8/12

CS 1510
Algorithm Design

Dynamic Programming

Problems 6, 7, and 8

Due Monday September 15, 2014

Buck Young and Rob Brown

44/65

67.7%

Problem 6

Input: An array of vertexes $P = [n_1, n_2, ..., n_N]$ such that n_1 connect with n_2, n_2 connects with n_3, and so on until n_N connects with n_1 to form a polygon. Note that each n_i is an ordered pair of (x, y) coordinates in Cartesian space.

Output: The triangulation of P into $N - 2$ triangles such that the sum of the perimeters of the triangles is minimized. Or, equivalently, that the sum of the cuts to form these triangles is minimized.

Algorithm:

6

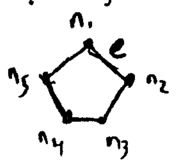
```
1: $CUT[N][N]$;
2: for $0 \leq i \leq N$ do
3:     for $0 \leq j \leq N$ do
4:         if $num\_nodes\_from(P[i], P[j]) == 3$ then
5:             $CUT[i][j] = 0$
6:         else if $num\_nodes\_from(P[i], P[j]) < 3$ then
7:             $CUT[i][j] = \infty$
8:
9: for $0 \leq i \leq N$ do
10:     for $N \leq j \leq 0$ do
11:         if $CUT[i][j] == \infty$ or $CUT[i][j] == 0$ then
12:             continue
13:         $e'_i = norm(P[(i-1)\%N], P[j])$
14:         $e'_j = norm(P[i], P[(j+1)\%N])$
15:         $CUT[i][j] = min(CUT[(i-1)\%N][j] + e'_i,\ CUT[i][(j+1)\%N] + e'_j)$
```

At a high level, each $(i,\ j)$ index pair represents a subset of the original polygon, containing only the nodes between P[i] and P[j] (the excluded nodes have been removed to form a "cut" for triangulation).

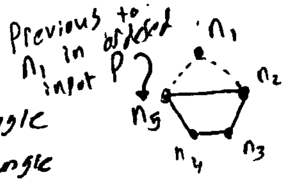
On lines 5 and 7 we see our two base-cases. Both of these depend on the theoretical (but easily implementable) function "$num\_nodes\_from(n_1, n_2)$". This function would find the number of nodes between (and including) n_1 and n_2 in P. The base-case at line 5 is equivalent to the final step of recursion, where the polygon has been reduced to 3 nodes/edges, and cannot be cut any further. Line 7 refers to illogical states where the polygon is no longer closed (ie, $nodes \leq 2$).

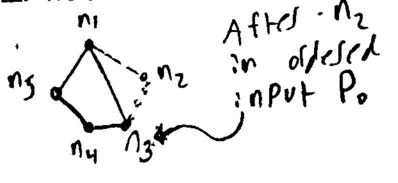
Pretty confusing

The two lengths on 13 and 14 are the lengths of the cuts that would be made if we excluded node P[i] and P[j], respectively. The only meaningful values in our table are when i and j are ±1 index from each other. Since our input is ordered so that the i^{th} node is connected to the $(i+1)^{th}$ node (and vice-versa), we can think of P[i] and P[j] as members of an edge in this case. This edge can be contained in two triangles: one where P[i] is excluded and P[j] is connected to the node preceding P[i] in P, and one where P[j] is excluded and P[i] is connected with the node following P[j] in P. See diagram below. These two pairs of nodes, again, can be considered to be edges or vectors, and we can compute the norm/magnitude of them. We call these two values e'_i and e'_j and they are the length of our cut.



$e = \{n_1, n_2\}$
exclude $n_1 \Rightarrow$ one triangle
exclude $n_2 \Rightarrow$ other triangle

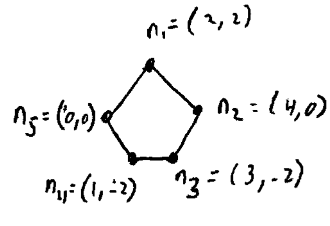




On line 15 we then choose our preferred cut to be the minimum between e'_i and all of its prior cuts and e'_j and all of its prior cuts.

