Algorithms: CSE 202 — Homework 2 Solutions

Problem 1: Maximum Length Chain of Subwords

Input: We are given a set of n distinct strings of length at most k over a finite alphabet σ.
Output: A sequence of strings that form a chain under the (consecutive) subword relation; i.e., if the output is $w_1, w_2, \ldots, w_t$ then we can write $w_{i+1} = uw_iv$ for some strings u, v.

Find a chain of maximum length.

Solution: Maximum length chain of words

Sort the words according to their length. Break ties arbitrarily if two words have the same length. Let $w_1, \ldots, w_n$ be the sequence of words after sorting.

For $1 \leq i, j \leq n$, we say that w_i is a *subword* of w_j if w_j can be written as uw_iv for some strings u and v. If $i \neq j$, either u or v or both must be nonempty since all the words are distinct. Note that if w_i is a subword of w_j, then $i \leq j$ since we ordered the words according to their length. We say that a sequence of words is a chain of subwords if each word is a subword of the following word. Following these observations, we define the following problem schema for $1 \leq i \leq n$.

$$M(i) = \text{ maximum length of a subsequence in the sequence } w_1, \ldots, w_i \text{ that form a chain of subwords which ends at } w_i$$

$M(n)$ is the solution to our problem. For $0 \leq i \leq n$, $M(i)$ can be computed by the following recursive formulation.

$$M(1) = 1$$
$$\text{For } i \geq 2, M(i) = \max_{1 \leq j \leq i-1 \text{ and } w_j \text{ is a subword of } w_i} M(j) + 1 \text{if there is a } 1 \leq j \leq i-1 \text{ such that } w_j \text{ is a subword of } w_i$$

Otherwise, $M(i) = 1$

Complexity: The algorithm takes $O(n^2k)$ time since since the computation of $M(i)$ takes at most $O(ik)$ steps. Here we use the standard algorithms for checking whether a string is a subword of another word which takes linear time in the length of the longest word.
Correctness: We prove by induction that for each $n \geq i \geq 1$, $M(i)$ computes the maximum length of the longest chain of words from among the words $w_1, w_2, \ldots, w_i$.

For $i = 1$, there is only one word and it forms a chain of length one ending at w_1. Hence, $M(1) = 1$.

Assume that $M(i)$ is computed correctly for $1 \leq j \leq i-1$. We will show that $M(i)$ is computed correctly (according to its definition) by the recursive formulation. Consider a chain of words from among the words $w_1, \ldots, w_i$ that ends in w_i. Let j be the index of the immediately preceding word in the chain. $M(j)$ is the length of the longest chain of subwords that ends at w_j from among the words $w_1, \ldots, w_j$. Since no word w_l can be a subword of w_j for $l > j$, we conclude that $M(j)$ is the length of the longest chain of subwords from among the words $w_1, \ldots, w_i$ that ends in w_j. Therefore, $M(j) + 1$ is the maximum length among all the chains that end at w_i with w_j as the immediate predecessor where the words in the chain are from the sequence $w_1, \ldots, w_i$. Since $M(i)$ is set to the maximum of $1 + M(j)$ where j ranges over all predecessor subword indices, we conclude that $M(i)$ is the length of the longest chain of words that ends at w_i assuming that there is a predecessor in the chain.

If there is no predecessor in the chain, then it is clear that $M(i) = 1$. Therefore, the recursive formulation computes $M(i)$ correctly in either case. This completes the induction proof.

Problem 2: Business plan

Consider the following problem. You are designing the business plan for a start-up company. You have identified n possible projects for your company, and for, $1 \leq i \leq n$, let $c_i > 0$ be the minimum capital required to start the project i and $p_i > 0$ be the profit after the project is completed. You also know your initial capital $C_0 > 0$. You want to perform at most k, $1 \leq k \leq n$, projects before the IPO and want to maximize your total capital at the IPO. Your company cannot perform the same project twice.

In other words, you want to pick a list of up to k distinct projects, $i_1, \ldots, i_{k'}$ with $k' \leq k$. Your *accumulated capital* after completing the project i_j will be $C_j = C_0 + \sum_{h=1}^{h=j} p_{i_h}$. The sequence must satisfy the constraint that you have sufficient capital to start the project i_{j+1} after completing the first j projects, i.e., $C_j \geq c_{i_{j+1}}$ for each $j = 0, \ldots, k'-1$. You want to maximize the final amount of capital, $C_{k'}$.

Solution: Business plan

Algorithm: We select, start and complete projects in a sequence. The next project is selected after the previous project has been completed. Assume the projects $s_1, s_2, \ldots, s_j$ have already been selected, started and completed. Let C_j denote the available capital after the project s_j has been completed and before the next project is selected. A project l is *eligible* at time j if it has not been selected so far and its capital requirement $c_l \leq C_j$. If $j \leq k-1$, select the next project using the following greedy strategy. Among all eligible projects, select the project with the maximum profit. If there is no such project, stop the selection process.

