CS 317, Assignment 5

Answers

The assignment is all about dynamic programming, and so your algorithms must be dynamic programming algorithms unless otherwise stated.

1. This question refers to the Matrix Chain Multiplication as discussed in class.

   (a) Refine the algorithms MatrixChainMult from the lecture notes such that in addition to the table m_{ij} (which stores the cost of the optimal bracketing) you also fill in a data structure s that stores the actual optimal bracketing. You decide what s is, within the constraints that (a) it has to be usable for the process of performing the actual multiplication, and (b) your refined MatrixChainMult must maintain the original $O(n^3)$ running time.
   Explain how the structure s can be used afterward. Provide an argument for the correctness of your algorithm and explain how the original running time is maintained.

   Answer:

   Recall the recursive solution of the problem:

   $$m_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}(m_{i,k} + m_{k+1,j} + r_{i-1} \times r_k \times r_j) & \text{if } i < j \end{cases}$$

   According to this solution, we have no bracketing when $i = j$, since there is nothing to bracket. In the recursive step, each k realizes a bracketing from i to k and from $k+1$ to j, with the cost of multiplication given by $m_{i,k} + m_{k+1,j} + r_{i-1} \times r_k \times r_j$. Obviously, the right bracketing is given by the value k that realizes the minimum $m_{i,j}$. We then simply store that k in a new table $s_{1 \le i \le n, 1 \le j \le n}$.
   We end up with the algorithm shown in Figure 1. The innermost for loop implements the typical minimum-finding process for $m_{i,j}$, also remembering the value k that realizes this minimum. At the end of that loop $s_{i,j}$ is assigned accordingly.
   Both correctness and running time follow immediately from the correctness and running time of the original algorithm since we do not alter that algorithm in any way other than performing an additional assignment to $s_{i,j}$ which gets to store the right bracketing as argued above.

   (b) Give a recursive algorithm MatrixChainMultiply(M, s, i, j) that actually performs the optimal matrix-chain multiplication $M_i \times \cdots \times M_j$. The inputs are the chain $M =$

```
algorithm MatrixChainMultOpt(r_{1...n}):
    for i = 1 to n do
        m_{i,i} ← 0
        s_{i,j} ← 0
    for r = 1 to n − 1 do
        for i = 1 to n − r do
            j ← i + r
            s_m ← ∞
            m_m ← ∞
            for k = i to j − 1 do
                m ← m_{i,k} + m_{k+1,j} + r_{i−1} × r_k × r_j
                if m < m_m then
                    m_m ← m
                    s_m ← k
            m_{i,j} ← m_m
            s_{i,j} ← s_m
```

Figure 1: Modified matrix chain multiplication algorithm.

$\langle M_1, M_2, \ldots, M_n \rangle$ of matrices to be multiplied, the structure s returned by the algorithm that you developed for Question 1a, and the two indices i and j, $1 \leq i \leq j \leq n$. The call MatrixChainMultiply$(M, s, 1, n)$ will thus optimally multiply the whole chain of matrices. Assume that MatrixMultiply(A, B) returns the product of the two matrices A and B.

Answer:

We already have the optimal bracketing as produced by the algorithm above, so we can proceed with a simple divide and conquer algorithm that multiply all the matrices following the table s:

```
algorithm MatrixChainMult(M, s, i, j):
    if i = j then
        return M_i
    else
        return MatrixMultiply ( MatrixChainMult(M, s, i, s_{i,j}),
                                MatrixChainMult(M, s, s_{i,j} + 1, j) )
```

---

2. You are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinguished vertices s and t. The weight of a path is the sum of the weights of the edges in the path. Describe a dynamic-programming approach for finding the path from s to t with the largest weight. Prove the correctness of your algorithm and analyze its running time.

   Hint: Pay attention to the order of evaluation in the memoization structure, as this not as straightforward as in other dynamic programming algorithms.