Finding the solution using the table In general, to find our triangulation, we start at the index $(i,\ j) = (N-1,\ 0)$ of CUT. From here, we calculate $e'_i = norm(P[i-1][[j])$ and $e'_j = norm(P[i][j+1])$ (the lengths of the cut edges). For one of these two norm values, we will find $CUT[i][j] - n_i = CUT[i-1][j]$ or $CUT[i][j] - n_j = CUT[i][j+1]$. For the matching case, add the corresponding cut (e'_i or e'_j), move to that index $(i,\ j)$, and continue until $CUT[i][j] = 0$.

As an example of reconstruction, consider the polygon and table below. This can be interpreted as looking at the polygon with nodes ranging from n_1 to n_5 (ie, the whole polygon). Per the solution instructions above, consider $CUT[N-1][0] = CUT[4][0] = 7.606$. We calculate $e'_i = norm(P[3],\ P[0]) = norm(n_4,\ n_1) = 4.1231$ and $e'_j = norm(P[4],\ P[1]) = norm(n_5,\ n_2) = 4$. Now $P[4][0] - e'_j = P[4][1]$. So we add e'_j to our cut list, increment j and move to $(i,\ j) = (4,\ 1)$. Again, we calculate e'_j and e'_i, and find $e'_j = norm(n_5,\ n_3) = 3.606$ and $e'_i = norm(n_4,\ n_2) = 3.606$. So our last cut is arbitrary and we choose e'_j. Thus we have $OUTPUT = \{(n_5,\ n_2), (n_5,\ n_3)\}$ and our polygon is segmented as shown below.



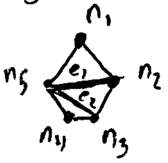
$P = [n_1\ n_2\ n_3\ n_4\ n_5]$

		n_1	n_2	n_3	n_4	n_5
				i		
		0	1	2	3	4
n_1	0	∞	∞	0	3.606	7.606
n_2	1	∞	∞	∞	0	3.606
n_3 j	2	∞	∞	∞	∞	0
n_4	3	∞	0	∞	∞	∞
n_5	4	∞	∞	0	∞	∞

