COMP 360 Algorithm Design Techniques

Assignment 1 Solutions

Please report typos and suspected errors to sue@cs.mcgill.ca.

1. *Counting Paths on a Grid.*

   (a) Let the grid rows be labelled $0, 1, 2, \ldots, n$ and the columns $0, 1, 2, \ldots, m$. The following dynamic programming algorithm will find the number of distinct paths, denoted $N_{i,j}$, from the origin to each point (i, j) on the grid, where the grid has "short-cut" diagonal edges as well as horizontal and vertical edges. In what follows, the word "arc" refers to a directed grid edge.

   Path-Counting
   1. $N_{0,0} = 1$
   2. for $i = 1$ to n
   3. $\quad N_{i,0} = N_{i-1,0}$
   4. for $j = 1$ to m
   5. $\quad N_{0,j} = N_{0,j-1}$
   6. for $i = 1$ to n
   7. $\quad$ for $j = 1$ to m
   8. $\quad\quad N_{i,j} = N_{i,j-1} + N_{i-1,j} + N_{i-1,j-1}$

   (b) (Here we use the notation $a^{\downarrow}_{i,j}$ to refer to the arc from $(i-1, j)$ to (i, j), and $a^{\rightarrow}_{i,j}$ to refer to the arc from $(i, j-1)$ to (i, j), and $a^{\searrow}_{i,j}$ to refer to the arc from $(i-1, j-1)$ to (i, j).

   There is only the trivial path from the origin to itself. In addition, the only way to reach $(i, 0)$ is using the arc $a^{\downarrow}_{i,0}$. Thus the number of distinct paths to $(i, 0)$ is exactly the same as the number to $(i-1, 0)$. Similar arguments apply with regard to $(0, j)$. Thus we obtain lines $1, 3$ and 5.

   For the main recurrence in line 8, consider a point (i, j), with $i, j \geq 1$. There are only three ways a path can reach (i, j): using the arc $a^{\downarrow}_{i,j}$, or the arc $a^{\rightarrow}_{i,j}$, or the arc $a^{\searrow}_{i,j}$. Also note, for example, that any path P_1 whose final arc is $a^{\downarrow}_{i,j}$ is definitely not the same as a path P_2 whose final arc is $a^{\rightarrow}_{i,j}$. (We don't want to count the same path more than once.) Two paths P_1 and P_2 whose final arcs are both $a^{\downarrow}_{i,j}$ are distinct if and only if $P_1' = P_1 - \{a^{\downarrow}_{i,j}\}$ and $P_2' = P_2 - \{a^{\downarrow}_{i,j}\}$ are distinct. Note though that P_1' and P_2' are distinct paths to $(i-1, j)$. A similar argument applies to two paths P_1 and P_2 whose final arcs are both $a^{\rightarrow}_{i,j}$; similarly for final arcs both $a^{\searrow}_{i,j}$. Hence, each path to (i, j) corresponds to a path to either $(i-1, j)$ or $(i, j-1)$ or $(i-1, j-1)$. Thus $N_{i,j} \le N_{i,j-1} + N_{i-1,j} + N_{i-1,j-1}$. Conversely, any path to $(i-1, j)$ or $(i, j-1)$ or $(i-1, j-1)$ leads to a path to (i, j). Thus $N_{i,j} \ge N_{i,j-1} + N_{i-1,j} + N_{i-1,j-1}$. The recurrence in line 8 follows.

(c) Applying the algorithm you should obtain the solution matrix

$$N = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 3 & 5 \\ 1 & 5 & 13 \\ 1 & 7 & 25 \end{pmatrix}$$

Thus, the number of distinct paths to $(3, 2)$ is $N_{3,2} = 25$.

2. *Longest Common Subsequence Problem.*

(a) Let row i, $1 \le i \le n$, be labelled by x_i, the ith element of X. Similarly, let column j, $1 \le j \le m$, be labelled by y_j, the jth element of Y. Row 0 and column 0 are given blank labels. Each vertical and horizontal arc is given length 0. A diagonal arc $a^{\searrow}_{i,j}$ (the arc from $(i-1, j-1)$ to (i, j)) is given cost -1 if $x_i = y_j$. All other diagonal arcs are given cost ∞ (or, alternatively, such an arc is omitted from the graph).

