Hard Performance Measurement

October 27 2021

Contents

1 Introduction

1.1 General Introduction

Given two dimentional shapes A and B which are represented as closed polygons, our goal is to find the optimal correspondences between points in A and B. If we simply aggregate all valid paths, the search will quickly grow to an exponential complexity.So we need to find some ways to stop tree expansion if some requirements are violated. After the tree expansion, we start the voting process, each pair within the path will receive a vote. Then put all elements in shape A and all elements in shape B into a vote table. Each element in the table stores the total votes of that pair. If pair (mi, ni) appears in a tree path, then the corresponding score in the table will increase by 1. A pair with higher votes indicates a more reliable match. Finally, our task is to search for the best match between A and B.

1.2 Input Specification

In this Project, each input file contains a test case, and each test case contains two integers$M, N(3 \leq M, N \leq 100)$, then followed by $N + M$different positions construct by two floats X, Y in shape A and B.

All the Datas are given in Rectangular coordinate system. There are two points need notice, first the given data is in float type, not integer type, second for each shape, the i-th point given is indexed as i in A or B.

1.3 Ontput Specification

In this Project, for each test case, print the correspondence points in the best match in the format (i_1, i_2), where i_1 is the index of a point in shape A, and i_2 in shape B. Each pair in a line, given in increasing order of shape A indices.

2 Algorithm Specification

2.1 Data Structure

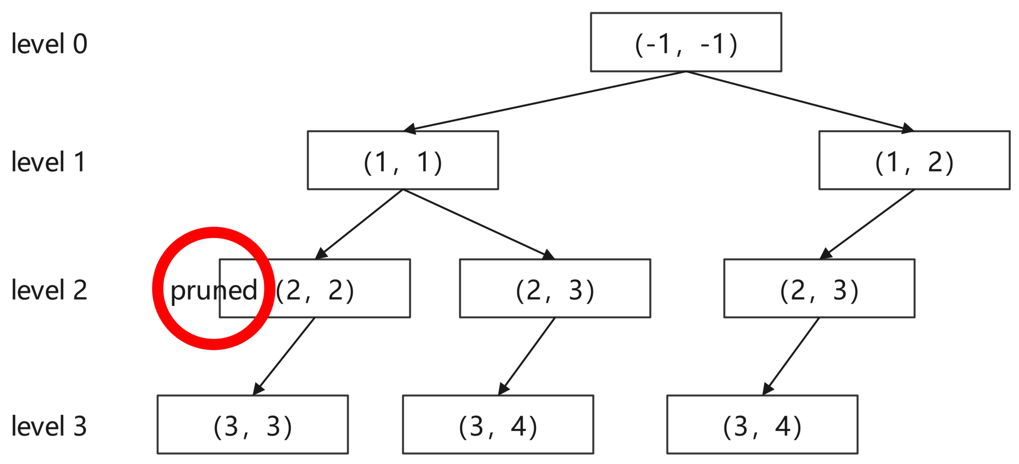

Figure 1: The Schematic Diagram of the Tree (the Sample Input)

2.1.1 struct pointxy

Record the x, y coordinates of a certain point. In this problem, It is used to record the coordinates of the point in A or B;

```
typedef struct pointxy point;
struct pointxy {
        double x, y;
};
```

And in the program, we use the code below to get two arrays of points in A and B respectively. So now we can express $i-th$ point in A as A[i] for example .

```
A = (point*)malloc(sizeof(point) * M);
B = (point*)malloc(sizeof(point) * N);
REAMP(A, M);
REAMP(B, N);
```

2.1.2 struct Tree Node

The Tree Node of the main Tree.

”int p and q ” is used to record the coresponding serial number of points from A and B respectively. This means in this node A[p] and B[q] is the points pair.

”int flag” shows how many childs the tree node has. If $theflag == -1$, we regard it as the tree has no child. It is important because in follow-up code we will use $theflag == -1$ to prune the tree (stop the tree to expend).

”int level” shows where the level the tree is in. We define the first level is 0, it used to present the root of the tree and has no real meaning. And if the level is larger than 0, it shows the serial number of points from A. What we should do is to find the best path from the level 0 (root) to the leaf.

”double sumdis” shows the sum delta distance and delta angle from the root to this child, and it is mainly used to grade the path.

”Tree *k” uses with i, to move to the child of the node. So it is not fixed num, but a num based on flag.