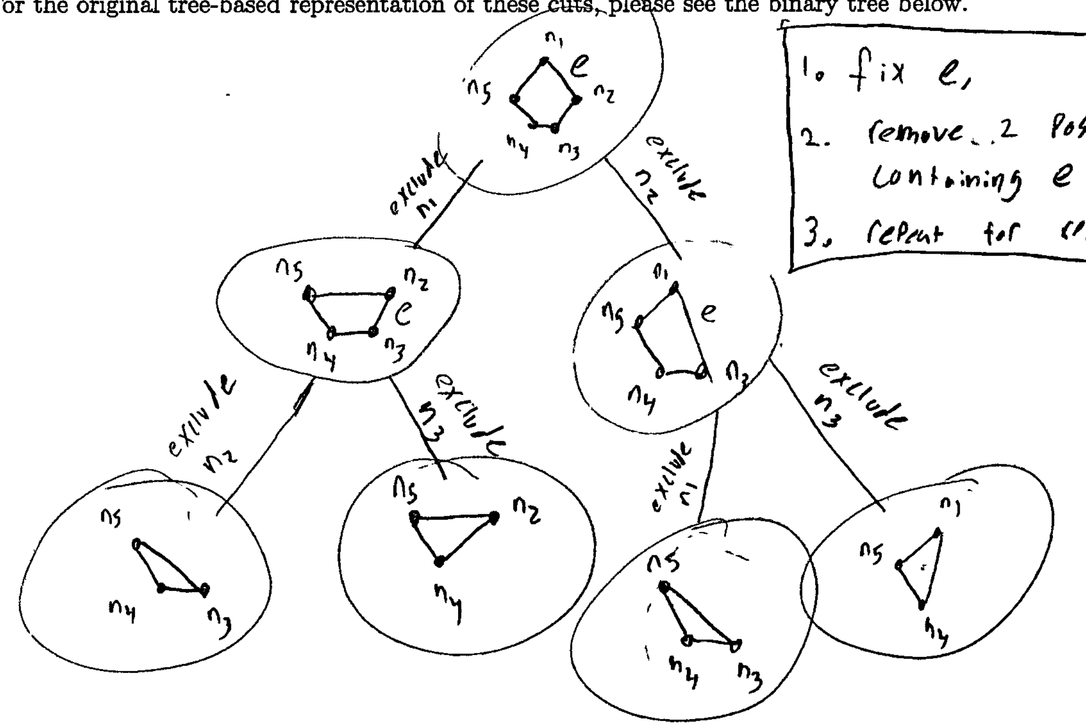
√ $7.606 - norm(n_5, n_2) = 3.606$

√ $3.606 - norm(n_5, n_3) = 0$

$output = \{e_1, e_2\}$



For the original tree-based representation of these cuts, please see the binary tree below.



1. fix e,
2. remove 2 possible triangles containing e
3. repeat for resulting polygons

Problem 7

Input: Sequence S of integers (positive, negative, or zero)

Output: A consecutive sub-sequence with the maximum sum (Max. Consecutive Sum, MCS)

Notes on the Naive Recursive Algorithm: The naive recursive algorithm would be knowing the n^{th} number and the Maximum Consecutive Sum of the first $n-1$ numbers [MCS($n-1$)]. This is not sufficient information for computing the Maximum Consecutive Sum of the first n numbers [MCS(n)] because we do not know if MCS($n-1$) is a subsequence next to the n^{th} number. That is, we do not know if the specific number $n-1$ is included in the MCS of the first $n-1$ numbers. This is an important piece of information if we want to compute MCS(n) because the solution must be *consecutive*. Therefore if we wanted to attempt to include the number n in our solution, the number $n-1$ **must** be contained in the solution. However, we do not have this information with the naive recursive algorithm.

Therefore, when we convert a proper algorithm to the iterative solution, we would be smart to compute both the MCS of the first $n-1$ numbers and also the the MCS of the first $n-1$ numbers with the number $n-1$ included. This way, we can compare MCS($n-1$) to MCS_with_n_minus_one_included($n-1$)+n. Now we know if including the number n would result in a more optimal solution.

Algorithm: Here is the iterative, array-based algorithm – I will explain it below.

1

```
1: for j=1 to n do
2:   for k=1 to n do
3:     if k == 1 or j == 1 or j > k then
4:       MCS[k][j] = -∞
5:     else
6:       int MCS_with_k_included = $\sum_{a=k-j+1}^{k} S_a$
7:       MCS[k][j] = max( MCS[k-1][j], MCS[k][j-1], MCS_with_k_included )
```

Not linear time

This algorithm will help us build a table to find the MCS. Let me explain the algorithm line-by-line.

L1&L2: Let j be the maximum length of the subsequence that is being considered. Let k be the maximum index of input S that is being considered. Let n be the number of items in input S. We will start building our table in the upper-left corner and work towards the lower-right corner, going "down" along k rows before changing columns by incrementing j.

L3&L4: This is basically our initialization for the table. We are doing it within the main loop because it is safe to do so. We want the row k=1, the column j=1, and an upper-right portion of the table to all be initialized to negative infinity. This will ensure that those cells are not seriously considered during a call to max() while we are building the table. Further, a significant upper-right portion of the table just doesn't make sense – that is the case when $j > k$. You cannot have a subsequence of length j that is longer than all the numbers up to S_k.

L6: Here we are finding the Maximum Consecutive Sequence which includes the number at S_k. Basically, this summation will add S_{k-1} to S_k if the length j is 2. Similarly, it will add S_{k-2}, S_{k-1}, and S_k together if length j is 3. Et cetera.