Given this schema, consider a path P from the origin to (n, m). Such a path corresponds to a common subsequence Z of X and Y by applying the following rules. Traverse the arcs in order along P. An arc $a^{\downarrow}_{i,j}$ in P corresponds to x_i **not** being chosen in Z; similarly, an arc $a^{\searrow}_{i,j}$ in P corresponds to y_j **not** being chosen in Z. For an arc $a^{\searrow}_{i,j}$ in P, if $x_i = y_j$ then we place $x_i = y_j$ in Z. If $x_i \ne y_j$ then neither x_i nor y_j is placed in Z. (Note that for a shortest path P, there will be no diagonal arcs $a^{\searrow}_{i,j}$ with $x_i \ne y_j$ since these arcs have infinite cost. Thus, this final rule is irrelevant for shortest paths).

(b) Applying the algorithm you should obtain the following solution matrix. The diagonal arcs of length -1 are shown. In addition a shortest path of length -6 is given in bold; this path corresponds to a maximum common subsequence $Z =$ `CTAGAT` of length 6. (Another possible solution is $Z =$ `CTAGAC`.)

$$
\left(\begin{array}{c|ccccccccc}
 & . & C & G & T & A & G & A & T & C \\
\hline
. & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
A & \mathbf{0} & 0 & 0 & 0 & \searrow -1 & -1 & \searrow -1 & -1 & -1 \\
C & 0 & \searrow \mathbf{-1} & \mathbf{-1} & -1 & -1 & -1 & -1 & -1 & \searrow -2 \\
T & 0 & -1 & -1 & \searrow \mathbf{-2} & -2 & -2 & -2 & \searrow -2 & -2 \\
A & 0 & -1 & -1 & -2 & \searrow \mathbf{-3} & -3 & \searrow -3 & -3 & -3 \\
G & 0 & -1 & \searrow -2 & -2 & -3 & \searrow \mathbf{-4} & -4 & -4 & -4 \\
G & 0 & -1 & \searrow -2 & -2 & -3 & \searrow \mathbf{-4} & -4 & -4 & -4 \\
A & 0 & -1 & -2 & -2 & \searrow -3 & -4 & \searrow \mathbf{-5} & -5 & -5 \\
C & 0 & \searrow -1 & -2 & -2 & -3 & -4 & \mathbf{-5} & -5 & \searrow -6 \\
C & 0 & \searrow -1 & -2 & -2 & -3 & -4 & \mathbf{-5} & -5 & \searrow -6 \\
T & 0 & -1 & -2 & \searrow -3 & -3 & -4 & -5 & \searrow \mathbf{-6} & \mathbf{-6}
\end{array}\right)
$$

3. *Planning a Company Party.*

(a) Let $c_i \geq 0$ be the conviviality rating of employee i. We wish to find a subset S of employees maximising $\sum_{j \in S} c_j$ with the property that if $i \in S$ then $parent(i) \notin S$ (we use this terminology because the supervisor of i is the parent of i in the company tree T).

Let T_i be the subtree of T rooted at employee i. We denote by C_i the maximum conviviality rating of a party, restricted to employees of T_i, to which employee i is invited. We denote by $\bar{C}_i$ the maximum conviviality rating of a party, restricted to employees of T_i, to which employee i is not invited. Observe that, if the president of the company is denoted by the root r, $\max(C_r, \bar{C}_r)$ is the maximum conviviality rating of any party. We now define our recursive procedure.

$$
\begin{aligned}
C_i &= c_i + \sum_{j:i=parent(j)} \bar{C}_j \\
\bar{C}_j &= \sum_{j:i=parent(j)} \max(C_j, \bar{C}_j)
\end{aligned}
$$

The reasoning behind these equations is as follows. The first equation states that the optimal way to select employees from T_i, if i must be included, is to select employees from the subtrees rooted at each child j of i such that j is *not* invited. The second equation states that the optimal way to select employees T_i, if i must not be included, is to optimally select employees from the subtrees rooted at each child j of i where there is no restriction upon whether or not j is invited.

This leads to the following dynamic program: for any leaf node i in T, let $C_i = c_i$ and $\bar{C}_i = 0$. Then working up the tree we evaluate C_i and $\bar{C}_i$ according to the recursive definition above.

Once the C_i and $\bar{C}_i$ are calculated we recover an optimal invitation list by running INVITE(r).

```
INVITE(i)
if C_i > C̄_i
    then
        invite employee i to party
        for all grandchildren j of i
            INVITE(j)
    else
        for all children j of i
            INVITE(j)
```

Note that if $C_r > \bar{C}_r$ then an optimal solution must involve inviting the company president r. If this happens then none of the children (supervisees) of the president may be invited, and so we attempt to invite optimally the grandchildren of the president in the company tree. If $C_r \leq \bar{C}_r$ then in some optimal solution the president is not invited. In this case we are free to consider inviting the children of the president. Building an invitation list in this manner recovers a solution of value $max(C_r, \bar{C}_r)$, as desired.

