[i][j]
        print k
        print_activities(c, act, i, k)
        print_activities(c, act, k, j)

if __name__ == "__main__":

    #    Sample date from textbook with fake activities
    #    a0 and a12

    #    0  1  2  3  4  5  6   7   8   9  10  11    12
    s = [0, 1, 3, 0, 5, 3, 5,  6,  8,  8,  2, 12, 1000]
    f = [0, 4, 5, 6, 7, 9, 9, 10, 11, 12, 14, 16, 1000]

    n = len(s) - 2

    c, act = dynamic_activity_selector(s, f, n)

    print "Number of scheduled activities: " + str(c[0][n+1])
    print "Activities scheduled: "

    print_activities(c, act, 0, n + 1)
```

(16.1-3) For the approach of selecting the activity with the shortest duration, consider three events $a_1 = [7, 10)$, $a_2 = [9, 11)$, and $a_3 = [10, 13)$. The event a_2 has the shortest duration, but choosing it results in only one event being scheduled; clearly, the optimal solution is to schedule a_1 and a_3.

For the approach of always selecting the compatible activity that overlaps the fewest other remaining activities:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	0	1	1	1	2	3	4	5	5	5	6
f_i	2	3	3	3	4	5	6	7	7	7	8
overlapping activities	3	4	4	4	4	2	4	4	4	4	3

This approach first selects a_6, and after that choice it can select only two other activities (one of $\{a_1, a_2, a_3, a_4\}$ and one of $\{a_8, a_9, a_{10}, a_{11}\}$). An optimal solution is $\{a_1, a_5, a_7, a_{11}\}$.

For the approach of always selecting the compatible remaining activity with the earliest start time, just add one more activity with the interval $[0, 14)$ to the example in Section 16.1. It will be the first activity selected, and no other activities are compatible with it.

(16.2-1) We have to show that any subproblem has an optimal solution that includes the greedy choice. Suppose that the items available to place in the knapsack are ordered by increasing unit value, so item 1 has the lowest unit value and item n has the largest. For $m \leq n$, let S_m be the subproblem defined by (1) the knapsack can carry weight $W - w$ where $w = w_{m+1} + w_{m+2} + \cdots + w_n$, and (2) items $1 \mathinner{..} m$ may be chosen to place in the knapsack, and the amount of each that is available is $w_1, w_2, \ldots, w_m$. That is, the top-value items $(m+1 \mathinner{..} n)$ have already been placed in the knapsack. Suppose we have an optimal solution A_m for S_m. If A_m includes weight w_m of item m, we are done, since this is the greedy choice; also, if A_m includes weight $u_m < w_m$ of item m and $W - w - u_m = 0$, we are also done, because A_m includes as much of item m as can be carried, which is still the greedy choice. If the solution contains $u_m < w_m$ of item m and $W - w - u_m > 0$, then it must include some quantity of an item, say item j, with the same unit value, i.e. $v_j = v_m$; if v_j were not equal to v_m, then we could replace some of item j with item m, yielding a solution with greater total value, which is a contradiction. But then we can create a solution A'_m by replacing some or all of item j with item m, and since they have the same unit value A'_m is also an optimal solution. If we have still not used all if item m and the knapsack has additional capacity, then we can repeat this argument to show that there must be an item k with $v_k = v_m$ included in the knapsack, so we can swap some of item k for item m, etc. If there are a number of items with the same unit value, we may have to repeat this argument several times, but eventually we will reach the point where we have constructed a new optimal solution that contains all of item m, or all that can fit before filling the knapsack. Therefore, there is a solution to the subproblem that includes the greedy choice.

