

**B.M.S College of Engineering**

**Bangalore-560 019**

**DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING**

**Optimal Triangulation of a Polygon using Dynamic Programming**

***A Programming Assignment Report submitted***

***By***

**Riya Gupta [1BM13IS064]**

**Rozelle Jain[1BM13IS066]**

**Amrutha Srinivasan [1BM13IS067]**

**Udit Shyam Padhi [1BM13IS083]**

***for the elective course***

**Advanced Data Structures [10CI6GEADS]**

Jan -April 2016

INTRODUCTION

Dynamic programming is a technique where the problem is solved by combining the solutions of smaller subproblems, similar to divide and conquer method.

However, the subproblems are overlapping here unlike divide and rule method. On calculation of the result of a subproblem the result is stored in a table and reused whenever required. Thus, this dependent nature of subproblems reduces the number of calculations to be done.

It can be claimed that dynamic programming gives a more optimal solution than some of greedy technique solutions and has better run time than brute force algorithms.

SOLUTION

Development of a dynamic programming algorithm can be broken down in 4 steps as shown below:

1. Characterizing the structure of the optimal solution

We should first understand the problem and find the parameter that would optimize the solution the most.

A polygon is a pair wise linear closed curve in a plane. It is a curve ending on itself that is formed by line segment or sides. The point at which two sides meet is called a vertex. A convex polygon is one which the z-components of the cross products of all the sides taken as vectors pointing towards the points in increasing order are either all positive or all negative.

Triangulation of the n-gon is a set of T chords that divide the polygon into disjoint triangles and intersect each other only on the vertices of the polygon. Every triangulation is due to n-3 chords such chords dividing the n-gon into n-2 triangles.

For triangulation we must check:

* Number of sides is more than or equals 3
* The polygon is simple
* The polygon is convex

To solve the problem we take a set P = { v0 , v1 ,….,vn-1 } as the set of n vertices of the n-gon. Ai represents side vi-1vi of the polygon. Ai…j represents the polygon made by vertices vi-1vi….vj . Thus the polygon with n+1 sides is represented by A1…n with vetices v0v1…vn.

Optimization would mean that sum of the perimeters of all the n-2 triangles should be minimum. This the weight function of this problem is w(triangle vivkvj)=dist(vivj)+dist(vjvk)+dist(vkvi)

On finding point i, j, k in the polygon that forms the least weighted triangle, we divide the polygon into 3 parts:

* The triangle vivkvj
* The polygon formed to the left of the triangle with side vkvi Ai…k
* The polygon formed to the right of the triangle with side vkvj Ak..j

We recursively can then find the cost of triangulation of the other 2 polynomials in the same way.

The main parameter that optimizes the solution is thus finding point k that divides the polynomial Ai..j.

1. Recursively define the value of the optimal solution

We first define table[i, j] to store the minimum cost of triangulation of polygon Ai...j.

We understand that Ai..j t=is the same as Aj…i  as the order in which sides are taken does not affect the optimal solution. This is why we need to fill only the upper diagonal of the matrix table containing the optimal solution to subproblems.

This means i < j for all j in n. We have to find k which is a point in this polygon that divides this polygon into a triangle and 2 more smaller polygons with optimal cost

table[i, i] = 0 as a point cannot be triangulated.

table[i, j]=0 for j<i+2 as we need minimum 3 points for form a triangle.

table[ i, j]=table [i, k]+table[k+1, j]+ w(triangle vi-1vkvj) where k can have any value between j-1 to i. We have to look for the k in this range which causes the table[i, j] to be min.

Thus recursive solution of the problem would be:

table[i, j] = 0 if j<=i+2

= mini<=k<=j-1(table[i, k] +table[k+1, j]+ w(triangle vi-1vkvj)) if j>i+2

1. Computing the optimal solution:

It is a simple matter to write a recursive algorithm based on recurrence to compute the minimum cost table[1, n]. We have relatively lesser subproblems - one for each choice of i, j satisfying 1<=i<=j<=n. A recursive algorithm may encounter same subproblem in several branches of its recursive tree. This property of overlapping subproblems is a hallmark of applicability of dynamic programming.