(b) See solution to part a).

(c) At node i in the tree, determining C_i requires looking up the answers to $2g_i$ sub-problems, where g_i is the number of grandchildren of i. (For each grandchild of i, we have to decide whether it is better to take that person or not.) Determining $\bar{C}_i$ requires looking up $2n_i$ sub-problems, where n_i is the number of children of i: this is because, for each child of i, we have to consider the possibility of inviting that child, or not inviting that child. Since we have a tree, $\sum_{i \in T} n_i = n-1$, where n is the number of employees. Also, $\sum_{i \in T} g_i = n - 1 - n_r$, since each employee except the root and the children of the root is the grandchild of exactly one other employee. It follows that the dynamic program runs in linear, $O(n)$, time, since doing the look-ups of two subproblems and the comparison between them takes constant time each time this is done.

4. *Longest Increasing Subsequence.*

(a) Now $f_1 = 1, f_2 = 2, f_3 = 3, f_4 = 2, f_5 = 4, f_6 = 3, f_7 = 4, f_8 = 5, f_9 = 6, f_{10} = 5$. Thus, the longest increasing subsequence ends with $x_9 = 9$ and has length 6. It follows that $(1, 2, 3, 5, 7, 9)$ is a longest increasing subsequence.

(b) Set up a recursive definition of f_i and prove its validity.

$$f_i = \max_{j<i \text{ AND } \mathrm{x_j < x_i}} f_j + 1$$

The longest subsequence Z_i ending in x_i contains only elements from $(x_1, x_2, \ldots, x_i)$. Now since x_i is the final element in the subsequence, it is also the largest element in the subsequence. Thus, if $x_j \geq x_i$ for some $j < i$ then x_j is not in Z_i. It follows that $Z_i = Z_j \cup \{x_i\}$ where Z_j is a increasing subsequence ending with some $x_j < x_i$ and $j < i$. Thus $f_i = |Z_j| + 1 \leq f_j + 1 \leq \max(f_j + 1 \ : \ j < i \text{ AND } \mathrm{x_j} < \mathrm{x_i})$. Take any $j < i$ with $x_j < x_i$. Let Z_j be a longest increasing subsequence of $(x_1, x_2, \ldots, x_j)$. Then $Z_j \cup \{x_i\}$ is an increasing subsequence of $(x_1, x_2, \ldots, x_i)$ of length $f_j + 1$. Thus $f_i \geq f_j + 1$, and as this holds for any $j < i$ with $x_j < x_i$ we obtain that $f_i \geq \max(f_j + 1 \ : \ j < i \text{ AND } \mathrm{x_j} < \mathrm{x_i})$. The recurrence follows.

(c) We will use the following dynamic program. Note that line 4 creates a pointer for each i stating the subproblem to whose solution x_i should be added in order to create a longest subsequence ending in x_i. These pointers allow us to recover the optimal solution. INCREASING-SUBSEQUENCE

1. $f_0 = 0$
2. for $i = 1$ to n
3. $\quad f_i = \max\ (f_j + 1 \ : \ j < i \text{ AND } \mathrm{x_j} < \mathrm{x_i})$
4. $\quad p_i = \text{argmax}\ (f_j + 1 \ : \ j < i \text{ AND } \mathrm{x_j} < \mathrm{x_i})$
5. $f_{\text{MAX}} = \max_i f_i$

i	x_i	f_i	p_i
0	$-\infty$	0	.
1	5	1	0
2	8	2	1
3	3	1	0
4	9	3	2
5	4	2	3
6	8	3	5
7	1	1	0
8	2	2	7
9	9	4	6
10	6	3	5

Therefore the optimal value is 4 since $f_{\text{MAX}} = f_9 = 4$. The longest increasing subsequence therefore ends in $x_9 = 9$. Reading backwards using the pointers we see that a longest increasing subsequence is $(3, 4, 8, 9)$.

5. *Ruler Folding.*

(a) Build a table with columns numbered $0, 1, \ldots, 2M$, where $M = max\{l_i\}, 1 \leq i \leq n$, and rows numbered $0, 1, \ldots, n$. Each row i corresponds to the initial portion of the ruler, $A_0, A_1, \ldots, A_i$, with the segments beyond joint A_i removed.

