Student: Yu Cheng (Jade)

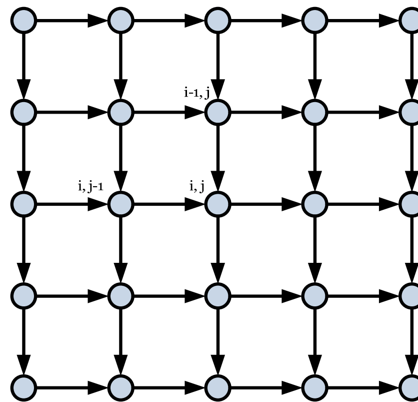
ICS 675

Assignment #2

Nov 06, 2009

Exercise 1

Question: Using dynamic programming, propose an algorithm that calculates the total number of possible paths in an LCS (Longest Common Subsequence) grid, similar to the one shown below. Your algorithm finds only the total number of paths, without listing them



The size of the grid is $(m \times n)$ where $m > 0, n > 0$. Note that at vertex (i, j) of the example grid, 2 paths are possible. Either coming east from $(i, j-1)$ or coming south from $(i-1, j)$.

a. Propose the algorithm.

Answer: First construct the simple recursive algorithm. It is expressed in the following pseudocode.

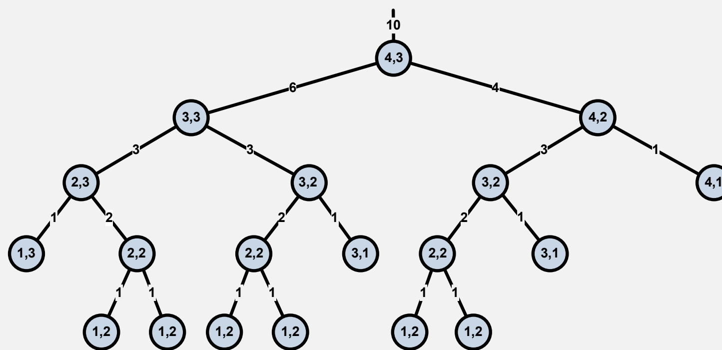
```
/**
 * Simple Recursive Algorithm
 *
 * Function takes the row count and column count of a node in a LCS grid graph, and
 * returns the number of possible paths that we can take to get to this node starting
 * from the top left node (1, 1).
 *
 * @param x The row count of the current node.
 * @param y The column count of the current node.
 *
 * @return The number of possible paths.
 */

int CountPath(x,y) {
        if x = 1 or y = 1
```

```
            return 1;                                    // 1 for first row and column
    return CountPath(x − 1, y) + CountPath(x, y − 1);    // Summation of two directions
}
```

The following example illustrates this algorithm. Every vertex in the tree structure represents a node in the grid graph. The numbers associated with the edges going upward from the vertices represent the numbers of possible paths to visit the nodes. The child nodes represent the previous node on the possible paths.

For a (4×3) grid, 4 columns and 3 rows, we have 10 possible paths to visit the right bottom node. This number is computed by computing all the nodes in the following tree. Clearly, we've computed the sub tree rooted at node $(3, 2)$ two times, the sub tree rooted at $(2,2)$ three times and so on.