L7: Finally, our algorithm will apply a value to the current cell based on the maximum of three numbers. Those numbers are the cell to the left, the cell above, and the MCS with S_k included. It looks to the left to see if there is a better MCS of a smaller length. It looks above to see if there is a better MCS at anytime before S_k. And finally it looks at the MCS with S_k included to determine the benefit of adding S_k to the solution (of length j). This process will continue until we reach MCS[n][n]. From here we will have to determine the solution by tracing through the table.

Finding the solution using the table: First we start at MCS[n][n] and take note of the number there. This value is the sum of our maximum consecutive sequence – but we want to know the sequence. Therefore we must trace left and up until we find the last occurrence of our number. That is, we want the smallest possible length j and smallest possible index k which is "connected" to our maximum sum. This tells us the length of the MCS and the last index of S to include. Using this information, we simply add S_k, $S_k - 1$, ... S_{k-j+1} to the solution and we have our answer! Below is an example input, table building, and finding the solution using the table.

k 1 2 3 4 5 6

S={$\boxed{6, -5, 6, 7,}$-10, 3}

Solution

j

		1	2	3	$\boxed{4}$	5	6
	1	-∞	-∞	-∞	-∞	-∞	-∞
	2	-∞	1	-∞	-∞	-∞	-∞
k	3	-∞	1	7	-∞	-∞	-∞
	$\boxed{4}$	-∞	13	13	14	-∞	-∞
	5	-∞	13	13	14	14	-∞
	6	-∞	13	13	14	14	14

If we follow 14 to the left and up the most, we get j=4 and k=4. Now we know that the maximum sum consecutive subsequence is of length 4 and it ends at S_4. Therefore we add those four numbers from the input to our solution and get the optimal sequence: {6, -5, 6, 7}.

Problem 8

Input: Tree T with integer weighted edges (positive, negative, or zero).

Output: The shortest simple path in T, (Shortest Simple Path, SSP).

Algorithm: Here is the iterative, array-based algorithm. Explanation below.

Note: The input is being interpreted as a set of edges and associated values given sequentially numerated n nodes from 1 to n. Example Input: $\{e_{1,2} = 5,\ e_{1,3} = 10\}$ would describe a tree with 3 nodes, with the edge from node-1 to node-2 having a weight of 5 and the edge from node-1 to node-3 having a weight of 10.

```
1: for i=n to 1 do
2:     for j=n to 1 do
3:         if $e_{i,j}$ does not exist or $i > j$ then
4:             SSP[i][j] = ∞
5:         else
6:             int min_in_col = ∞
7:             for a = j to n do
8:                 if SSP[j][a] < min_in_col then
9:                     min_in_col = SSP[j][a]
10:            SSP[i][j] = min( $e_{i,j}$, ($e_{i,j}$ + min_in_col) )
```

Not linear time

This algorithm will help us build a table to find the SSP. Let me explain the algorithm line-by-line.

$L1\&L2$: We will build the table from the bottom-right to the upper-left – going "up" along j rows before decrementing the i column. Both i and j represent nodes and $e_{i,j}$ is the edge between node i and node j.

$L3\&L4$: Here is the initialization of the table, basically. We are performing it in the main loop because it is safe to do so. If the edge does not exist, we will set that cell to infinity. Note that any node "to" itself will get a value of infinity by this definition. Further the upper right half of the table will get infinity values. Therefore, we are basically creating a diagonal of infinities down the center-diagonal of the table and also throwing out the upper right half. Finally, to reiterate, we are also putting infinity in for any edges that do not exist as we build the table.

$L5$: Else if the edge does exist...