We maintain a heap of all eligible projects ordered by profit so that the project with the maximum profit is at the top of the heap. After completing a project, if the heap is not empty, we select an eligible project with the maximum profit, remove it from the heap and continue. Observe that C_j is increasing with j as profits are nonnegative. As projects get completed, additional projects may become eligible in which case we insert them into the heap. A project never leaves the heap unless it is selected for execution. The heap can be managed with $O(n \lg n)$ time.

Let S^g be the sequence of projects selected by the greedy strategy. We show that S^g is an optimal sequence of projects. We use induction on the number of projects to prove the correctness. In particular, we show that replacing any other choice with the greedy choice in the sequence will produce at least as a good a solution as the optimal solution.

For a feasible sequence of projects $T = t_1, \ldots, t_l$, let value$(T) = C_0 + p_{t_1} + \cdots + p_{t_l}$ denote the capital after all the projects in the sequence have been completed. For an empty sequence T, we define its value as value$(T) = C_0$.

Claim 1. *The greedy strategy produces an optimal solution on every instance.*

Proof. We obtain a contradiction by assuming that there is an instance on which the greedy strategy fails to produce an optimal solution.

Let n be the smallest integer such that there exists an instance of size n for which the greedy strategy does not produce an optimal solution. We argue that $n \geq 2$. If $n = 1$ there is only one feasible solution. If the initial capital is large enough, we select and execute the project. Otherwise, no project is selected. Hence the greedy solution is optimal if there is only one project.

Let I be an instance of size n such that the greedy algorithm fails to produce an optimal solution on I. We are given $k, C_0, c_1, \ldots, c_n, p_1, \ldots, p_n$. Let $S^g = s_1, s_2, \ldots, s_{k'}$ be the greedy solution for I for some $0 \leq k' \leq k$ where s_i is the i-th project selected by the algorithm. If $k' = 0$, S^g is an empty sequence. Observe that the sequence S^g satisfies the following property: for all $1 \leq j < l$, s_{j+1} is the project with the largest profit among all eligible projects at the completion of the first j projects in S^g.

Let $S_1^g = s_2, \ldots, s_{k'}$.

Let $T = t_1, \ldots, t_{k''}$ be an optimal sequence of projects. Without loss of generality, assume that the projects in T are ordered so that project t_{j+1} is the project with the maximum profit among all projects which are eligible at the time the first j projects in the sequence T are completed. Let $T_1 = t_2, \ldots, t_{k''}$.

Case I — $s_1 = t_1$: Consider the instance I' of size $n-1$ obtained from I by deleting the project s_1 and setting its initial capital to $C_1 = C_0 + p_{s_1}$ and the number of projects to be performed to $k-1$. Observe that the greedy strategy produces the solution S_1^g on I' (explain why), which by assumption, is an optimal solution. Also, we have that T_1 is a feasible solution for I' and value(T) = value(T_1) (explain why). We get

$$\begin{aligned}\text{value}(S^g) &= C_0 + p_{s_1} + p_{s_2} + \cdots + p_{s'_k}\\ &= \text{value}(S_1^g)\\ &\geq \text{value}(T_1)\\ &= \text{value}(T)\end{aligned}$$

which shows that S^g is indeed an optimal solution which leads to a contradiction.

Case II — $s_1 \neq t_1$**:**
Let $T' = s_1, t_2, \ldots, t_{k''}$ if s_1 is not in the sequence T. Otherwise, let T' be the sequence obtained from T by swapping t_1 with s_1. In either case, T' is a feasible solution for I (explain why). Moreover value(T') $\geq$ value(T) since $p_{s_1} \geq p_{t_1}$. This implies that T' is an optimal solution.

Now we are in the situation where the first choices of S^g and the optimal solution T' coincide which leads to a contradiction to the fact that S^g is the smallest instance on which the greedy strategy fails.

□

Complexity; The algorithms runs in time $O(n \lg n)$ since the total time required to manage the heap is $O(n \lg n)$.

Problem 3: Minimum Cost Sum

You are given a sequence $a_1, a_2, \ldots, a_n$ of nonnegative integers, where $n \geq 1$. You are allowed to take any two numbers and add them to produce their sum. However, each such addition has a cost which is equal to the sum. The goal is to the find the sum of all the numbers in the sequence with minimum total cost. Describe an algorithm for finding the sum of the numbers in the sequence with minimum total cost. Argue the correctness of your algorithm.