We use another 2 dimensional matrix index[1..n , 1…n] to keep track of where the split of the polygon is happening to achieve the optimal cost.

The table matrix and index matrix are filled in a diagonal fashion with the lower triangle staying empty as Ai…j is the same as A j…i.

Pseudo code:

for gap🡨 0 to n do //to keep track of which column to fill

for i🡨0, j🡨gap to j<n do //i for rows and j for columns

if j<i+2

then table[i, j]🡨0

else

table[i, j]=MAX

for k🡨i+1 to j-1 do

val🡨 table[i, k]+table[k+1, j]+cost(i, j,k)

if val < table[i, j] then

table[i, j]🡨 val , index[i, j]🡨k

return table, index

1. Constructing an optimal solution:

Here we construct an optimal solution from the computed information.

We find a point k such that the polygon of n+1 sides represented by A1…n can be split into two polygons A1…k  and Ak+1...n and a triangle v0vkvj.

The matrix table [0,n] would represent the optimal cost of triangulation of the polygon.

To print the index of the n-2 disjoint triangle formed, the pseudo code used is:

printTriangles(i, j, count,index[][])

if count>n-2 OR j< i+2 then return

else

print vertices vi vkvj

printTriangles(i,k,count++,index)

printTriangles(k+1 mod n,j,count++,index)

end

We can claim that this is the optimal solution to the given problem as each subproblem is an optimal solution at that level. Therefore, when this result is used to solve a bigger subproblem, obviously an optimal solution is generated. In the end, when the final solution is calculated, the result would be determined from all the small optimized subproblems. A parse tree can be used to show the various overlapping subproblems and mergers of sides to form triangles. We see that for an n-gon there are n-1 leaves representing the sides of the polygon and n-2 mergers showing the disjoint triangles.

Analyzing the algorithm:

The running time of this algorithm would be O(n3). This is because of the three nested for loops in the table-filling section of the algorithm. All other operations /sections take have a complexity of O(n) or O(1).

The space complexity of the algorithm would be θ(n2). This is because we use a 2D array to store the result of the subproblems.

SOURCE CODE

class Point

{

public int x,y;

Point()

{

x=y=0;

}

Point(int a,int b)

{

x=a;

y=b;

}

//returns false if polygon is concave and true if it is convex

boolean isConvex(Point points[],int n)

{

boolean negetive = false;

boolean positive = false;

int a,b,c;

float crossProduct;

for(a=0;a<n;a++)

{

b=(a+1)%n;

c=(b+1)%n;

crossProduct=getCrossProduct(points[a],points[b],points[c]);

if(crossProduct<0)

negetive=true;

else if(crossProduct>0)

positive=true;

if(negetive && positive)

return false;

}

return true;

}

public float getCrossProduct(Point a,Point b,Point c)

{

float BAx = a.x-b.x;

float BAy = a.y-b.y;

float BCx = c.x-b.x;

float BCy = c.y-b.y;

// Calculate the Z coordinate of the cross product.

return (BAx \* BCy - BAy \* BCx);

}

//returns true if points of the polygon are linear else returns false

public boolean isLinear(Point points[], int n)

{

Point a,b,c;

int i,j,k;

for(i=0;i<n;i++)

{

j=(i+1)%n;

k=(j+1)%n;

a=points[i];

b=points[j];

c=points[k];

if(((a.y-b.y)\*(a.x-c.x))!=((a.y-c.y)\*(a.x-b.x)))

return false;

}

return true;

}

}

import java.util.\*;

class PolygonTriangulation

{

Point points[];

int n;

public void input()

{

Scanner sc=new Scanner(System.in);

//user inputs number of points in the polygon

System.out.print("Enter no of vertices of the polygon : ");

n=sc.nextInt();

System.out.println();

//user inputs the x and y co-ordinates of the points

points=new Point[n];

int x,y;

System.out.println("Enter x and y co-ordinates of the points in cyclic order");

for(int i=0;i<n;i++)

{

x=sc.nextInt();

y=sc.nextInt();

points[i]=new Point(x,y);