Each column corresponds to a position with integer x-coordinate in the interval $[0, 2M]$.

Note: Imagine an optimum folding of the ruler into the interval $[0, 2M]$. This optimum folding may only occupy some small portion of this interval. However, it can be slid to the left until some joint hits the left end. In other words, there is always an optimum folding such that some joint has x-coordinate $= 0$.

As in class, put a check in row i, column j if and only if the initial portion of the ruler ending with A_i (with the remaining links removed) can be placed so that joint A_i lands at x-coordinate j in $[0, 2M]$. We can initialize the table so that row 0 has checks in all its entries, because the point A_0, with the rest of the ruler removed, can be placed at any position in the interval.

In addition, add the following information at each table entry i, j that contains a check: consider all the ways that subchain $A_0, \ldots, A_i$ can be placed in $[0, 2M]$ so that A_i lands at $x = j$, and for each way, note the maximum of the x-coordinates of its joints. (Some of these joints may be at x-coordinates that are greater then j, but not greater than $2M$. Record the *smallest* of these maximum values along with the check in entry (i, j). In other words, considering all foldings that place A_i at j, record the *minimum* possible maximum travel to the right achieved by any of these foldings. If there is a check in table entry i, j, let's call the extra information we store there $bestx(i, j)$.

<u>Fill order:</u> As before, fill from the top row to the bottom row, left to right in each row. A check in row $i - 1$, col. j generates 0, 1, or 2 checks in the next row, depending on whether $j - l_i \geq 0$ (we try to fold link $[A_{i-1}, A_i]$, which has length l_i, to the left of A_{i-1}, which is located at $x = j$), or $j + l_i \leq 2M$ (we try to fold link $[A_{i-1}, A_i]$ to the right of A_{i-1}). If a check already appears in the position, then the information $bestx(i, j)$ must be updated if a folding with less extreme travel to the right has just been determined. We want $bestx(i, j)$ to be as small as possible.

A check *must* appear in the last row, since we know that any ruler can be folded into $[0, 2M]$. The optimum folding length is the minimum of $bestx(n, j)$, for $0 \leq j \leq 2M$. This is because the rightmost travel of a folding gives the length of that folding when the folding has some joint placed at $x = 0$; otherwise, it gives a bigger number than its true folded length. We know from above that some optimum folding does have such a joint at $x = 0$.

<u>Backtracking to find an optimum folding:</u> To determine the actual folding, not just its length, we record, at each check, even more than $bestx(i, j)$; namely, we record a pointer to the entry in the previous row $i - 1$ that generated the check

at position i, j *and* that resulted in the value of $bestx(i, j)$. This entry is either the one at $i-1, j-l_i$ or the one at $i-1, j-l_i$ – whichever one gives the smaller value for $bestx(i, j)$. (Exercise: How would you compute the value of $bestx(i, j)$?) To determine the actual folding, we follow the pointers back from the position in row n with the minimum $bestx$ (this is not the position of A_n in general, as usually some previous joint is the one that is positioned farthest to the right.)

(b) Running Time: The time is in $O(nM)$. However M may be arbitrarily huge compared to n. The number of bits we use to write down the number M is only lgM. So $M = 2^{lgM}$, and our algorithm is in $O(n2^{lgM}lgM)$ (it may take time in $O(lgM)$ to do a single arithmetic operation involving numbers with lgM bits.) In fact, in terms of the number $|X|$ of bits needed to supply the lengths of all the links, lgM (the number of bits it takes to write down the length of the longest link) may be nearly as big as $|X|$. So our algorithm, in terms of the length $|X|$ of the problem description, is in $O(|X|^2 2^{|X|})$. To see this, note that it takes time to fill out a table entry. If M is huge, the numbers may be huge, so the time to do an arithmetic operation such as adding two large integers together (even though only a constant number of arithmetic operations are needed to compute each entry) may be $O(|X|)$ also. The table itself has $O(|X|2^{|X|})$ entries.

If the numbers are reasonable, and M is bounded by a constant so that each link length l_i can be recorded in one computer storage place, then arithmetic operations can be done in one "step" as usual, and the number of columns $2M + 1$ is bounded by a constant, and the running time of the algorithm is in $O(n) = O(|X|)$, which is a *linear* function of the amount of input data.

The assumption that the sizes of the lengths don't get too big is essential to get the linear running time. (In the problem about planning a company party, what would happen to the running time analysis if there were some c_i's that were absurdly large and didn't fit into one unit of memory?)