(16.2-2) The solution is based on the optimal-substructure observation in the text: let i be the highest-numbered item in an optimal solution S for W pounds and items $1 \dots n$. Then $S' = S - \{i\}$ must be an optimal solution for $W - w_i$ pounds and items $1 \dots i-1$, and the value of the solution S is v_i plus the value of the subproblem solution S'. We can express this relationship in the following formula: define $c[i, w]$ to be the value of the solution for items $1 \dots i$ and maximum weight w. Then (i) $c[i, w] = 0$ if $i = 0$ or $w = 0$; (ii) $c[i, w] = c[i-1, w]$ if $w_i > w$; or (iii) $c[i, w] = \max(v_i + c[i-1, w-w_i], c[i-1, w])$ if $i > 0$ and $w \geq w_i$. The last case says that the value of a solution for i items either includes item i, in which case it is v_i plus a subproblem solution for $i-1$ items and the weight excluding w_i, or doesn?t include item i , in which case it is a subproblem solution for $i-1$ items and the same weight. That is, if the thief picks item i, he takes v_i value, and he can choose from items $1 \dots i-1$ up to the weight limit $w - w_i$, and get $c[i-1, w-w_i]$ additional value. On the other hand, if he decides not to take item i, he can choose from items $1 \dots i-1$ up to the weight limit w, and get $c[i-1, w]$ value. The better of these two choices should be made. The algorithm takes as inputs the maximum weight W, the number of items n, and the two sequences $v = \langle v_1, v_2, \dots, v_n \rangle$ and $w = \langle w_1, w_2, \dots, w_n \rangle$. It stores the $c[i, j]$ values in a table $c[0 \,.\,.\, n, 0 \,.\,.\, W]$ whose entries are computed in row-major order. (That is, the first row of c is filled in from left to right, then the second row, and so on.) At the end of the computation, $c[n, W]$ contains the maximum value the thief can take.

(16.2-4) Suppose the route across North Dakota is L miles long. We can imagine Professor Gekko's route as the interval $[0, L]$ in the real line. Suppose there are n possible water stops, including the end-point of the route, and let x_i be the distance of the i^{th} stop from the start of the route. In addition, let $x_0 = 0$ denote the start of the route. Then the possible water stop satisfy

$$0 = x_0 < x_1 < x_2 < \cdots < x_{n-1} < x_n = L.$$

Since the Professor can only skate m miles on two litres of water, it must be that the water stops are no more than m miles apart, or

$$x_i - x_{i-1} \leq m \text{ for } 1 \leq i \leq n.$$

If not, then the Professor will not be able to complete the trip.

Here is a greedy strategy for minimizing the number of water stops: from the current stop, Professor Gekko should skate to the furthest stop that is at most m miles away. For example, suppose Professor Gekko is starting the trip at x_0 and there are two possible water stops within m miles, x_1 and x_2. Since $x_2 > x_1$, she would skate until x_2 and fill her water bottles there. To prove that this greedy strategy will work, we need to define subproblems and show that the greedy choice can lead to an optimal solution for any subproblem.

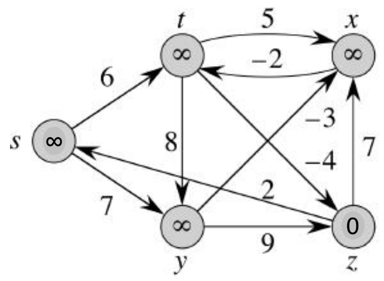
Let $S_i = \{x_i, x_{i+1}, \dots, x_n\}$, $0 \leq i \leq n$. S_i is just the subproblem of determining the optimal water stops when skating from x_i to x_n. Suppose A_i is an optimal solution for S_i, meaning that A_i consists of a subset of S_i that includes x_i, the successive stops in S_i are no more than m miles apart, and the number of stops is minimized. suppose the first stop in A_i, which we will denote by y, is *not* the greedy choice. Then there is some $y' \in S_i$ such that $y' > y$ and $y' - x_i \leq m$ (she could have skated further to y', but chose to stop at y). Define a new set A'_i by replaceing y in A_i with y',

$$A'_i = A_i \setminus \{y\} \cup \{y'\}.$$

Since $y' > y$ and $y' - x_i \leq m$, all the stops in A'_i are within m miles of each other, and the number of stops in A'_i is the same as in A_i. Since A_i was assumed optimal, A'_i is also optimal, and includes the greedy choice. Therefore, the greedy choice can lead to an optimal solution.

Finally, we also need to establish that the problem has optimal substructure. Suppose we have an optimal solution $A_i = \{x_i, x_{i+d_1}, x_{i+d_2}, \ldots, x_{i+d_k}\}$ for the subproblem S_i. We claim that $B = \{x_{i+d_1}, x_{i+d_2}, \ldots, x_{i+d_k}\}$ is an optimal solution for S_{i+d_1}. Since the distance between successive elements of A_i is at most m, the distance between successive elements of B is at most m. Also, there can not be a smaller set of stops for S_{i+d_1}, for, if there were such a solution C, we could construct $A'_i = C \cup \{x_i\}$, in which all the stops are at most m miles apart and there are fewer stops than in A_i, which was assumed to be optimal.