ANSWER:

In the dynamic programming examples and exercises you have seen so far the hard part was to come up with a recursive solution. The memoization structure and the ordering of sub-problems were then obvious. This exercise is for a change a good example of non-obvious evaluation order (I think that the memoization structure is always obvious, or at least I cannot think of any problem for which this is not the case).

Note that the graph is acyclic and so the longest path from u to v cannot contain u. Otherwise the concept of longest path does not make any sense. With this in mind, consider the longest path from some vertex u to t ($u \overset{t}{\rightsquigarrow} p$) and let v be the next vertex after u on this path that is, $p = u \frown v \overset{t}{\rightsquigarrow} p'$. It is easy to see that if p is the longest path from u to t then p' is the longest path from v to t. Indeed, suppose that the longest path from v to t is p'' instead. If so, $u \frown v \overset{t}{\rightsquigarrow} p''$ is longer than $p = u \frown v \overset{t}{\rightsquigarrow} p'$, a contradiction ($p$ is the longest path from u to t).

With this property in mind the following recursive solution for the length d_u of the shortest path from u to t becomes obvious:

$$d_u = \begin{cases} 0 & \text{if } u = t \\ \max_{(u,v)\in E}\{w_{u,v} + d_v\} & \text{otherwise} \end{cases}$$

We obtain a dynamic programming version of the recursive solution using the memoization structure of an array $d_{i\in V}$. To find the actual vertices along the longest paths we can also maintain an extra array $v_{i\in V}$.

Next we need to establish an order of sub-problems that is, the order in which d (and v) are computed. To do this, we notice that we can start with the base case of the recursive solution that the longest path from t to t has cost 0. From here we can determine in a bottom-up fashion the cost of longest paths from all the vertices adjacent to t using the recursive case. Once have these we can proceed to the vertices adjacent to those vertices we have determined the cost for in the previous step. Therefore we can establish a suitable order over vertices in V by topologically sorting V starting with $t = 1$ and with respect to the graph $G^{\mathbb{T}}$ (recall from Assignment 2 that the transpose $G^{\mathbb{T}}$ of a directed graph $G = (V, E)$ reverses all its edges that is, $G^{\mathbb{T}} = (V, \{(v, u) \in V \times V : (u, v) \in E\})$). An immediate inductive argument establishes that under such an ordering we can use the recursive solution to fill in d (and v) from left to right.

In all, we end up with the algorithm shown in Figure 2. The discussion above establishes the correctness of this algorithm. Note that the algorithm computes the longest path from all the vertices in V to t. The reconstruction of longest path from s to t is left as an exercise to the reader (hint: it goes the same as for the output of Floyd's algorithm).

The running time of the algorithm is $O(|V| + |E|)$. Indeed, this is the complexity of the topological sort (as discussed in class) as well as the number of vertices v visited inside the foreach loop according to an argument similar to the one for the running time of the graph transposition algorithm.

```
algorithm LongestPath(G = (V, E), t = 1):
    Compute G^T = (V, {(v, u) ∈ V × V : (u, v) ∈ E})
    Let V' be the sequence of exactly all vertices in V
    topologically sorted over G^T starting from t
    d_t ← 0
    v_t ← t
    for i = 2 to |V| do
        d_{V'_i} ← −∞
    for i = 2 to |V| do
        u ← V'_i
        foreach v : (u, v) ∈ E do
            if d_u + w_{u,v} ≥ d_v then
                d_v ← d_u + w_{u,v}
                v_v ← v
```

Figure 2: Longest path algorithm.

3. A palindrome is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes include all strings of length 1, `civic`, `racecar`, and `aibohphobia` (which is the officially unofficial name for the fear of palindromes). Design an efficient algorithm for finding the longest palindrome that is a substring of a given input string. For this problem a substring of the string s is obtained by deleting an arbitrary number of characters from s; the deleted characters need not be adjacent to each other. For example, given the input `character` your algorithm should return `carac`. Prove the correctness of your algorithm and analyze its running time.

