Voting Tree

Date:2022-10-27

Content

Chapter 1: Introduction

Given two **2-dimentional shapes** A and B which are represented as closed polygons, our goal is to find the optimal correspondences (or "match") between points in A and B.

In this program we need to find the **similarity** of two graphs and output its subgraph.

Input Specification:

Each input file contains one test case. For each case, the first line gives two positive integers M and N ($3\le M,N\le 100$) which are **the total number of points** in shape A and B respectively. The next $M+N$ lines each contains **the *x* and the *y* coordinates** (in float type) of a point in shape A and B, respectively, in clockwise order. Hence for each shape, the i-th point given is indexed as i ($i=1,\cdots,M$ or N).

Output Specification:

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.

Chapter 2: Algorithm Specification

2.1 The main function

First **read in the data**, if the size of the first picture is larger than the second picture, then **change the two positions** and record their changed positions.

Loop each point pair as a starting point to build a path, traverse it with a **depth-first search** algorithm, build a tree and get the voting results.

Next, we need to select the optimal path through voting results. Here I give two algorithms: **greedy algorithm** and **dynamic programming algorithm** (one topic requirement and one bonus).

2.2 The data structure

```
//Store the coordinates of the two images,
//the first line is the x coordinate,
//the second line is the y coordinate
double A[MAXN][2], B[MAXN][2];
//Whether the record array has been reversed
int isReversed;
//Store the vote score for each peer pair
int votingTable[MAXN][MAXN];
//The size of the two images, where A must be larger than B
int size_A, size_B;
//Store the path traveled so far
int route[MAXN][2];
//Store the path where similar graphs are finally found
int res[MAXN][2][4];
```

```
//We obtained three results by traversing
//two graphs in order at the same time,
//traversing two graphs in reverse order at the same time,
//and traversing two graphs in order and one in reverse order.
//When obtaining the final path,
//the sum of the votes for these four paths is recorded
//and stored in this array.
long score[4];
//This array is used to store the number of points
//contained in each of the four paths
int val[4];
```

Since this problem does not need to use other features of the tree, I choose to use **arrays to simulate a tree** to implement voting, which can combine the two steps of tree building and voting to **facilitate the optimization** of the algorithm.

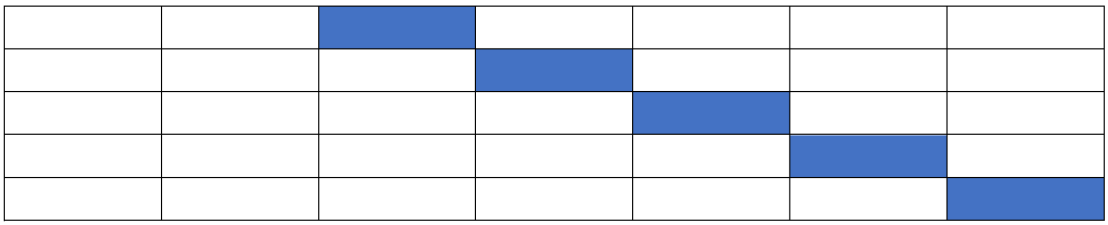
2.3 vote

First traverse each point pair as the **starting point**, and for each starting point, find the points that meet the conditions **backwards**. If this point meets the conditions, put it in the **path** we generated, and then continue to **recursively search** for the next point that meets the conditions. After this path is found, **add one to the score of all points on this path**. No extra points will be awarded if the path does not exist.

When **judging** whether a point meets the conditions, I take this method: if this point is the **first two points** of the path, directly determine that it meets the conditions, if this point and the latest two points in this path can be **composed a pair of similar triangles**, then it also qualifies, otherwise it does not.

2.4 greedy algorithm

First find **the largest element** in the first row of votingTable, the number of columns of this element must be less than the total number of columns in votingTable minus the total number of rows, then look for the largest element to the **lower right of it**, and traverse until the end.