Problem 4: Speech recognition

We can use dynamic programming on a directed graph $G = (V, E)$ for speech recognition. Each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. The labeled graph is a formal model of a person speaking a restricted language. Each path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds produced by the model. The label of a directed path is defined to be the concatenation of the labels of the edges on that path.

Describe an efficient algorithm that, given an edge-labeled graph G with distinguished vertex v_0 and a sequence $s = (\sigma_1, \cdots, \sigma_k)$ of characters from Σ, returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of the algorithm. Clearly write any dynamic programming formulation you may use to solve this problem.

Solution: Speech recognition

Let's say that a directed graph G with n vertices $(v_1, v_2, \ldots v_n)$ is encoded as a matrix M in the following manner: Let M be an n x n matrix such that (for $1 \leq$ both $i, j \leq n$):

$$M(i,j) = \begin{cases} \sigma(i,j), \text{ the label on edge } (i,j), & \text{if there is an edge } (i,j) \text{ from } v_i \text{ to } v_j \\ \\ \epsilon \text{ (the empty string)}, & \text{if there is no edge from } v_i \text{ to } v_j \end{cases}$$

For convenience, let's call the distinguished vertex v_1 instead of v_0. Remember that we want our algorithm to return a path (i.e., a sequence of vertices) in G that begins at v_1 and has $s = (\sigma_1, \cdots, \sigma_k)$ as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Note that s is a sequence of length k; we will consider the "s-suffix" of length m (i.e., $(\sigma_{k-m+1}, \cdots, \sigma_k)$), where m will vary from 0 to k. The suffix of length 0 is the empty string, ϵ.

Let's define the n x $(k+1)$ boolean matrix P as follows $(1 \le i \le n, 0 \le m \le k)$:

$$P(i,m) = \begin{cases} \text{TRUE}, & \text{if } \exists \text{ a path in } G \text{ starting at vertex } v_i \text{ such that the} \\ & \text{sequence of labels along the path is the } s\text{-suffix} \\ & \text{of length } m \text{ (i.e., } (\sigma_{k-m+1}, \cdots, \sigma_k)\text{). We will then} \\ & \text{say that } v_i \text{ "produces" this suffix.} \\ \text{FALSE}, & \text{otherwise} \end{cases}$$

Each row i of P indicates the s-suffix lengths that are produced by vertex v_i. Each column m of P indicates the vertices that produce the s-suffix of length m.

If there exists a path that begins at v_1 and has s as its label, we will need to output the actual path (i.e., the sequence of vertices that allow s to be produced). We will store information in an n x k integer matrix, Q $(1 \le i \le n, 1 \le m \le k)$. Q's entries are all initialized to 0 (i.e., Q is initialized as a "zero matrix"). Q will serve as a "roadmap" matrix; this will be made clear later.

Our dynamic programming algorithm will construct the P and Q matrices for our graph column-by-column. At the beginning of the algorithm, the P matrix is empty. However, we can immediately fill in P's first column (column 0) with "TRUE"s, since we can say that every vertex in the graph produces the empty string, ϵ. In other words, $P(i,0) = \text{TRUE}$ for all i $(1 \le i \le n)$.

P's second column (column 1) is also easy to fill out. We assign "TRUE" to vertices that produce the s-suffix of length 1 (just (σ_k)) and "FALSE" to the other vertices. In other words, $P(i,1) = \text{TRUE}$ if and only if there is an edge from vertex v_i to some vertex v_j with edge label σ_k. Instead of doing this as part of the initialization process, however, we can do this during the iteration process (since the filling of P's column 0 can serve as our initialization step). In other words, we don't have to write separate code to fill out column 1 in particular.

What about the Q matrix? Recall that $P(i,1) = \text{TRUE}$ if and only if there is an edge from vertex v_i to some vertex v_j with edge label σ_k. If such an edge exists, we will let $Q(i,1)$ equal one such j (in our algorithm, it will equal the lowest such j).

Let's go back to the P matrix. Given the TRUE/FALSE values in column m (where m is between 0 and $k-1$), we can fill in column $m+1$. $P(i,m+1) = \text{TRUE}$ if and only if vertex v_i produces the s-suffix of length $m+1$ $(\sigma_{k-m}, \cdots, \sigma_k)$, which happens if and only if there is an edge with label σ_{k-m} going from v_i to a vertex v_j that produces the s-suffix of length m $(\sigma_{k-m+1}, \cdots, \sigma_k)$. In other words, $P(i,m+1) = \text{TRUE}$ if and only if $M(i,j) = \sigma_{k-m}$ and $P(j,m) = \text{TRUE}$ for some j. If $P(i,m+1) = \text{TRUE}$, $Q(i,m+1)$ will then be assigned one such j (in our algorithm, it will equal the lowest such j).

