CSE 101 Homework 2 Answers

Note that you wouldn't necessarily be expected to give the same answers as those given here. Also, no answer is given for the run-this-algorithm-on-this-example problem - this is fairly straightforward and follows the example in the book; ask a TA if you have any questions.

1. (Exercise 16.1-1)

2. This problem is similar to LCS (Section 16.3), and the solution is similar. For this problem, we will describe optimal substructure in terms of suffixes of the string (substrings that start at an arbitrary point in the string, and include all characters from that point until the end of the string), rather than prefixes as was done for LCS.

   Let X and Y be strings. Denote an optimal transformation sequence to turn source X into target Y by $S(X, Y)$, and let $E(Q)$ be the sum of the costs of the operations in an arbitrary transformation sequence Q (so $E(S(X, Y))$ is the edit distance from X to Y). Let $(Q)(R)$ denote the transformation sequence obtained by concatenating transformation sequence Q to transformation sequence R. For notational simplicity below, define a special transformation sequence NULL with $E(\text{NULL}) = \infty$ (this is used in cases below when a particular type of transformation sequence does not apply). Let ϵ denote the empty string, and let $|X|$ denote the length of string X (so $|\epsilon| = 0$). Let X_i be the suffix of X that begins at the i'th character of X, and let x_i be the i'th character of X. The optimal substructure analogous to that given in Theorem 16.1 can now be stated as:

   If $|X_i| = 0$ and $|Y_j| = 0$, then $S(X_i, Y_j)$ is the empty sequence. Otherwise,

   $$S(X_i, Y_j) = \arg \min_{Q \in \{C,T,R,D,I,K\}} E(Q)$$

   where

   (a) If $|X_i| \geq 1$ and $|Y_j| \geq 1$ and $x_i = y_j$, then $C = (\text{copy})(S(X_{i+1}, Y_{j+1}))$, else $C = \text{NULL}$.

   (b) If $|X_i| \geq 2$ and $|Y_j| \geq 2$ and $x_i = y_{j+1}$ and $x_{i+1} = y_j$, then $T = (\text{twiddle})(S(X_{i+2}, Y_{j+2}))$, else $T = \text{NULL}$.

   (c) If $|X_i| \geq 1$ and $|Y_j| \geq 1$, then $R = (\text{replace})(S(X_{i+1}, Y_{j+1}))$, else $R = NULL$.

   (d) If $|X_i| \geq 1$, then $D = (\text{delete})(S(X_{i+1}, Y_j))$, else $D = NULL$.

   (e) If $|Y_j| \geq 1$, then $I = (\text{insert})(S(X_i, Y_{j+1}))$, else $Y = NULL$.

   (f) $K = (\text{kill})(S(\epsilon, Y_j))$.

   In words, this says that the optimal transformation sequence is obtained by dividing into six cases based on the *first* operation of this transformation sequence, filling out the rest of the transformation sequence by taking the optimal transformation for the remaining suffixes, and minimizing over the cost of these six cases. Since at least one of these remaining suffixes is shorter, and neither is longer, than the original, we are reducing the problem to the solution of a subproblem.

   To prove the above statement, first note that if $|X_i| = 0$ and $|Y_j| = 0$ the transformation is completed, so the empty transformation sequence is optimal. Otherwise, let Q be an optimal transformation sequence from X_i to Y_j; we want to show that the cost of the transformation sequence described above is no worse than that of Q. There are six cases, one for each possible choice of first operation of Q, but they are all essentially the same so we describe them here in a unified way. Let Q_1 be the first operation in Q, and let X' and Y' be the remaining suffixes (given above for each operation) after Q_1 is applied. Let Q' be the remainder of Q, so that $Q = (Q_1)(Q')$; Q' must transform X' into Y' for the transformation sequence to be correct. Assume towards a contradiction that Q' is not optimal, so that $E(Q') > E(S(X', Y'))$. Then we could replace Q' by $S(X', Y')$ to get $(Q_1)(S(X', Y'))$, a transformation sequence from X_i to Y_j with cost $E(Q_1) + E(S(X', Y')) < E(Q_1) + E(Q') = E(Q)$, which contradicts the optimality of Q. So, Q' must have the same cost as $S(X', Y')$, so Q must have total cost equal to the cost of the transformation sequence given in the case above corresponding to

Q_1. Since the transformation sequence given by the above statement has minimum cost over possible choices of first operation, its cost is no worse than that of Q, and it is therefore optimal.