(24.1-1)(a) Show the operation of the Bellman-Ford algorithm using z as the source.



Iteration 1

- (t, x): N/C
- (t, y): N/C
- (t, z): N/C
- (x, t): N/C
- (y, x): N/C

- (y, z): N/C
- (z, x): $x.d \leftarrow 0 + 7 = 7$, $x.\pi = z$
- (z, s): $s.d \leftarrow 0 + 2 = 2$, $s.\pi = z$
- (s, t): $t.d \leftarrow 2 + 6 = 8$, $t.\pi = s$
- (s, y): $y.d \leftarrow 2 + 7 = 9$, $y.\pi = s$

From now on, we will only list the edges that cause a change to the path weights.

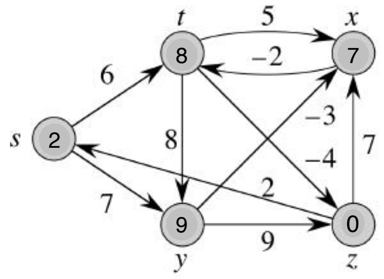


Figure 1: After Iteration 1

Iteration 2

- (x, t): $t.d \leftarrow 7 - 2 = 5$, $t.\pi = x$
- (y, x): $x.d \leftarrow 9 - 3 = 6$, $x.\pi = y$

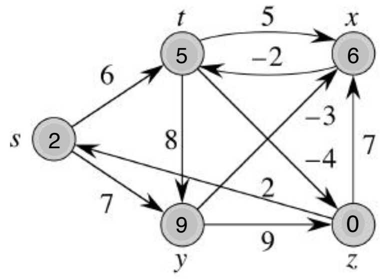


Figure 2: After Iteration 2

Iteration 3

- (x, t): $t.d \leftarrow 6 - 2 = 4$, $t.\pi = x$

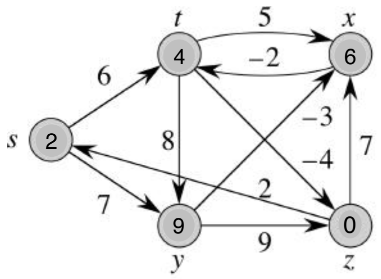
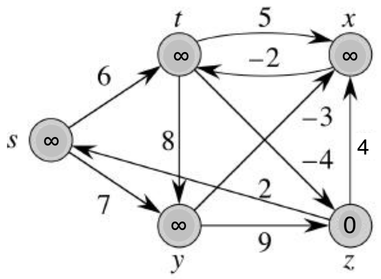


Figure 3: After Iterations 3 and 4

Iteration 4

- There are no changes in the fourth iteration.

(b) Show the operation of the Bellman-Ford algorithm using z as the source and changing the weight of (z, x) to 4.



Iteration 1

- (z, x): $x.d \leftarrow 0 + 4 = 4$, $x.\pi = z$
- (z, s): $s.d \leftarrow 0 + 2 = 2$, $s.\pi = z$
- (s, t): $t.d \leftarrow 2 + 6 = 8$, $t.\pi = s$
- (s, y): $y.d \leftarrow 2 + 7 = 9$, $y.\pi = s$

Iteration 2

- (x, t): $t.d \leftarrow 4 - 2 = 2$, $t.\pi = x$

Iteration 3

- (t, z): $z.d \leftarrow 2 - 4 = -2$, $z.\pi = t$

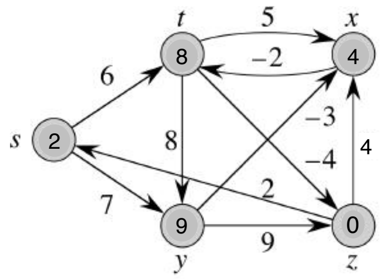


Figure 4: After Iteration 1

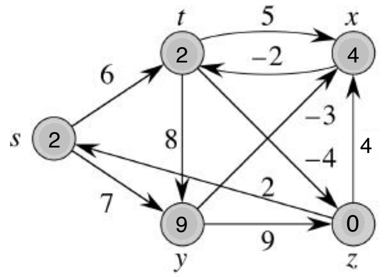


Figure 5: After Iteration 2

- (z, x): $x.d \leftarrow -2 + 4 = 2$, $x.\pi = z$
- (z, s): $s.d \leftarrow -2 + 2 = 0$, $s.\pi = z$
- (s, y): $y.d \leftarrow 0 + 7 = 7$, $y.\pi = s$

Iteration 4

- (x, t): $t.d \leftarrow 2 - 2 = 0$, $t.\pi = x$

Note that the edge (t, z) can be further relaxed, so there must be a negative weight cycle in the graph and Bellman-Ford will return False. In fact, the cycle $t \to z \to x \to t$ has weight -2.