So, we can fill in all the columns of P and Q. If $P(1,k) = \text{FALSE}$, then v_1 does not produce the desired sequence s, and we return NO-SUCH-PATH. If $P(1,k) = \text{TRUE}$, then v_1 does produce s, and we can find the appropriate path (sequence of vertices) by working from right to left in the "roadmap" matrix Q. Our desired path begins with vertex v_1. To find the next vertex in the path, read off the $(1,k)$ entry of Q in the upper-right-hand corner of the matrix and call it w_1. The next vertex in the path is v_{w_1}. To find the next vertex, read off the $(w_1, k-1)$ entry of Q in the column to the left and call it w_2. The next vertex in the path is v_{w_2}. Continue reading off vertex indices w_z in this fashion from right to left. The desired path is then described by this sequence of $k+1$ vertices: $(v_1, v_{w_1}, v_{w_2}, \cdots, v_{w_k})$.

Note 1: We will actually get more than we want. Since we only care about paths that begin at v_1, we could eliminate the last column of the P matrix and the last column of the Q matrix and take care of the "full-sequence" case separately. (The $(1,k)$ entries of P and Q were the only relevant entries in the last column of both matrices.)

Note 2: You don't need the P matrix if you have the Q matrix. Just remember that the nonzero entries in Q correspond to "TRUE"s and the zero entries correspond to "FALSE"s, and that P's column 0 was really unnecessary. $Q(i,m) \ne 0$ if and only if $P(i,m) = \text{TRUE}$ $(1 \le i \le n,\ 1 \le m \le k)$. If you just use the Q matrix, you will save storage space, but your algorithm may be a little bit harder to understand!

Here's the algorithm (finally!):

Inputs: s, the desired sequence of characters $(\sigma_1, \cdots, \sigma_k)$ from Σ; we will assume that s is an array of k strings indexed from 1 to k. k, the length of s. M, the matrix representation of the graph G. n, the number of vertices in G.

Outputs: $Path$, an array of vertex indices describing a path in G that begins at v_1 and has s as its label, if such a path exists. $Path[0]$ will equal 1 if such a path exists; entries $1..k$ will contain the other vertex indices in the proper order. If no such path exists, NO-SUCH-PATH is returned. **procedure** VITERBI

```
$(s, k, M, n)$;
var
  $P$ : array[1..$n$, 0..$k$] of boolean;
  $Q$ : array[1..$n$, 1..$k$] of integer;
  $Path$ : array[0..k] of integer;
begin
  $P[, 0]$ := TRUE
      # Initialize column 0 of $P$ by filling it with “TRUE”s.
  $Q$ := $n$ x $k$ matrix of zeros
      # Initialize $Q$.
  for $m = 0$ to $k - 1$ do
      # Suffix-length loop.
    for $i = 1$ to $n$ do
        # Loop to label vertices one-by-one.
      begin
        $P[i, m + 1]$ := FALSE
            # “FALSE” will be the final value of $P[i, m + 1]$ if the
            # $j$ loop is completed “without success” (i.e., without an
            # updating to “TRUE”).
        for $j = 1$ to $n$ do
            # Loop to find “next” vertex in potential path.
          if $M[i, j] = s[k - m]$ and $P[j, m]$ = TRUE then
            begin
              $P[i, m + 1]$ := TRUE
              $Q[i, m + 1] := j$
              break out of $j$ loop
                  # We just need one such $j$!
            end
      end
  if $P[1, k]$ = FALSE then
    return NO-SUCH-PATH
  else
    begin
      $Path[0] := 1$
          # $v_1$ must be the starting vertex.
      for $z = 1$ to $k$ do
        $Path[z] := Q[Path[z - 1], k - z + 1]$
            # Observe that as $z$ increases from 1 to $k$, $k - z + 1$
            # goes from $k$ down to 1; we are tracing through the
            # $Q$ matrix from right to left. Each $Q$ entry we include
            # in our $Path$ tells us the next row (vertex) we should
            # “look at.”
      return $Path$
    end
end;
```

This algorithm is $O(n^2k)$. Here's a nice way of seeing this: There are nk subproblems (consider the TRUE/FALSE labeling for each vertex for each nonzero suffix length); each subproblem corresponds to an entry position (i, j) in the P and Q matrices (ignore the trivial column 0 in P). It takes $O(n)$ time to do the work for each of the nk subproblems, so the algorithm is $O(n^2k)$. In other words, for each of the k nonzero suffix lengths, we label each of the n vertices TRUE or FALSE (and update Q accordingly), and each labeling takes $O(n)$ time. The reconstruction of the desired path (if one exists) from the Q matrix is a paltry $O(k)$ operation.