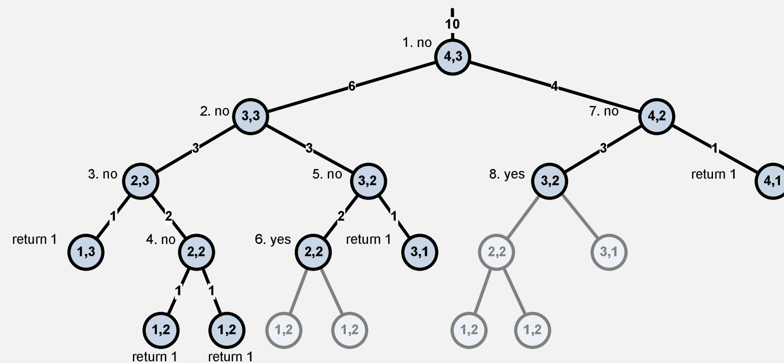
Sub trees were computed repetitively. To avoid this, dynamic programming is used. We keep a table with m rows and n columns. The table contains the values of possible paths for node (i, j). Within the algorithm, instead of computing the values whenever they're needed, we look up the table and try to fetch the value. If the node that we are looking for is not yet recoded in the table, we would go ahead computing it, like what we did in the recursive algorithm. In this case, we have to record this computed value in our table. This way, further steps don't have to re-compute this value. In other words, we are building a$(m \times n)$ table (Dynamic Programming Matrix). The entries of this table denote the numbers of possible paths to visit the nodes.

```
/**
 * Dynamic Programming Algorithm
 *
 * Function takes the row count and column count of a node in a LCS grid graph, and
 * returns the number of possible paths that we can take to get to this node starting
 * from the top left node (1, 1).
 *
 * @param x The row count of the current node.
 * @param y The column count of the current node.
 * @param M The dynamic programming matrix.
 *
 */

void CountPath(x, y, M) {
    if x = 1 or y = 1
            return 1;                                    // 1 for first row and column
```

```
        int top, left;
        if M_{x-1,y} doesn't contain a valid value {        // Obtain value of node (x-1,y)
                top = CountPath(x-1,y);                     // if possible, otherwise update
                M_{x-1,y} = top;                            // the recording table
        }
        else
                top = M_{x-1,y};
        if M_{x,y-1} doesn't contain a valid value {        // Obtain value of node (x,y-1)
                left = CountPath(x,y-1);                    // if possible, otherwise update
                M_{x,y-1} = left;                           // the recording table
        }
        else
                left = M_{x,y-1};
        return top + left;                                  // Summation of two directions
}
```

Apply the dynamic programming algorithm to the same example, we have the following procedures. The labels with numeric numbers indicate the order of corresponding dynamic programming matrix entry that was queried. The faded out sections indicate the duplicated computations in the simple recursive algorithm, which, in this algorithm, were done by looking up the dynamic programming matrix. The actual nodes that were computed in this algorithm are represented by the dark colored nodes. As we can see, we've saved many operations.



The order of query operations happened in the dynamic programming matrix entries. The alphabetic letters indicate the order.

i/j	*1*	*2*	*3*	*4*
1	none	none	returned	returned
2	returned	d. queried	e. queried	f. queried
3	returned	c. queried	b. queried	a. queried

The order of filling in the dynamic programming matrix cells. The numbers of possible paths to each node is recorded in the dynamic programming matrix. The letters indicate the order.

i/j	*1*	*2*	*3*	*4*
1	—	—	—	—
2	—	a. recorded	c. recorded	e. recorded
3	—	b. recorded	d. recorded	f. recorded

The values that were recorded as the algorithm executes. They indicate the numbers of possible paths to visit that nodes.

i/j	*1*	*2*	*3*	*4*
1	1	1	1	1
2	1	2	3	4
3	1	3	6	10

b. What is the complexity of your algorithm?

Answer: The simple recursive algorithm spends time building the tree, and it is slow, since the tree size grows fast as the height goes up. The dynamic programming algorithm save many branches of the tree and therefore increase the runtime performance. In order to answer by how much the dynamic programming approach saves, we can look at it from a different perspective. The algorithm is in fact building the dynamic programming matrix. Once the matrix is built, the last entry of the matrix is the number of possible paths from top left to the bottom right. On top of that, there are query operations. We can easily implement a collector data structure that has $O(1)$ access time. If we do so, the time consumption for query operations should be considerably inferior. Therefore the overall time complexity is building the table, which is $O(m \times n)$.

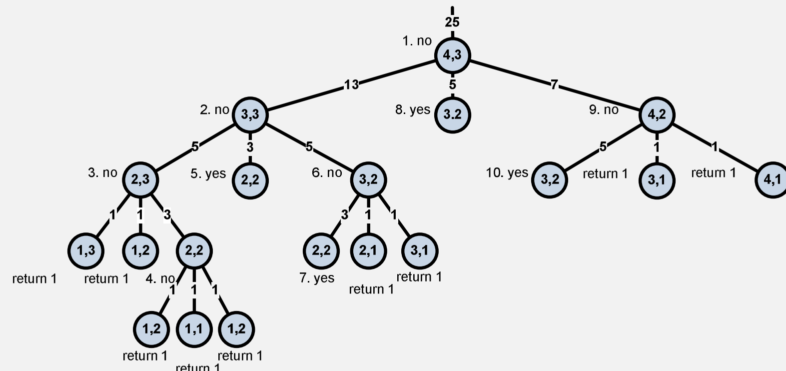
c. Suppose that each vertex has an in-degree of 3 instead of 2, what would be the complexity of the new algorithm?