”Tree parent” is uesd for searching parent.

```c
typedef struct TreeNode* Tree;
struct TreeNode {
        int p;
        int q;
        int flag;
        Tree *k;
        int level;
        double sumdis;
        Tree parent;
};
```

An example of how to malloc the *k and the Tree Node of the number of flag which *k points to is below. And you can know it obeviously is the *k of the tree root. And it also contains how to initialize a node.

```c
root->k = (Tree*)malloc(sizeof(Tree) * (M - N));
for (i = 0; i <= M - N; i++)
{
        root->k[i] = (Tree)malloc(sizeof(struct TreeNode));
        root->k[i]->parent = root;
        root->k[i]->p = 0;
        root->k[i]->q = i;
        root->k[i]->flag = -1;
        root->k[i]->level = 0;
        root->k[i]->sumdis = 0;
        expend(root->k[i], score1, mode);
}
```

2.1.3 Voting / Grading mat

Voting / Grading mat is a table to record the votes or the grades a pair of points in A and B gets. By this table, wecan finally calculate and find the best path in the tree. So we use *malloc*() to make the table as two-dimension array.

```c
score = (double **)malloc(sizeof(double *) * N);
for (i = 0; i < N; ++i)
{
        score[i] = (double *)malloc(sizeof(double) * M);
        for (j = 0; j < M; j++)
        {
                score[i][j] = 0;
        }
}
```

2.2 Limitation Factor

I use two limitation factor – the distance and the angle. And I give the detailed process below.

2.2.1 The Distance between two points

```c
double distance(point A, point B)
{
        double dis;
        dis = sqrt(pow(A.x - B.x, 2) + pow(A.y - B.y, 2));
        return dis;
}
```

2.2.2 The Angle $\angle BAC$ of three points A, B, C

```
double angle(point A, point B, point C)
{
        double ab, bc, ca;
        double theta;
        ab = distance(A, B);
        bc = distance(B, C);
        ca = distance(C, A);
        theta = (ab*ab + ca * ca - bc * bc) / (2 * ab*ca);
        return theta;
}
```

2.3 Algorithm: Tree Building, Expending and Pruning

In the next three part, I will giving the three most important algorithms.

2.3.1 Tree Building

The code is given in **2.1.2**. The main code is to build the root node and to introduce tree expending function.

2.3.2 Tree Expending

It is a recursive function, using a tree node to expend its childs or prune the tree. It mainly contains 4 parts – Parameter definition, calculate the limit factor and judge expending or pruning, Voting or Grading, expending.

PART 2: Calculate the limit factor and judge expending or pruning. In this part, we use two limit factor functions to calculate the delta distance between the node and its possible child of A and B seperately and the delta angle of the node to its child and its parent. After this, we use the coefficient constant to judge whether to expend or not. And the detailed introduction of coefficient constant is below. If the child pass the judgement, it will be add to the node, and the flag of the node adds. Also, it is necessary to initialize the new child.

```
//the max acceptable distance between child and parent
#define DISCOE 3
//the max acceptable angle between child and parent
#define ANGLECOE 0.2
//In mode one, assuming that the incomplete path can get 1 vote, the complete
    path can get PASS votes.
#define PASS 6
//The coefficient to judge the pruning and to calculate the grading of one path
#define DIFFCOE DISCOE/ANGLECOE
#define SUMCOE 3
#define VOTE(x, y)  x / (y+SUMCOE)
#define SUMDIS(x) sqrt(x + 1) * DISCOE * SUMCOE
```

PART 3: Voting or Grading This part I unified use voting or grading after the whole path is end (flag == -1). And I give two possible solution to measure the best path.

Mode 1:

The method the exercise is given. To voting the path, I revise it better to distinguish the complete and incomplete path. And assign different situation different votes (showing above – the coefficient constant PASS).

Mode 2:

The New method I give. Using the result of delta distance and delta angle to assign different path different grades. by this, we can make the proper path contribute more to the results.

```
void expend(Tree Origin, double** score, int mode)
{
//PART1: Parameter definition
int i;
```

```
double disadd = Origin->sumdis;
double deldis = 0, delang = 0;
int min, max;
int p = Origin->level + 1;
min = Origin->q + 1;
max = M - N + Origin->level + 1;
//PART2: calculate the limit factor and judge expending or pruning
Origin->k = (Tree*)malloc(sizeof(Tree) * (max-min+1));
int j = 0;
if (Origin->level < N - 1)
{
	for (i = min; i <= max; i++)
	{
	deldis = fabs(distance(A[Origin->p], A[p]) - distance(B[Origin->q], B[i]));
	if (Origin->level >= 1)
	{
		delang = fabs(angle(A[Origin->p], A[p], A[Origin->parent->p]) - angle(B[Origin->q], B[i], B[Origin->parent->q]));
	}
	disadd = deldis + delang*DIFFCOE + Origin->sumdis;
	if (deldis <= DISCOE && delang <= ANGLECOE && disadd <= SUMDIS(Origin->level))
	{
		Origin->flag++;
		Origin->k[j] = (Tree)malloc(sizeof(struct TreeNode));
		Origin->k[j]->flag = -1;
		Origin->k[j]->level = Origin->level + 1;
		Origin->k[j]->p = Origin->p + 1;
		Origin->k[j]->q = i;
		Origin->k[j]->parent = Origin;
		Origin->k[j]->sumdis = disadd;
		j++;
	}
	}
}

//PART3: Voting or Grading
Tree ptr = Origin;
if (mode == 2)
{
	if (Origin->flag == -1)
	while (ptr->level != -1)
	{
		score[ptr->p][ptr->q] += VOTE(ptr->level + 1, ptr->sumdis);
		ptr = ptr->parent;
	}
}
if (mode == 1)
{
	if (Origin->flag == -1)
	{
		Tree ptr = Origin;
		if (Origin->level != LEVEL)
		{
			while (ptr->level != -1)
```

```c
            {
                score[ptr->p][ptr->q]++;
                ptr = ptr->parent;
            }
        }
        else
        {
            while (ptr->level != -1)
            {
                score[ptr->p][ptr->q] += PASS;
                ptr = ptr->parent;
            }
        }
    }
}

//PART4: expending
for (i = 0; i < j; i++)
{
    expend(Origin->k[i], score, mode);
}
}
```