}

//checking input

boolean flag1=new Point().isConvex(points,n);

boolean flag2=new Point().isLinear(points,n);

if(flag1==false)

{

System.out.println("The polygon entered in concave. Thus triangulation cannot be done!");

return;

}

else

if(flag2==true)

{

System.out.println("All points of polygon entered are linear. Thus triangulation cannot be done!");

return;

}

else

minTriangulationCost();

}

// A utility function to find distance between two points in a plane

public double dist(Point p1, Point p2)

{

return Math.sqrt((p1.x - p2.x)\*(p1.x - p2.x) + (p1.y - p2.y)\*(p1.y - p2.y));

}

// A utility function to find cost of a triangle. The perimeter is considered as cost of the triangle

public double cost(int i, int j, int k)

{

Point p1 = points[i], p2 = points[j], p3 = points[k];

return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);

}

// Main logic of dynamic programming

public void minTriangulationCost()

{

Scanner sc=new Scanner(System.in);

if(n<3)

{

System.out.println("The polygon should have atleast 3 sides, else triangulation cannot be done!");

return;

}

// Table to store results of subproblems is table[][]

// table[i][j] stores cost of triangulation of polygon with vertices/points from Pi to Pj.

// The entry table[0][n-1] stores the final result.

double table[][]=new double[n][n];

// table filled using recursive formula in diagonal fashion

// region above the diagonal is only filled and the lower region is empty

int index[][]=new int[n][n];

for(int gap = 0; gap < n; gap++)

{

//this loop keeps track of the no of columns to left empty in every row,

// since we are only fillinng the upper triangle of the matrix

for(int i=0, j=gap; j<n; j++)

{

//since atleast 3 points are needed to form a triangle

if(j < i+2 )

table[i][j] = 0.0;

else

{

table[i][j] = 1000000.0;//max value

for(int k=i+1;k<j;k++)

{

double val = table[i][k] + table[k][j] + cost(i,j,k);

if(val < table[i][j] )

{

table[i][j]=val;

index[i][j]=k;

}

}

}

}

}

//printing result

System.out.println("\n\nMinimum cost of triangulation of the polygon is : "+table[0][n-1]);

System.out.println("Triangles formed are:");

printTriangles(0,n-1,1,index);

}

public void printTriangles(int i,int j,int count,int index[][])

{

if(j<i+2 || count>n-2)

return;

else

{

int k=index[i][j];

System.out.println("V"+i+" V"+k+" V"+j);

printTriangles(i,k,count++,index);

printTriangles((k+1)%n,j,count++,index);

}

}

public static void main(String args[])

{

System.out.println("Program to find minimum cost of triangulation of polygon using dynamic programing");

new PolygonTriangulation().input();

System.out.println("\n\n-Riya Gupta\n-Rozelle Jain\n-Amrutha Srinivasan\n-Udit Shyam Padhi");

}

}

OUTPUT

Example 1

V1 V0 V3Program to find minimum cost of triangulation of polygon using dynamic programing

Enter no of vertices of the polygon : 3

Enter x and y co-ordinates of the points in cyclic order

0

0

2

0

3

0

All points of polygon entered are linear. Thus triangulation cannot be done!

Example 2

Program to find minimum cost of triangulation of polygon using dynamic programing

Enter no of vertices of the polygon : 4

Enter x and y co-ordinates of the points in cyclic order

0

0

4

0

1

2

0

6

The polygon entered in concave. Thus triangulation cannot be done!

Example 3

Program to find minimum cost of triangulation of polygon using dynamic programing

Enter no of vertices of the polygon : 3

Enter x and y co-ordinates of the points in cyclic order

0

0

6

0

0

8

Minimum cost of triangulation of the polygon is : 24.0

Triangles formed are:

V0 V1 V2

Example 4

Program to find minimum cost of triangulation of polygon using dynamic programing

Enter no of vertices of the polygon : 5

Enter x and y co-ordinates of the points in cyclic order

0

0

2

0

4

2

2

4

0

4

Minimum cost of triangulation of the polygon is : 10.47213595499958

Triangles formed are:

V0 V1 V4

V2 V0 V4

V1 V0 V4