Giving an algorithm based on this optimal substructure, we iterate bottom-up over successively longer suffixes of X and Y, so that whenever we need the solution to a subproblem consisting of shorter suffixes, that subproblem solution is already available. For each particular pair of suffixes, we choose the initial operation that minimizes total cost, as described by the optimal substructure property above. In the algorithm below, X_{m+1}, where $m = |X|$, is the empty suffix consisting of zero characters. $e[i, j]$ is an array holding the edit distance from X_i to Y_j for each i and j, and $s[i, j]$ holds the first operation in the optimal transformation sequence from X_i to Y_j. As each operation is tried, *curcost* holds the cost of the transformation sequence starting with that operation. A minimum is taken over these, and the minimum for the operations tried so far is held in $e[i, j]$ (which starts at ∞ since we are minimizing).

```
EDIT-DISTANCE(X, Y)
m ← |X|
n ← |Y|
for i ← m + 1 downto 1
     do for j ← n + 1 downto 1
          if i = m + 1 and j = n + 1
               then e[i, j] ← 0
               else e[i, j] ← ∞
                    curcost ← cost(kill) +e[m + 1, j]
                    if curcost < e[i,j]
                         then s[i, j] ← kill
                              e[i, j] ← curcost
                    if j < n + 1
                         then curcost ← cost(insert) +e[i, j + 1]
                              if curcost < e[i,j]
                                   then s[i, j] ← insert
                                        e[i, j] ← curcost
                    if i < m + 1
                         then curcost ← cost(delete) +e[i + 1, j]
                              if curcost < e[i,j]
                                   then s[i, j] ← delete
                                        e[i, j] ← curcost
                    if i < m + 1 and j < n + 1
                         then curcost ← cost(replace) +e[i + 1, j + 1]
                              if curcost < e[i,j]
                                   then s[i, j] ← replace
                                        e[i, j] ← curcost
                    if i < m + 1 and j < n + 1 and x_i = y_j
                         then curcost ← cost(copy) +e[i + 1, j + 1]
                              if curcost < e[i,j]
                                   then s[i, j] ← copy
                                        e[i, j] ← curcost
                    if i < m and j < n and x_i = y_{j+1} and x_{i+1} = y_j
                         then curcost ← cost(twiddle) +e[i + 2, j + 2]
                              if curcost < e[i,j]
                                   then s[i, j] ← twiddle
                                        e[i, j] ← curcost
return e and s
```

After this algorithm is executed, $e[1, 1]$ holds the edit distance from X to Y. As with LCS, a separate algorithm takes the resulting array (s, here) and prints the optimal sequence. This algorithm should be initially called with $i = 1$ and $j = 1$; these specify the suffixes of X and Y, respectively, for which the transformation sequence is to be printed.

```
EDIT-PRINT(X, Y, s, i, j)
if i < |X| + 1 or j < |Y| + 1
    then print s[i, j]
        if s[i, j] = twiddle
            then EDIT-PRINT(X, Y, s, i + 2, j + 2)
        if s[i, j] = copy or s[i, j] = replace
            then EDIT-PRINT(X, Y, s, i + 1, j + 1)
        if s[i, j] = insert
            then EDIT-PRINT(X, Y, s, i, j + 1)
        if s[i, j] = delete
            then EDIT-PRINT(X, Y, s, i + 1, j)
        if s[i, j] = kill
            then EDIT-PRINT(X, Y, s, |X| + 1, j)
```

EDIT-DISTANCE loops through suffixes of X and Y, doing constant work for each pair of suffixes, for a total of $\Theta(|X||Y|)$ time. EDIT-PRINT does constant work inside each call, and then makes a recursive call that increases at least one of i and j by one. Neither i nor j can increase beyond the length of the respective string plus one, so total time for EDIT-PRINT is $O(|X| + |Y|)$.

3. Least-duration: say there are 3 activities. Activity 1 starts at time 1 and finishes at time 5, activity 2 starts at time 4 and finishes at time 7, and activity 3 starts at time 6 and finishes at time 10. Choosing activities by least-duration, activity 2 gets chosen first; activities 1 and 3 are incompatible with activity 2 so no further activities can be chosen; the least-duration algorithm yields a schedule with only one activity. This is not optimal, since a better schedule consists of the activities 1 and 3.