2.4 Matching problem and Reverse order problem

Because B can bereverse order to pair with A, and the pointsin A and B are difficult to determine. So I will expend the tree with B in order and reverse order, and alse, in the begining, I judge the pairing and paired point.

```c
if (M > N)
{
    LEVEL = N - 1;
    A = (point*)malloc(sizeof(point) * N);
    B = (point*)malloc(sizeof(point) * M);
    REAMP(A, N);
    REAMP(B, M);
}
```

```c

for (j = 0; j < M / 2; j++)
{
    c = B[M - j - 1];
    B[M - j - 1] = B[j];
    B[j] = c;
}
```

2.5 Using mat to find the best path

To find the best way, I add the largest element on the left side of the element in the next line to the element, and repeat it repeatedly to get the best path and the number of votes or scores.

```c
double wayfind(double **score, int N, int M, int fl)
{
    double scoref = 0;
    int i, j, k, flag = 0;
    double max = 0;

```

```

        for (j = 0; j < M; j++)
        {
                if (score[N - 1][j] != 0)
                flag = 1;
        }
        if (flag == 0) return 0;
        for (i = N - 2; i >= 0; i--)
        {
                for (j = 0; j < M; j++)
                {
                        if (score[i][j] != 0)
                        {
                                for (k = j + 1; k < M; k++)
                                {
                                        if (score[i + 1][k] > max)
                                        {
                                                max = score[i + 1][k];
                                        }
                                }
                                score[i][j] += max;
                        }
                }
        }
        scoref = score[0][0];
        return scoref;
}
```

3 Analysis and Comments

Comparing the accuracy of the two programs, it can be found that MODE 1 has certain advantages for some simple graphics. But when the two paths are both available, the program actually has multiple solutions (although only one is selectively output). And MODE 2 not only considers the comparison problem after the multi-channel is completed, but also considers the influence of the previous points on the latter points.

In the overall algorithm, the areas that can be improved are as follows. One is that when determining the end of the path, you can consider the influence of the distance of the previous point on the back point, rather than just the influence of a simple linear summation. The second is to add restrictions, such as area, etc., to make it more accurate, and even define and determine the accuracy under reasonable circumstances.

4 The TEST RESULT

4.1 SMALL NUM

4.1.1 Input

```
3 4
0 4
3 0
0 0
8 4
8 1
4.25 6
8 6
```

4.1.2 MODE 1

```
1       2       3       4
1  |    2.0     6.0  XXXXXXXXXXXX
2  |XXXXXX   1.0     7.0  XXXXXX
3  |XXXXXXXXXXXXXXXXXX   6.0

1       2       3       4
1  |XXXXXXXXXXXXXXXXXXXXXXXX
2  |XXXXXX   1.0     2.0  XXXXXX
3  |    2.0     1.0  XXXXXXXXXXXX
(1, 2)
(2, 3)
(3, 4)
```

4.1.3 MODE 2

```
1       2       3       4
1  |    0.7     0.3  XXXXXXXXXXXX
2  |XXXXXX   1.2     2.5  XXXXXX
3  |XXXXXXXXXXXXXXXXXX   2.2

1       2       3       4
1  |XXXXXXXXXXXXXXXXXXXXXXXX
2  |XXXXXX   1.2     2.6  XXXXXX
3  |    0.7     0.3  XXXXXXXXXXXX
(1, 1)
(2, 3)
(3, 4)
```

4.2 LARGE NUM

4.2.1 Input

```
12 20
1.732 1
1 1.732
0 2
-1 1.732
-1.732 1
-2 0
-1.732 -1
-1 -1.732
0 -2
1 -1.732
1.732 -1
2 0
3.03 1.732
4.1 5
2 3
0 3.47
1 2.16
-1.74 3
10 9
-3.1 1.8
```

```
0.7 8.5
−3.46 0.1
−10 4.6
−3.1 −1.7
−1.73 −3
0 −3.5
1.2 3
1.3 1.9
1.732 −3
3 −1.65
3.46 0
0 0
```

4.2.2 MODE 1

```
(1, 19)
(2, 18)
(3, 17)
(4, 13)
(5, 12)
(6, 11)
(7, 6)
(8, 5)
(9, 4)
(10, 3)
(11, 2)
(12, 1)
```

4.2.3 MODE 2

```
(1, 19)
(2, 18)
(3, 14)
(4, 13)
(5, 12)
(6, 8)
(7, 6)
(8, 5)
(9, 4)
(10, 3)
(11, 2)
(12, 1)
```

5 Declaration

I hereby declare that all the work done in this project titled *"Hard Performance Measurement"* is of my independent effort.