Answer: The algorithm is basically the same; we just need to sum up the number of paths contributed by the third possible previous node. In the tree representation below, I didn't show the faded out braches. But we can see that every node showing as "yes" in the graph contains a sub tree if we were using the simple recursive algorithm. We've saved even more operations versus having two possible precedence paths.

```
/**
 * Dynamic Programming Algorithm
 *
 * Function takes the row count and column count of a node in a LCS grid graph, and
 * returns the number of possible paths that we can take to get to this node starting
 * from the top left node (1, 1).  The grid graph contains diagonal lines.
 *
 * @param x The row count of the current node.
 * @param y The column count of the current node.
 * @param M The dynamic programming matrix.
 *
 */

void CountPath(x,y,M) {
        if x = 1 or y = 1
                return 1;                                // 1 for first row and column
```

```
        int top, left, diagnol;
        if M_{x-1,y} doesn't contain a valid value {
                top = CountPath(x - 1, y);              // Obtain value of node (x - 1, y)
                M_{x-1,y} = top;                        // if possible, otherwise update
        }                                               // the recording table
        else
                top = M_{x-1,y};
        if M_{x-1,y-1} doesn't contain a valid value {
                diagnol = CountPath(x - 1, y - 1);      // Obtain not value (x - 1, y - 1)
                M_{x-1,y-1} = diagnol;                  // if possible, otherwise update
        }                                               // the recording table
        else
                diagnol = M_{x-1,y-1};
        if M_{x,y-1} doesn't contain a valid value {
                left = CountPath(x, y - 1);             // Obtain value of node (x, y - 1)
                M_{x,y-1} = left;                       // if possible, otherwise update
        }                                               // the recording table
        else
                left = M_{x,y-1};
        return top + left + diagnol;                    // Summation of two directions
}
```



The order of operations and the operations happened in the dynamic programming matrix entries. The alphabetic letters indicate the order.

i/j	*1*	*2*	*3*	*4*
1	returned	returned	returned	returned
2	returned	d. queried	e. queried	f. queried
3	returned	c. queried	b. queried	a. queried

The order of the dynamic matrix was filled in. The order of the numbers of possible paths to each node is recorded in the dynamic programming matrix. The letters indicate the order.

i/j	*1*	*2*	*3*	*4*
1	—	—	—	—
2	—	a. recorded	c. recorded	e. recorded
3	—	b. recorded	d. recorded	f. recorded

The values that were recorded as the algorithm executes. They indicate the number of possible paths to visit that node.

	1	*2*	*3*	*4*
1	1	1	1	1
2	1	3	5	7
3	1	5	13	25

We've queried more times for dynamic programming matrix entries. The number of computations it took to fill up the table stays the same. Hence, the overall time complexity stays the same, $O(m \times n)$.

Exercise 2

Question: The score of a multiple alignment can be defined as the sum of the entropies of the columns, which are defined to be:

$$\sum_{x \in A'} p_x \log_2 p_x$$

where p_x is the frequency of the letter x in A' in a given column. Note that, the more conserved the column is, the larger the entropy sore is. Refer to the example seen in class. To align a sequence to a profile, one can replace the sore matrix delta with the entropy formula given above and use the pair-wise dynamic programming algorithm seen in class.