   Least-overlap: this strategy chooses the activity that overlaps with fewest other remaining activities, then eliminates all activities that are incompatible with the chosen activity. Say there are 11 activities, as follows:

Activity	Start time	End time
1	1	3
2	2	5
3	2	5
4	2	5
5	4	7
6	6	9
7	8	11
8	10	13
9	10	13
10	10	13
11	12	14

   (It helps to draw these to visualize what is going on.) Note that activities 2-4 are identical, and activities 8-10 are identical. Initially, all activities remain because none have been chosen yet. The least-overlap strategy will choose activity 6 first, because it overlaps with only two other activities (5 and 7); all other activities overlap with at least three. After activity 6 is chosen, 5 and 7 no longer remain compatible. After this, only one of activities 1-4 can be chosen, since these are mutually incompatible. Similarly, only one of activities 8-11 can be chosen. So, the final schedule produced by the least-overlap strategy includes activity 6, one of activities 1-4, and one of activities 8-11, for a total of 3 activities. This is not optimal; a better schedule consists of activities 1,5,7, and 11.

4. (a) To make change for n cents, we choose the largest coin that will fit into n cents (that has denomination $g \leq n$), and then greedily solve the subproblem of making change for the remaining number of cents $n - g$. The proof of optimality below reasons about this incremental version of the algorithm, that chooses one coin at a time. For the sake of efficiency, though, we can save time by dividing n by the denomination of the largest coin that fits into n, to see how many times that largest coin can be chosen. This is then repeated for the next-smaller denomination, and so on, yielding a faster equivalent algorithm:

CHANGE(n)
$\quad$ *quarters* $\leftarrow \lfloor n/25 \rfloor$
$\quad$ $r \leftarrow n \bmod 25$
$\quad$ *dimes* $\leftarrow \lfloor r/10 \rfloor$
$\quad$ $r \leftarrow r \bmod 10$
$\quad$ *nickels* $\leftarrow \lfloor r/5 \rfloor$
$\quad$ $r \leftarrow r \bmod 5$
$\quad$ *pennies* $\leftarrow r$
$\quad$ **return** *quarters, dimes, nickels, pennies*

Each line uses a single arithmetic operation, assumed to take constant time in this problem, for a total time of $O(1)$.

To prove that this produces an optimal solution, we follow the proof of theorem 17.1. This proof has two parts. Given n cents, we first show that there is an optimal solution that includes the greedy coin (coin with largest denomination $\leq n$) chosen by the algorithm. Then, we show that this optimal solution consists of this greedy g-cent coin, plus an optimal solution to the remaining subproblem solved by algorithm: that of making change for the remaining $n - g$ cents.

For the first part, assume that we could make optimal change for n cents with an optimal solution S that does not use the greedy coin g. Say the largest coin used by S is $g' \neq g$. There are several cases (one for each coin that g might be).

If g is a penny, then $n \leq 4$, so g' cannot be anything besides a penny.

If g is a nickel, then $5 \leq n < 10$, so g' is not a dime or quarter. Since we assumed $g' \neq g$, g' is also not a nickel, so it must be a penny. Since g' is the largest-denomination coin in S, S consists entirely of n pennies. Since $n \geq 5$, we could replace 5 pennies of S by a single nickel, yielding an improved solution that consists of fewer coins, contradicting the optimality of S.

If g is a dime, then $10 \leq n < 25$, so g' is a nickel or a penny. If it is a penny, then S consists entirely of pennies as above, and since $n \geq 10$ we could replace 5 pennies of S by a single nickel to get an improved solution, contradicting the optimality of S. If g' is a nickel, then S either includes at least one more nickel, or at least 5 pennies, to make up the remaining $n - 5 \geq 5$ cents. In either case, we can remove exactly 10 cents by removing at least two coins from S, and replace these by a single dime to get an improved solution, again contradicting the optimality of S.

Finally, if g is a quarter, then $n \geq 25$, and g' is a nickel, dime, or penny. If g' is a penny, then we can replace 5 pennies of S with a single nickel as above to get an improved solution, contradicting the optimality of S. If g' is a nickel, then S either includes at least one more nickel, or at least 5 pennies, as above. In either case, we can replace exactly 10 cents with a single dime to get an improved solution, again contradicting the optimality of S. For the final possibility, g' is a dime, and the rest of S consists of change for the remaining $n - 10 \geq 15$ cents. To make up at least 15 cents, this remaining change must include:

(a) At least one dime and at least one nickel, or

(b) At least one dime and at least five pennies, or

(c) At least three nickels, or

(d) At least two nickels and at least five pennies, or

(e) At least one nickel and at least ten pennies, or

(f) At least fifteen pennies

since these are all of the ways to make change for 15 cents. In each of these cases, by removing g' and the coins specified above from S, we remove exactly 25 cents from S and can replace it with a single quarter to get an improved solution using fewer coins, contradicting the optimality of S.