(24.1-3) On each pass, the algorithm relaxes every edge in the graph. After m iterations, the algorithm will have fully relaxed all the edges in every shortest path from s (by the upper bound property). Since we are assuming there are no negative weight cycles reachable from the source, this means the path weights will not change on the $(m+1)^{st}$ pass. If we modify the Relax routine to flag whether or not any path weights have been modified, we can detect when there are no more changes, and thus can stop the algorithm after $m + 1$ passes.

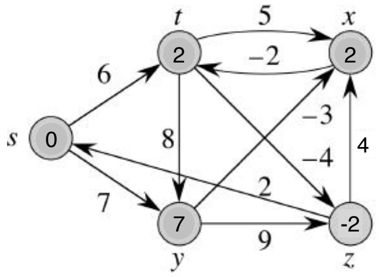


Figure 6: After Iteration 3

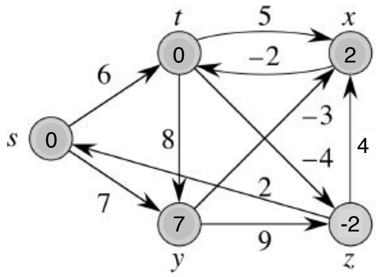
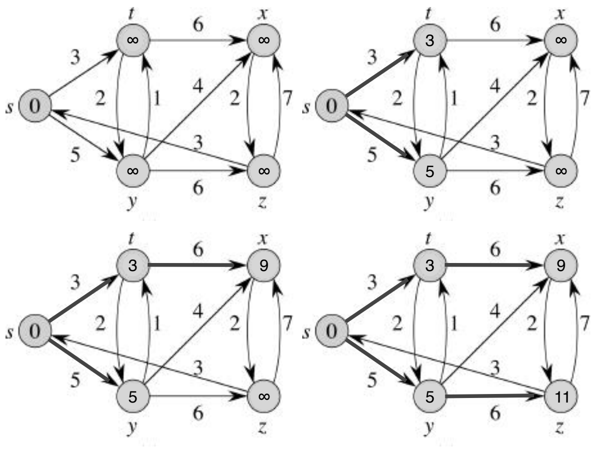


Figure 7: After Iteration 4

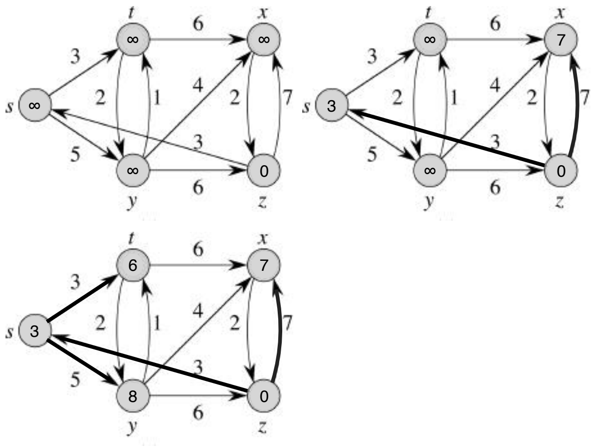
```
BELLMAN-FORD-MOD($G, w, s$)
1  INITIALIZE-SINGLE-SOURCE($G, s$)
2  $changes$ = TRUE
3  while $changes$ == TRUE
4      $changes$ = FALSE
5      for each edge $(u, v) \in G.E$
6          RELAX-MOD(u, v, w)
```

```
RELAX-MOD($u, v, w$)
1  if $v.d > u.d + w(u, v)$
2      $v.d = u.d + w(u, v)$
3      $v.\pi = u$
4      $changes$ = TRUE
```

(24.3-1) Using s as source, we first process edges out of s. On the second iteration, t has the smallest path-length, so we process its outgoing edges. Third is y, fourth is x, and finally z. There are no changes to the path lengths when we process x and z.