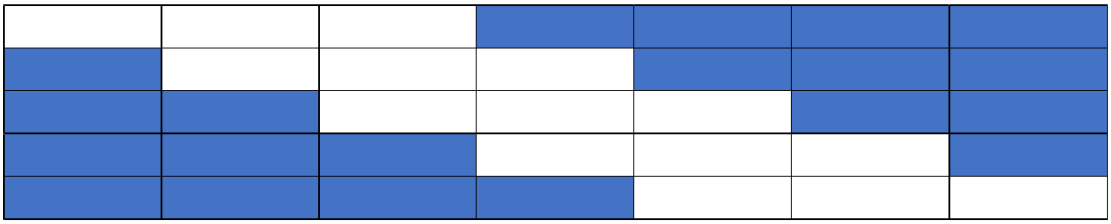


The above is the worst case. The maximum value in the first row is outside the boundary, so we have to take the right column as the starting point

2.5 Bonus——dynamic programming

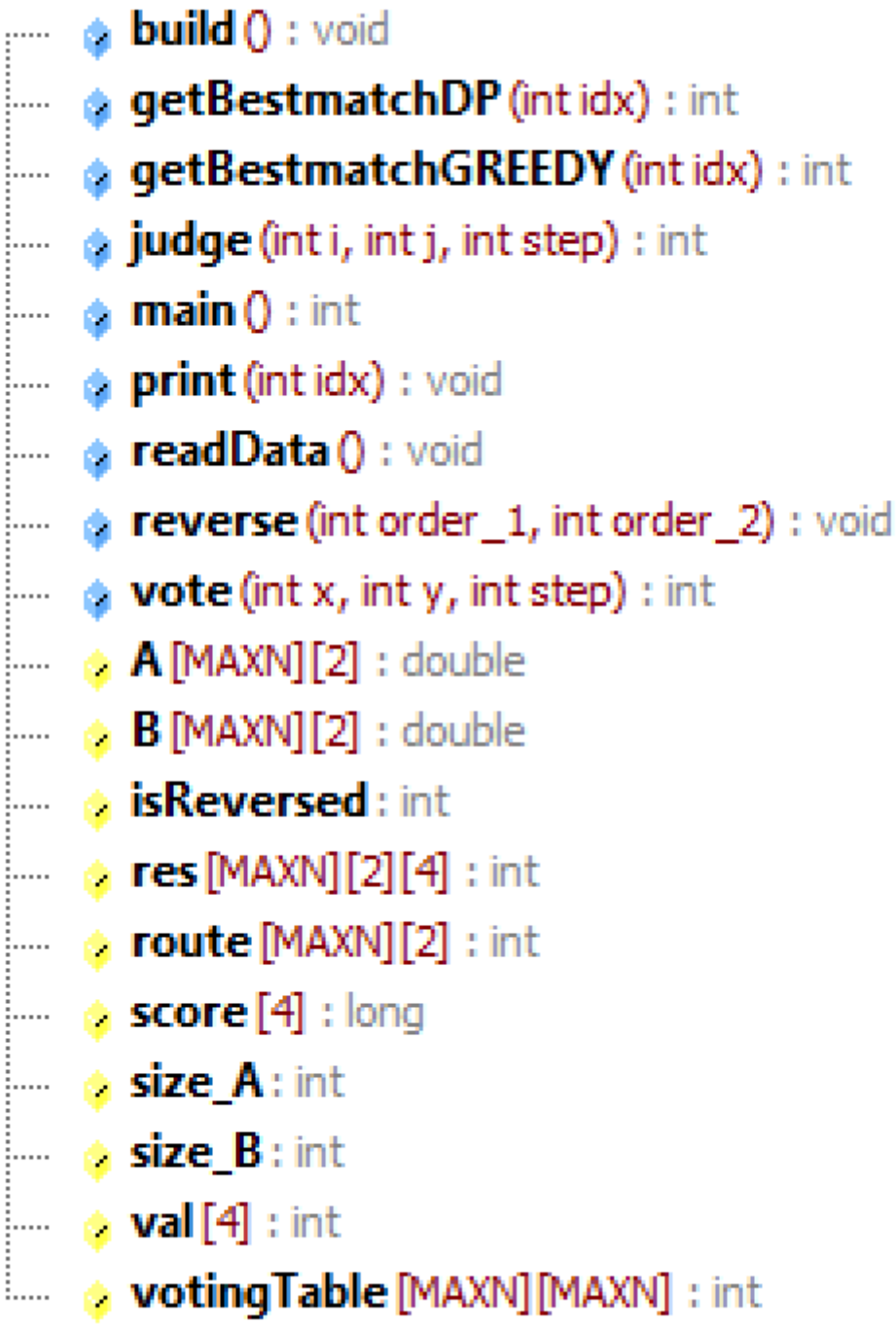
An array is declared to store the value of dynamic programming. Each element represents **the maximum number of votes for the point pair participating in the path** at the corresponding position of the votingTable. This value is calculated by the following formula:

$$dp[i][j] = max(dp[i][j-1], dp[i-1][j-1] + votingTable[i][j])$$



The blue part will never be used.