These are all the possibilities for g; in each case, we reach a contradiction by assuming an optimal solution that does not include g. Therefore, an optimal solution must include the greedy coin g.

For the second part of the proof, we wish to show that the optimal solution S includes optimal change for $n-g$ cents, where g is the largest coin in S (and also the greedy coin, as shown above). By removing g from S, we do get change T for $n-g$ cents. Assume this change is not optimal; that there is some change T' for $n-g$ cents that uses fewer coins. Then we could add g to T' to get change for n cents using fewer coins than S, contradicting its optimality.

So, the optimal solution S for n cents consists of the greedy coin g, plus optimal change for $n-g$ cents. By induction on the number of coins chosen by the CHANGE algorithm, it produces optimal change for n cents.

(c) For coins with denominations 1 cent, 4 cents, and 5 cents, the greedy algorithm does not always yield an optimal solution. Consider the problem of making change for 13 cents. The greedy algorithm will choose two five-cent coins, leaving 3 cents that must be handled with three one-cent coins; the greedy algorithm uses a total of five coins. But a better solution consists of three four-cent coins and one one-cent coin, for a total of four coins.

5. FASTER-ALL-PAIRS-SHORTEST-PATHS uses EXTEND-SHORTEST-PATHS repeatedly to find paths using an increasing number of edges. If there is a negative-weight cycle, then there will be a path from vertex i on this cycle, back to vertex i, using at most $n-1$ edges (by the definition of a cycle), with a total weight that is negative. Let e be the number of edges in the shortest (fewest edges) negative-weight cycle. When FASTER calls EXTEND on line 5 when $2m \geq e$ for the first time, EXTEND-SHORTEST-PATHS will find the path from i back to i that has less weight than the default path of weight 0; this will be reflected by a weight of less than 0 on the diagonal in the matrix returned by EXTEND. So, we insert the following between lines 5 and 6 of FASTER:

```
for i ← 1 to n
    do if D^(2m)[i, i] < 0
        then return NEGATIVE-CYCLE-FLAG
```

The first time a negative-weight cycle is found, this terminates the algorithm, returning a flag indicating this fact. Before any negative-weight cycle is found, all cycles have positive weight and the diagonal elements all remain 0, so execution of FASTER is unchanged. If there is no negative-weight cycle in the graph, the above condition will never be triggered and the algorithm finds all shortest paths as usual. But if there is any negative-weight cycle, there is a negative-weight cycle using at most $n-1$ edges, yielding a negative path weight along the diagonal which is detected by the above change.

6. In this problem, we score spanning trees in a different way. Rather than scoring them by total weight on the edges used, we score them by the maximum weight among the edges used. That is, if T is a spanning tree (a set of edges defining a tree that spans the graph), then

$$\text{score}(T) = \max_{\text{edge } (a,b) \in T} w(a, b).$$

It turns out that the original MST algorithms solve this problem, unchanged; safe edges under the original type of score are also safe edges under the new scoring. To see this, reconsider Theorem 24.1

of the textbook, which shows that a light edge (u, v) crossing a cut that respects A is safe for A. Let T be a minimum (under the *new* type of score used in this problem) spanning tree that includes A, and does not include (u, v), and define the new spanning tree T' as in the proof of theorem 24.1. Now, under the new scoring of spanning trees,

$$\mathrm{score}(T') = \max_{\text{edge } (a,b) \in T - (x,y) \cup (u,v)} w(a, b)$$

$$\mathrm{score}(T') \leq \max_{\text{edge } (a,b) \in T} w(a, b)$$

since $w(u, v) \leq w(x, y)$. This last expression is $\mathrm{score}(T)$, so

$$\mathrm{score}(T') \leq \mathrm{score}(T).$$

Since T is optimal, so is T'; so (u, v) is included in some minimum spanning tree (under the new score), and is therefore safe for A.

The correctness of Kruskal's and Prim's algorithms depends simply on this method for finding safe edges, so these algorithms remain correct under the new scoring for spanning trees. Choosing Prim's algorithm, using a Fibonacci heap to implement the priority queue, we get a running time of $O(E + V \lg V)$.