CS1510 Dynamic Programming 7

Rebecca Negley, Sean Myers

September 23, 2011

7. Give an efficient algorithm for the following problem. The input is an n sided convex polygon. Assume that the polygon is specified by the Cartesian coordinates of its vertices. The output should be the triangulation of the polygon into $n-2$ triangles that minimizes the sums of the perimeters of the triangles. Note that this is equivalent to minimizing the length of the cuts required to create the triangle.

   Solution: First consider the recursion-based approach to solving this problem. Note that any cut between two non-adjacent vertices separates the n-sided convex polygon into two new convex polygons, each with fewer than n sides. If the first cut made is in an optimal solution, then the minimum perimeter triangulation of the original n-sided polygon is equivalent to the sum of the minimum perimeter triangulation of the each of the two new, smaller polygons. We do not know, however, which cut to make first, so we must try them all and accept the one that gives the minimum perimeter. The algorithm to find the minimal triangulation perimeter would then look like:

   TriPoly(n-sided convex polygon)
   If $n = 3$
   return s_1+s_2+s_3 //s_1, s_2, s_3 are side lengths
   else
   return $\min_{cut}$(TriPoly(new polygon 1) + TriPoly(new polygon 2))
   //new poly 1 and new poly 2 are new polygons created by cut

   For each k-sided polygon with $k > 3$, we check k $(k-1)$-sided inner-polygons along with other smaller polygons. Hence, this algorithm's runtime will be $runtime(n\text{-sided poly}) <$n*runtime((n-1)-sided poly) $< \ldots < n*(n-1)*\ldots*(p+1)*runtime(p-sidedpoly)$. Hence, this algorithm makes at least $n*(n-1)*\ldots*(p+1) = \frac{n!}{p!}$ recursive calls to p-sided inner-polygons. However, there are $C(n,p) = \frac{n!}{p!(n-p)!} \leq \frac{n!}{p!}$ possible p-sided inner-polygons. For $p < (n-1)$, the inequality is strictly less. Hence, by the pigeonhole principle, the recursive algorithm unnecessarily repeats recursive calls.

   We can replace this inefficient recursive algorithm with a dynamic programming algorithm. First order the n vertices, $v_1, \ldots, v_n$, in clockwise order, starting at the topmost vertex. For each p such that $3 \leq p \leq n$, construct a matrix M_p of dimension $(n-p+1)\times(n-p+1)\times\ldots\times(n-p+1) = (n-p+1)^p$. We can describe any p-sided polygon by its p vertices. We will order these vertices from least to greatest (according to the ordering defined above) so that our polygon is unique. Then, we will construct these matrices so that any $M_p[v_{i_1}][v_{i_2}]\ldots[v_{i_p}]$ such that $i_1 < i_2 < \ldots < i_p$ will contain the minimal triangulation perimeter of the p-sided polygon defined by vertices $v_{i_1}, \ldots, v_{i_p}$. Instead of using recursive calls, we will work from the bottom up and go through a series of matrix look ups to find the minimal triangulation perimeter. First, we construct the M_3 matrix, since 3 is the fewest number of sides a polygon can have. In each $M_3[v_{i_1}][v_{i_2}][v_{i_3}]$, we record the sum of the lengths of the three sides on the triangle defined by the vertices. Let $3 < p \leq n$. For each cell in M_p with increasing indices, we then record the minimum sum of all possible sets of 2 smaller polygons that create the p-sided polygon defined by the vertices of that cell. By only looking at increasing vertices, we avoid repetitive calculation. Our more efficient dynamic programming algorithm then looks as follows:

TriPoly($v_1, \ldots, v_n$)
    for i from 1 to n-2
        for j from i+1 to n-1
            for k from j+1 to n-2
                $M_3[i][j][k] = s_{i,j} + s_{j,k} + s_{i,k}$ //$s_{p,q}$ denotes the length of segment connecting v_p to v_q
    for i = 4 to n
        for j_1 from 1 to $(n - i + 1)$
            for j_2 from $(j_1 + 1)$ to $(n - i + 2)$
                $\ldots$
                for j_i from $(j_{i-1} + 1)$ to $(n - i + i)$

$$M_i[j_1][j_2]\ldots[j_i] = \min_{1 \le \alpha < n-1, \alpha+1 < \beta \le n} \Big(M_{(i-(\beta-\alpha)+2)}[j_1]\ldots[j_\alpha][j_\beta]\ldots[j_i]) + (M_{(\beta-\alpha)}[j_\alpha]\ldots[j_\beta] \Big)$$

Each inner-polygon is only calculated once, so this algorithm avoids the efficiency pitfalls of the recursive algorithm.