2.6 structure

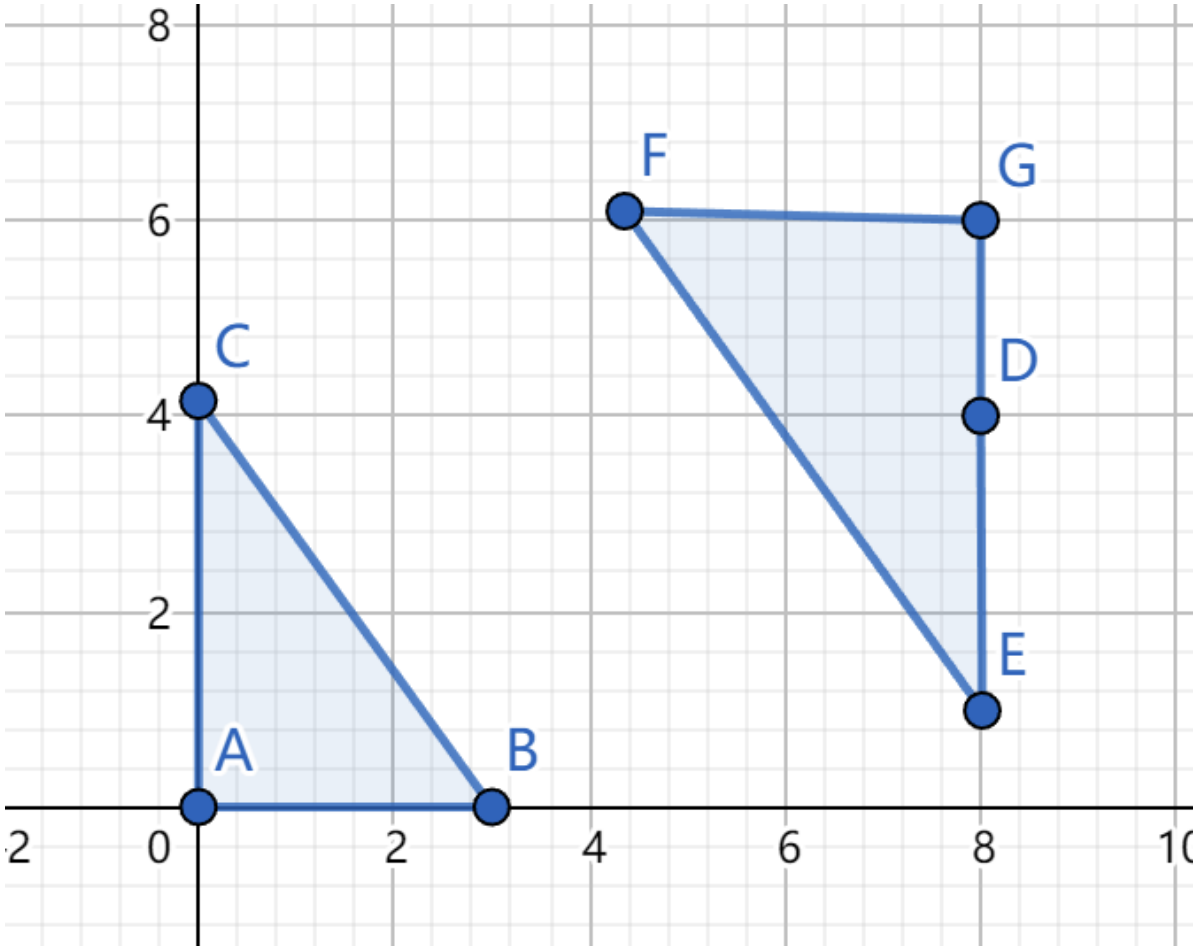


Chapter 3: Testing Results

3.1 TestCase1

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

case:



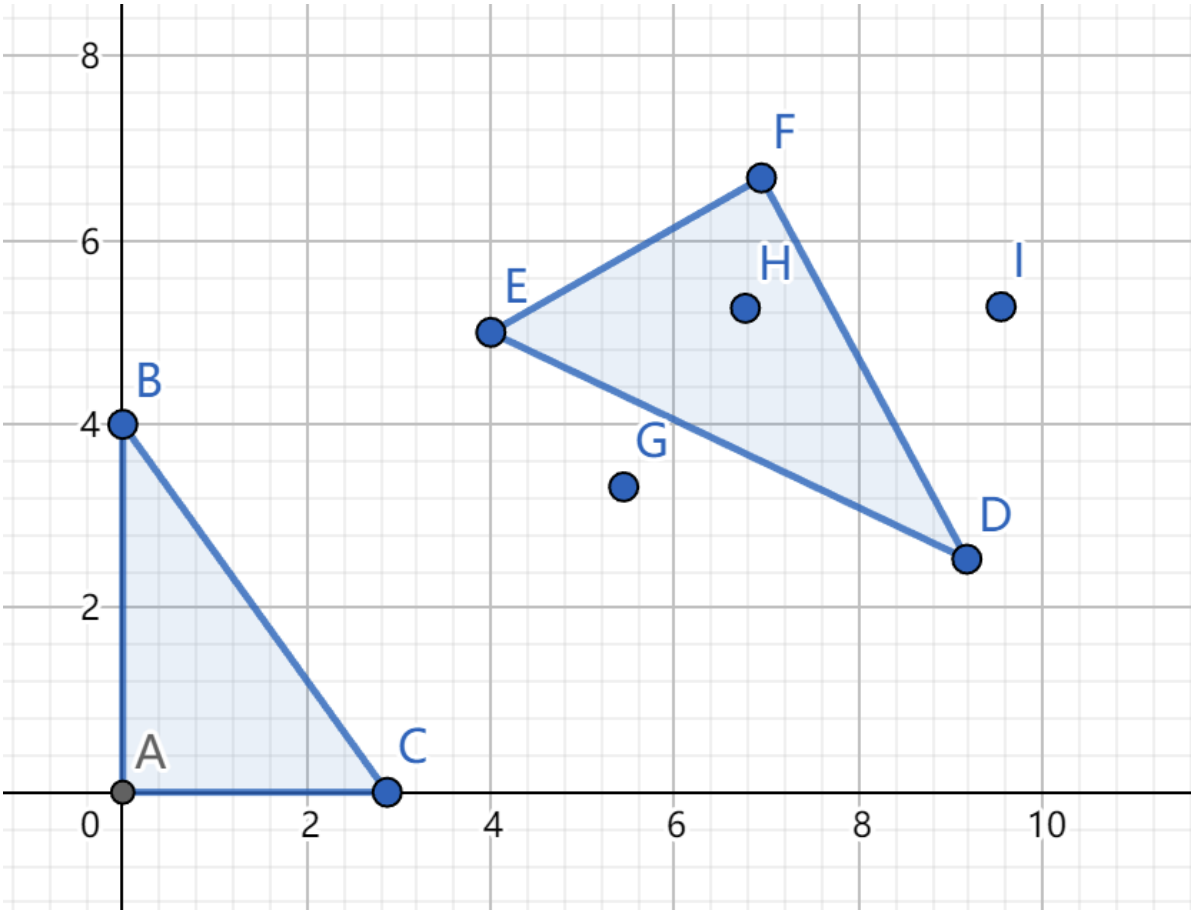
result:

```
Input the size of two shapes:
3 4
please input 3 coordinates, for each coordinates they have two values:
0 4
3 0
0 0
please input 4 coordinates, for each coordinates they have two values:
8 4
8 1
4.25 6
8 6
(1, 2)
(2, 3)
(3, 4)
```

3.2 TestCase2

```
3 6
0 4
3 0
0 0
5.3 3.2
9.1 2.6
4 5
7 6.7
6.8 5.1
9.3 5.1
```

case:



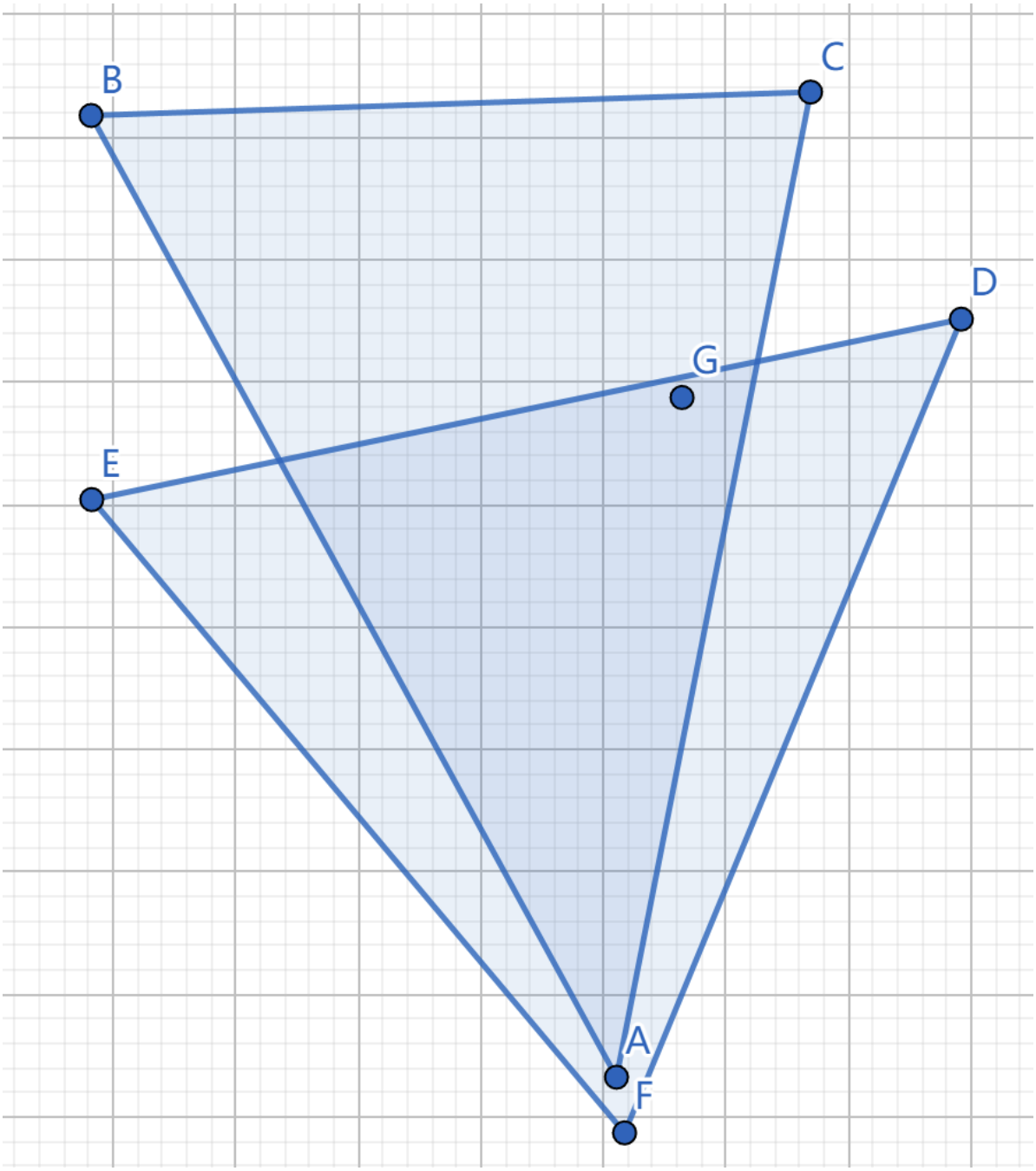
result:

```
Input the size of two shapes:
3 6
please input 3 coordinates, for each coordinates they have two values:
0 4
3 0
0 0
please input 6 coordinates, for each coordinates they have two values:
5.3 3.2
9.1 2.6
4 5
7 6.7
6.8 5.1
9.3 5.1
(1, 2)
(2, 3)
(3, 4)
```

3.3 TestCase3

```
4 3
12.3 2.4
3.7 18.3
15.4 18.7
13.2 13.7
17.9 15
3.7 12.1
12.3 1.7
```

case:



result:

```
Input the size of two shapes:
4 3
please input 4 coordinates, for each coordinates they have two values:
12.3 2.4
3.7 18.3
15.4 18.7
13.2 13.7
please input 3 coordinates, for each coordinates they have two values:
17.9 15
3.7 12.1
12.3 1.7
(1, 1)
(2, 2)
(3, 3)
```

Chapter 4: Analysis and Comments

4.1 The space complexity

Because we mostly use two-dimensional arrays, so the space complexity is **O(N^2)**.