The scores of aligning the following sequence and the following profile

Sequence: CTA

$$\text{Profile: } \begin{matrix} C & - & A \\ C & G & A \\ C & G & - \end{matrix} \quad \Rightarrow \quad \begin{matrix} 1 & 2 & 3 \\ C & - & A \\ C & G & A \\ C & G & - \\ C & T & A \end{matrix}$$

The scores for the columns are:

$$Score_{column_1} = \frac{4}{4} \log \frac{4}{4} = 0$$

$$Score_{column_2} = \frac{1}{4} \log \frac{1}{4} + \frac{2}{4} \log \frac{2}{4} + \frac{1}{4} \log \frac{1}{4} = -1.5$$

$$Score_{column_3} = \frac{3}{4} \log \frac{3}{4} + \frac{1}{4} \log \frac{1}{4} = -0.81$$

The alignment score is:

$$Score_{Alignment} = Score_{column_1} + Score_{column_2} + Score_{column_3}$$
$$= 0 - 1.5 - 0.81$$
$$= -2.31\,.$$

a. Based on the pair-wise dynamic programming algorithm seen in class, write the recurrence equation that align a sequence with a profile using the entropy formula as a scoring function. Remember that, as seen in class, for every (i,j) where $i > 0$ and $j > 0$, there are 3 possible alternatives.

1. Aligning position i in sequence S with a gap
2. Aligning position j in profile P with a gap
3. Aligning position i in sequence S with position j in profile P

Answer: I would initialize the first cell $a_{0,0} = 0$. The rest of the cells following the recurrence relation described below. There are three possible paths. If we start from $a_{i-1,j-1}$, it's aligning the i,jth entry with the corresponding profile $profile_j$. If we start from $a_{i,j-1}$, then we are aligning a gap with the corresponding profile $profile_j$. If we start from $a_{i-1,j}$, then we are aligning the i,jth entry with a gap in the profile. We would consider the maximum of these three paths and record it as the cost of the current node i,j. Therefore, we have the following recurrence relation:

$$a_{i,j} = max\begin{cases} a_{i-1,j-1} & + \quad Score_{i,j\leftrightarrow profile_j} \\ a_{i,j-1} & + \quad Score_{"-"\leftrightarrow profile_j} \\ a_{i-1,j} & + \quad Score_{i,j\leftrightarrow profile"-"} \end{cases}$$

Further examining this relation, we have $Score_{i,j\leftrightarrow profile"-"} = 0.81$ because all three gaps in the profile and one different entry as the ith element in the sequence. We can also pre-calculate the values of $Score_{"-"\leftrightarrow profile_j}$ for each column. So the recurrence relation is:

$$a_{i,j} = max\begin{cases} a_{i-1,j-1} & + \quad Score_{i,j\leftrightarrow profile_j} \\ a_{i,j-1} & + \quad Score_{"-"\leftrightarrow profile_j} (look\ up\ from\ the\ table\ below) \\ a_{i-1,j} & + \quad 0.81 \end{cases}$$

	1	2	3	4	5	6	7
	A	G	G	T	—	C	—
0	—	G	—	T	T	C	G
	T	G	—	A	A	C	—
$Score_{"-"\leftrightarrow profile_j}$:	1.5	0.81	0.81	1.5	1.5	0.81	0.81

If we use symbols \, <, and ^ to represent coming from $a_{i-1,j-1}$, $a_{i,j-1}$, and $a_{i-1,j}$ respectively, the table in this example would look like:

		1	2	3	4	5	6	7
		A	G	G	T	—	C	—
	0	—	G	—	T	T	C	G
		T	G	—	A	A	C	—
0	0	<-1.5	<-2.31	<-3.12	<-4.62	<-6.12	<-6.93	<-7.74
G	^-0.81	\-2	\-1.5	<-2.31	<-3.81	<-5.31	<-6.12	<-6.93
T	^-1.62	\-2.31	^-2.31	\-3	\-3.12	<-4.62	<-5.43	<-6.24
C	^-2.43	^-3.11	\^-3.11	\-3.31	^-3.93	\-5.12	\-4.62	<-5.43

Then we would back track to find out the alignment for the specified profile with the example sequence: GTC.