Answer:

Let $w = \langle w_1, w_2, \ldots, w_n \rangle$ be the input sequence. Let for convenience $w_{i,j} = \langle w_i, \ldots, w_j \rangle$, let $p = \langle p_1, p_2, \ldots, p_m \rangle$ be the longest palindrome subsequence of w, and let $p_{i,j} = \langle p_i, \ldots, p_j \rangle$. We have:

(a) If $n = 1$ then obviously $m = 1$ and $p_1 = w_1$.

(b) If $n = 2$ and $w_1 = w_2$ then equally obviously $m = 2$ and $p_1 = p_2 = w_1 = w_2$.

(c) If $n = 2$ but $w_1 \neq w_2$ then $m = 1$ and $p_1 = w_1$ or $p_1 = w_2$.

(d) If $n > 2$ and $x_1 = x_n$ then $m > 2$, $p_1 = x_1$, $p_m = x_n$ and $p_{2,m-1}$ is a largest palindrome subsequence of $w_{2,n-1}$.

Indeed, we can choose the ends of w as the ends of p, and we can always choose one additional character in the center of p, so that $m > 2$. Now if $p_1 \neq x_1$ then $\langle x_1 \rangle + p + \langle x_n \rangle$ is a palindrome subsequence lagrer than p, a contradiction with the fact that p is the largest such a sequence. The same goes for $p_m \neq w_m$. Suppose now that there exists a palindrome subsequence p' of $w_{2,n-1}$ longer than $p_{2,m-1}$. Then $\langle x_1 \rangle + p' + \langle x_n \rangle$ is a palindrome of w longer than p, a contradiction. It follows that $p_{2,m-1}$ is a longest palindrome sequence of $w_{2,n-1}$.

```
algorithm LongestPalindrome(w = ⟨w_1, w_2, ..., w_n⟩):
    for i = 1 to n do
        c_{i,i} ← 1
        j ← i + 1 if w_i = w_j then
            c_{i,j} ← 2
            b_{i,j} ← " ↙ "
        else
            c_{i,j} ← 1
            b_{i,j} ← " ↓ "
    c_{n,n} ← 1
    for i = n − 2 down to 1 do
        for j = i + 2 to n do
            if w_i = w_j then
                c_{i,j} ← c_{i+1,j+1} + 2
                b_{i,j} ← " ↙ "
            KwDown else if c_{i,j−1} ≤ c_{i+1,j} then
                c_{i,j} ← c_{i+1,j}
                b_{i,j} ← " ↓ "
            else
                c_{i,j} ← c_{i,j−1}
                b_{i,j} ← " ← "
```

Figure 3: Longest palindrome algorithm.

(e) If $n > 2$ and $x_1 \neq x_n$ then $p_1 \neq w_1$ implies p is a longest palindrome of $w_{2,n}$.
If $p_1 \neq w_1$ then p must be a subsequence of $w_{2,n}$. If a longer such a palindrome subsequence p' exists then p' is also a palindrome subsequence of w and is longer then p, a contradiction.

(f) If $n > 2$ and $x_1 \neq x_n$ then $p_m \neq w_n$ implies p is a longest palindrome of $w_{1,n-1}$.
The proof is almost identical to the previous property.

The properties above establish the following recursive solution for the cost $c_{i,j}$ of the longest palindrome sequence of $w_{i,j}$.

$$c_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 2 & \text{if } j = i + 1 \text{ and } x_i = x_j \\ 1 & \text{if } j = i + 1 \text{ and } x_i \neq x_j \\ c_{i+1,j+1} + 2 & \text{if } j > i + 1 \text{ and } x_i = x_j \\ \max\{c_{i,j-1}, c_{i+1,j}\} & \text{if } j > i + 1 \text{ and } x_i \neq x_j \end{cases}$$

With a memoization matrix for $c_{i,j}$ we end up with the dynamic programming solution shown in Figure 3. The matrix $c_{i,k}$ will be filled in in decreasing order for i and increasing order for j (since i is the beginning of the subsequence and j the end of it).

The actual largest palindrome subsequence can be obtained by tracing the arrows from the matrix b.

Make sure you review the submission guidelines posted on the course's Web site before submitting.