4.2 The time complexity

4.2.1 The vote step

We walk through each pair of points, which requires $size_A * size_B$ calculations. For the case where one of the point pairs is taken as the starting point, we may prune it. Here we analyze the worst case, that is, in the absence of pruning, there are two cycles from each starting point, so the final complexity is **O(N^4)**.

4.2.2 The greedy algorithm

In the greedy algorithm, we traverse each row and find the maximum value in this row, so for each votingTable, we cycle twice, the time complexity is **O(N^2)**.

4.2.3 Bonus——dynamic programming

In this algorithm, we find the maximum value of the path weight by traversing each point pair, and find the path by backtracking. Two layers of loop nesting are used, so the time complexity is **O(N^2)**.

4.3 Summary

In summary, the space complexity of this program is **O(N^2)**, the time complexity of this program is **O(N^4)**.

Appendix: Source Code

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>

#define MAXN 105

//Store the coordinates of the two images,
//the first line is the x coordinate,
//the second line is the y coordinate
double A[MAXN][2], B[MAXN][2];
//Whether the record array has been reversed
int isReversed;
//Store the vote score for each peer pair
int votingTable[MAXN][MAXN];
//The size of the two images, where A must be larger than B
int size_A, size_B;
//Store the path traveled so far
int route[MAXN][2];
//Store the path where similar graphs are finally found
int res[MAXN][2][4];
//We obtained three results by traversing
//two graphs in order at the same time,
//traversing two graphs in reverse order at the same time,
```

```
//and traversing two graphs in order and one in reverse order.
//When obtaining the final path,
//the sum of the votes for these four paths is recorded
//and stored in this array.
long score[4];
//This array is used to store the number of points
//contained in each of the four paths
int val[4];

//Read in two arrays and make sure the small array is A
void readData();
//Traverse all point pairs in order,
//so that each point pair can be a head point pair
void build();
//Depth-first search, while doing pruning operations to get voting results
int vote(int x, int y, int step);
//Determine whether a point pair meets the requirements
int judge(int i, int j, int step);
//Choose the most suitable route from the voting results use greedy
algorithm
int getBestmatchGREEDY(int idx);
//Choose the most suitable route from the voting results use dynamic
programming algorithm
int getBestmatchDP(int idx);
//output the most suitable route
void print(int idx);
//Reverse the two arrays as needed
void reverse(int order_1, int order_2);

int main() {
//  define an ordinal array
    int idx[4] = {0, 1, 2, 3}, i, j, k;
//  Read data and make A array small
    readData();
//  Start the loop to get the voting results
    build();
//  Get the optimal path and record the result
    val[0] = getBestmatchGREEDY(0);
//  val[0] = getBestmatchDP(0);

//  for (i = 0; i < size_A; i++) {
//      for(j = 0; j < size_B; j++) {
//          printf("%d ", votingTable[i][j]);
//      }
//      printf("\n");
//  }
    for (i = 0; i < size_A; i++) {
        for(j = 0; j < size_B; j++) {
            votingTable[i][j] = 0;
        }
    }
//  Reverse array B so that array A is in order and B is in reverse order
    reverse(0,1);
//  Start the loop to get the voting results
    build();
```

```
//  Get the optimal path and record the result
    val[1] = getBestmatchGREEDY(1);
//  val[1] = getBestmatchDP(1);
    for (i = 0; i < size_A; i++) {
        for(j = 0; j < size_B; j++) {
            votingTable[i][j] = 0;
        }
    }

//  Reverse the A array so that the A array is in reverse order and B is in reverse order
    reverse(1,0);
//  Start the loop to get the voting results
    build();
//  Get the optimal path and record the result
    val[3] = getBestmatchGREEDY(3);
//  val[3] = getBestmatchDP(3);
    for (i = 0; i < size_A; i++) {
        for(j = 0; j < size_B; j++) {
            votingTable[i][j] = 0;
        }
    }

//  Reverse the B array so that the A array is in reverse order and B is in order
    reverse(0,1);
//  Start the loop to get the voting results
    build();
//  Get the optimal path and record the result
    val[2] = getBestmatchGREEDY(2);
//  val[2] = getBestmatchDP(2);

//  printf("%ld %ld %ld %ld\n", score[0], score[1], score[2], score[3] );
//  Find the path with the highest score
    for(i = 0; i < 3; i++) {
        for(j = 0; j < 3-i; j++) {
//          Swap the values of the ordinal array and the score array synchronously
            if(score[j] < score[j+1]) {
//              conversion score
                k = score[j];
                score[j] = score[j+1];
                score[j+1] = k;
//              Convert ordinal array
                k = idx[j];
                idx[j] = idx[j+1];
                idx[j+1] = k;
            }
        }
    }
//  printf("%ld %ld %ld %ld\n", idx[0], idx[1], idx[2], idx[3] );
//  printf("%d\n", idx[0]);
    print(idx[0]);
}

```