		A	G	G	T	—	C	—
	0	—	G	—	T	T	C	G
		T	G	—	A	A	C	—
0	**0**	**<-1.5**	<-2.31	<-3.12	<-4.62	<-6.12	<-6.93	<-7.74
G	^-0.81	\-2	**\-1.5**	**<-2.31**	<-3.81	<-5.31	<-6.12	<-6.93
T	^-1.62	\-2.31	^-2.31	\-3	**\-3.12**	**<-4.62**	<-5.43	<-6.24
C	^-2.43	^-3.11	\^-3.11	\-3.31	^-3.93	\-5.12	**\-4.62**	**<-5.43**

This indicates the following alignment between the given profile and the sequence:

	A	G	G	T	—	C	—
Profile:	—	G	—	T	T	C	G
	T	G	—	A	A	C	—
	↕	↕	↕	↕	↕	↕	↕
Sequence:	—	G	—	T	—	C	—

By observing the sequence and profile, because they are simple enough to look at, it's clear that the result alignment is a reasonable alignment output. The score of this alignment is:

$$
\begin{aligned}
Score_{Alignment} &= Score_{column_1} + Score_{column_2} + \cdots + Score_{column_7} \\
&= -1.5 - 0 - 0.81 - 0.81 - 1.5 - 0 - 0.81 \\
&= -5.43\,.
\end{aligned}
$$

It is the same as what we obtained from tracing the algorithm in the table above.

b. Use the recurrence equations to align the sequence $CGTCAG$ and the profile:

	1	2	3	4	5	6	7
	A	G	G	T	—	C	—
Profile:	—	G	—	T	T	C	G
	T	G	—	A	A	C	—

Fill up the following corresponding dynamic programming matrix.

Answer: Follow the procedure illustrated as the previous simple example, we obtain the following matrix.

		1	2	3	4	5	6	7
		A	G	G	T	—	C	—
	0	—	G	—	T	T	C	G
		T	G	—	A	A	C	—
0	0	<-1.5	<-2.31	<-3.12	<-4.62	<-6.12	<-6.93	<-7.74
C	^-0.81	\-2	\-2.31	\-2.43	<-3.93	<-5.43	<-6.24	<-7.05
G	^-1.62	\^-2.81	\-2	<-2.81	\-3.93	<-5.43	\-6.24	<-6.43
T	^-2.43	\-3.12	^-2.81	<^-3.62	\-3.62	<-5.12	<-5.93	<-6.74
C	^-3.24	^-3.93	^-3.62	\-4.31	^-4.43	\-5.62	\-5.12	<-5.93
A	^-4.05	\^-4.74	^-4.43	\^-5.12	^-5.12	\-5.93	^-5.93	\^-6.74
G	^-4.86	^-5.55	\-4.74	\-5.43	^-6.02	<-6.74	\^-6.74	\-6.93

We trace back to obtain the alignment:

		1	2	3	4	5	6	7
		A	G	G	T	—	C	—
	0	—	G	—	T	T	C	G
		T	G	—	A	A	C	—
0	0	<-1.5	<-2.31	<-3.12	<-4.62	<-6.12	<-6.93	<-7.74
C	^-0.81	\-2	\-2.31	\-2.43	<-3.93	<-5.43	<-6.24	<-7.05
G	^-1.62	\^-2.81	\-2	<-2.81	\-3.93	<-5.43	\-6.24	<-6.43
T	^-2.43	\-3.12	^-2.81	<^-3.62	\-3.62	<-5.12	<-5.93	<-6.74
C	^-3.24	^-3.93	^-3.62	\-4.31	^-4.43	\-5.62	\-5.12	<-5.93
A	^-4.05	\^-4.74	^-4.43	\^-5.12	^-5.12	\-5.93	^-5.93	\^-6.74
G	^-4.86	^-5.55	\-4.74	\-5.43	^-6.02	<-6.74	\^-6.74	\-6.93

From this back track, we have the following alignment:

	1	2	3	4	5	6	7	8
	A	G	G	T	—	C	—	—
Profile:	—	G	—	T	T	C	—	G
	T	G	—	A	A	C	—	—
	↕	↕	↕	↕	↕	↕	↕	↕
Sequence:	C	G	—	T	—	C	A	G

We can compute the overall score of this alignment, which should be the same as in the table.

$$Score_{Alignment} = Score_{column_1} + Score_{column_2} + \cdots + Score_{column_8}$$

$$= -2 - 0 - 0.81 - 0.81 - 1.5 - 0 - 0.81 - 1$$

$$= -6.93\,.$$