CS 1510: Homework 10

John Hofrichter jmh162@pitt.edu
Kyra F. Lee kfl5@pitt.edu
Zach Sadler zps6@pitt.edu

September 19, 2013

Dynamic Programming

Problem 11a

```
s = L
for k = N down to 1 do
  if val[k-1, s] != val[k, s] then
    solutions.add(objects_{k-1})
    s = s - objects_{k-1}.weight
  end if
end for
```

Let $v_1, v_2, \ldots v_n$ be the vertices of the polygon, numbered clockwise.

Recursively, this algorithm works for vertices numbered between l and r by fixing the edge between v_l and v_r, and then trying all of the other possible vertices (between l+1 and r-1) as possible best third points for the triangle containing that edge. (All of the polygon's edges must be part of some three-vertex triangle.) The best third point is the one that minimizes the sum of the new edges (between v_l and v_x and between v_x and v_r) with the recursive solutions to the new polygons bounded by l and x and x and r.

To transform this into a dynamic programming solution, we build an n-by-n array of all possible bounds, of which we are only interested in the upper-right triangle, where the right bound is to the right of the left bound. We fill the array in bottom-to-top and left-to-right.

In order for this algorithm to return only the length of the additional cuts made, we define the triangulation distance of three or fewer points to be

zero, and we let the distance function return zero when given two consecutive vertices (so $\text{dist}(v_i, v_{i+1}) = 0$). Otherwise, the distance is calculated normally. To return the complete sum of perimeters, simply add the perimeter of the original polygon.

```
for i=n downto 1 do
  for j=i to n do
    if j-i < 3 then
      perimeter[i][j] = 0
    else
      min = ∞
      for x=i+1 to j-1 do
        x-diff = dist(v_i, v_x) + dist(v_x, v_j) + perimeter[i][x] + perimeter[x][j]
        if x-diff < min then
          min = x-diff
        end if
      end for
      perimeter[i][j] = min
    end if
  end for
end for
```

Dynamic Programming Problem 8

Talking through the algorithm

Let our tree T be given as input.
We will look at the tree from the bottom up, each time looking at the leaves at the deepest level of the tree. As we work through the problem we will consider the parents of these leaves and store values into these nodes as appropriate. The general idea is that we will store the maximum value we can attain if we include this vertex in our final path, as well as the maximum value we can attain if we exclude this vertex from our final path.
In our algorithm, each step we are looking at the current deepest level of subtrees, so we can immediately throw out any edges which have value positive or zero. Our reasoning is that if we only have one move left to make, it is always better to choose not to move than to move down a positive or

zero-weighted path, as we will have no more moves after this point to potentially 'make up' for taking this path which temporarily increases our total sum which we are looking to minimize.
Thus in the first step we go through each of the subtrees which have maximum depth. We throw away all edges who have positive or zero edge weight as discussed above. Then we store into the root of each subtree the minimum sum of negative edges (or minimum negative edge if there is only one), and we call this value the minimum sum if we stay in that subtree, S_i. We also store into the root of each subtree the minimum single value of all the negative edges L_i.
So we have effectively stored for each subtree of deepest level the minimum value we can get if we continue upwards through the tree and connect to more sections of the tree L_i, as well as the minimum value we can get if we do not continue and simply stay in this subtree S_i.
Next we move to the next deepest level of subtrees. At this level we have values in the leaves themselves as well as edge weights. So replace each edge weight at this level with the edge weight + L_i of the leaf vertex it is connecting to the subtree root. This represents the minimum value we can obtain if we take this path. Next simply continue the algorithm as before, calculating the best value we can get if we leave throw a single edge and connect to other parts of the tree, as well as calculating the best value we can get if we stay in this subtree. However, make sure to save S_i as the minimum value of S_i for this subtree (as calculated before) as well as all leaves in this subtree. This value represents the best value we can obtain if we do not connect the subtree root to other parts of the tree, and as part of that it may be best to simply reuse an S_i from earlier (further down in the tree), so we must consider those in addition to the S_i we calculated two paragraphs ago.
We continue this algorithm until we are at the highest level and have only one subtree, and then take the minimum of L_i and S_i.

The algorithm

for all vertices v in tree T **do**
  $v.L_i = 0$
  $v.S_i = 0$
end for
for i=deepest level of tree to top level **do**
  for j=first subtree at level i to last subtree at level i **do**

```
      min1 = 0
      min2 = 0
      min3 = 0
      for k=first leaf of subtree j to last leaf of subtree j do
        k.L_i = k.edgeLength(j) + k.L_i
        if k.L_1 < min1 < min2 then
          min1 = k.L_i
        end if
        if min1 < k.L_i < min2 then
          min2 = k.L_i
        end if
        if k.S_i < min3 then
          min3 = k.S_i
        end if
      end for
      j.L_i = min{min1, min2}
      j.S_i = min{min3, min1+min2}
    end for
  end for
  return max{T.root.L_i, T.root.S_i}
```