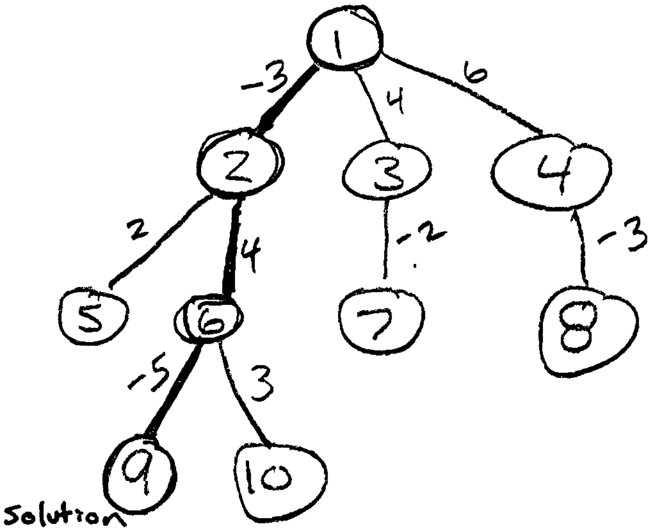
$L1 - L9$: Here we are simply finding the minimum value in a column. The column we are looking in is where $i{=}j$. When we state "for a=j to n", we are simply looking at every value in the column from value j on down (this is an optimization because we know that every value in the column $i{=}j$ from j on up is infinity). So, we are simply trying to find the minimum number in the column $i{=}j$. We do this so we can, essentially, find the SSP of that subtree.

$L10$: Here we finally set a value in the cell. This value should be the minimum between two numbers. Those two numbers are the edge itself and the edge plus min_in_col (which is the SSP of a subtree). By doing this, we can determine if the descendant subtree plus the edge in consideration is worth adding to the solution or if we should just add the considered edge by itself. Next we will discuss how to find the solution by using the table which we just created.

Finding the solution using the table: This process is relatively simple. First we find the absolute minimum number on the table and remember that number. Then we add that edge ($e_{i,j}$) to our solution set. Now we go to the column i=j and find the minimum number in that column and add $e_{i,j}$ to our solution set. We again goto the column i=j and... etc. etc. We continue in this manner while the absolute minimum number is still less than the sum of the edges in the solution set. As soon as they are equal, we stop.

More simply, we have an absolute minimum number and each edge we add contributes to that minimum value. We continue to add the edges in the solution set until we have reached our minimum value. Below is an example input tree, the created table, and the trace through the table using the methods described.

Tree:



Input: {$e_{1,2}$=-3, $e_{1,3}$=4, $e_{1,4}$=6, $e_{2,5}$=2, $e_{2,6}$=4, $e_{3,7}$=-2, $e_{4,8}$=-3, $e_{6,9}$=-5, $e_{6,10}$=3}

Table:

i=j i=2

i=j i=6

i

	j	$\boxed{1}$	$\boxed{2}$	3	4	5	$\boxed{6}$	7	8	9	10
	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
$e_{1,2}$	$\boxed{2}$	-4	∞	∞	∞	∞	∞	∞	∞	∞	∞
	3	2	∞	∞	∞	∞	∞	∞	∞	∞	∞
	4	3	∞	∞	∞	∞	∞	∞	∞	∞	∞
	5	∞	2	∞	∞	∞	∞	∞	∞	∞	∞
	$\boxed{6}$	∞	-1	∞	∞	∞	∞	∞	∞	∞	∞
$e_{2,6}$	7	∞	∞	-2	∞	∞	∞	∞	∞	∞	∞
	8	∞	∞	∞	-3	∞	∞	∞	∞	∞	∞
	$\boxed{9}$	∞	∞	∞	∞	∞	-5	∞	∞	∞	∞
$e_{6,9}$	10	∞	∞	∞	∞	∞	3	∞	∞	∞	∞

absolute min = −4

solution $\sum e_{1,2} = -3$, ... abs min < $\sum$ solution set

$e_{2,6} = 4$, ... abs min < $\sum$ solution set

$e_{6,9} = -5$ } ... abs min = $\sum$ solution set

∴ Stop!

Solution: {$e_{1,2}$, $e_{2,6}$, $e_{6,9}$}