Using z as source, we first process the edges out of z, then s, then t, y, and x. There are no changes to the path lengths when processing t, y, and x.



(24.3-6) We can not use Dijkstra's algorithm directly on the problem as stated. We are asked to find the path that *maximizes* the probability that a transmission from s to v succeeds, wheras Dijkstra's algorithm minimizes the path length. Also, the probability that a path succees is the product of the edge probabilities, not the sum. Stated mathematically, we wish to find the path from s to v, $s = v_0 \to v_1 \to \cdots \to v_k = v$ that maximizes

$$\prod_{i=1}^{k} r(v_{i-1}, v_i).$$

This is equivalent to minimizing

$$\frac{1}{\prod_{i=1}^{k} r(v_{i-1}, v_i)} = \prod_{i=1}^{k} \frac{1}{r(v_{i-1}, v_i)} = \prod_{i=1}^{k} r(v_{i-1}, v_i)^{-1},$$

and this, in turn, is equivalent to minimizing the logarithm of the last product, which is

$$\sum_{i=1}^{k} \log\left(r(v_{i-1}, v_i)^{-1}\right) = -\sum_{i=1}^{k} \log\left(r(v_{i-1}, v_i)\right).$$

So we can just set $w(u, v) = -\log\left(r(v_{i-1}, v_i)\right)$ and apply Dijkstra's algorithm to G with weights w.

(24.3-8) Looking at the pseudocode for Dijkstra's algorithm, we see that the primary costs are $|V|$ calls to EXTRACT-MIN and calling RELAX for each of the $|E|$ edges. To achieve a running time of $O(W|V|+|E|)$, we must replace the usual EXTRACT-MIN with some $O(W|V|)$ method. Assuming we can maintain the $O(1)$ running time for RELAX, this will give $O(W|V|)+O(|E|) = O(W|V|+|E|)$ running time.

A simple path can have at most $|V| - 1$ edges, so if each edge has maximum weight W, the maximum weight of any path is $(|V| - 1)W$. Also, note that the path-lengths returned by EXTRACT-MIN are monotonically increasing: if $u.d$ is the current minimum length, then for any v, $v.d \geq u.d$ (or $v.d$ would have been returned by EXTRACT-MIN) and even after the edge (u, v) is relaxed, $v.d \geq u.d$ since all edge weights are non-negative.

Create an array of pointers, A, of length $k = W(|V| - 1) + 2$; the entries of the array correspond to the possible path weights $0, 1, \ldots, W(|V| - 1), \infty$. Modify the node data structure so that, in addition to variables $x.d$ and $x.\pi$, it has $x.prev$ and $x.next$, which are the previous and next pointers for a linked list. Initialize the array so that $A[0]$ points to the one-element linked list containing only the node s and all other nodes are in a linked list pointed to by $A[k - 1]$ (which corresponds to weight ∞). All other entries of A are initialized to NIL. Create a variable *min* to point to the smallest non-NIL entry in A, and initialize $min = 0$. The initialization procedure can be performed in $O(|V| + W(|V| -$

$1)) = O(W|V|)$ time since inserting to the head of a linked list is $O(1)$ and setting the remaining entries of A to NIL is $O(W(|V|-1))$.

Implement a new EXTRACT-MIN as follows. If $A[min]$ is non-NIL, return the node at the head of $A[min]$'s linked list and remove it from the list. If $A[min]$ is NIL, then the last element of the linked list must already have been extracted, so increment min until we find a non-NIL entry of A; if we do not find a non-NIL entry, the priority-queue is empty. Incrementing min will produce the correct result since the weigths returned by the usual EXTRACT-MIN are monotonically increasing. Since there are initially $|V|$ nodes inserted into the data structure, there are at most $|V|$ nodes that must be returned and min is incremented at most k times, so the total running time for all calls to EXTRACT-MIN is $O(|V|+k) = O(|V|+W|V|) = O(W|V|)$.

The REDUCE-KEY procedure, used by RELAX, can be implemented with $O(1)$ running time. If a node's $x.d$ value is reduced, remove the node from its current linked list and add it to the head of the linked list corresponding to the new value of $x.d$. Since the list is doubly-linked, both of these operations can be performed in $O(1)$ time, which preserves the $O(1)$ running time for RELAX.

Therefore, the overall running time is $O(W|V|+|E|)$.