```
void readData() {
    int i;
    printf("Input the size of two shapes:\n");
    scanf("%d%d", &size_A, &size_B);

//  Enter the element values of A and B
    printf("please input %d coordinates, for each coordinates they have two values:\n", size_A);
    for(i=0; i<size_A; i++) {
        scanf("%lf%lf", &A[i][0], &A[i][1]);
    }
    printf("please input %d coordinates, for each coordinates they have two values:\n", size_B);
    for(i=0; i<size_B; i++) {
        scanf("%lf%lf", &B[i][0], &B[i][1]);
    }
//  Force A to be smaller than B
    if(size_A > size_B) {
        isReversed = 1;
        size_A = size_A ^ size_B;
        size_B = size_A ^ size_B;
        size_A = size_A ^ size_B;
//      swap two arrays
        for(i=0; i<size_B; i++) {
            double temp = A[i][0];
            A[i][0] = B[i][0];
            B[i][0] = temp;
            temp = A[i][1];
            A[i][1] = B[i][1];
            B[i][1] = temp;
        }
    }
//  printf("%d %d",size_A, size_B);

}

void build() {
    int i, j;
//  Loop through each pair of points in the two graphs as the starting point
    for (i = 0; i < size_A; i++) {
        for(j = 0; j < size_B; j++) {
//          record starting point
            route[0][0] = i;
            route[0][1] = j;
//          Start a depth-first search
            vote(i, j, 1);
        }
    }
}

int vote(int x, int y, int step) {
    int i, j;
    int isLoop = 0, result=0;
//  Find the next suitable point pair
```

```
    for(i = x + 1; i < size_A; i++) {
        for(j = y + 1; j < size_B; j++) {
//          printf("%d",step);
//          printf("%d\n", judge(i, j, step));
//          If this point is not suitable, continue enumerating
            if(!judge(i, j, step)) {
//              printf("jump\n");
                continue;
            } else {
//              printf("ok\n");
//              There are points found in the recording cycle that meet the requirements
                isLoop = 1;
//              record this pair
                route[step][0] = i;
                route[step][1] = j;
//              Step plus one, continue to search for the next set of point pairs
                result += vote(i, j, step + 1);
            }
        }
    }
//  make res equal to 1 if no pair is found
    if(!isLoop)result = 1;
//  If the number of steps is less than 2, it means that no polygon is formed, and returns 0
    if(!isLoop && step <= 2)result = 0;
//  Calculate voting results
    votingTable[x][y] += result;
    return result;
}

int judge(int i, int j, int step) {
//  The number of steps is less than 2 as true
    if(step<2) {
//      printf("skip\n");
        return 1;
    }
//  return 0;
//  printf("%d %d\n", i, j);
//  Calculate the three distances between the selected point in the A graph and the first two selected points
    double x12 = A[route[step-1][0]][0] - A[route[step-2][0]][0];
    double y12 = A[route[step-1][0]][1] - A[route[step-2][0]][1];
    double x01 = A[i][0] - A[route[step-1][0]][0];
    double y01 = A[i][1] - A[route[step-1][0]][1];
    double x02 = A[i][0] - A[route[step-2][0]][0];
    double y02 = A[i][1] - A[route[step-2][0]][1];
    double dis1 = x12 * x12 + y12 * y12;
    double dis2 = x01 * x01 + y01 * y01;
    double dis3 = x02 * x02 + y02 * y02;

//  printf("dis1: %lf\n",dis1);
//  printf("dis2: %lf\n",dis2);
//  printf("dis3: %lf\n",dis3);
```

```
232
233 //  Calculate the ratio between the three distances
234     double ratio1 = dis1/dis2;
235     double ratio2 = dis1/dis3;
236     double ratio3 = dis2/dis3;
237
238 //  Same as above, similar to the operation done in Figure A
239     x12 = B[route[step-1][1]][0] - B[route[step-2][1]][0];
240     y12 = B[route[step-1][1]][1] - B[route[step-2][1]][1];
241     x01 = B[j][0] - B[route[step-1][1]][0];
242     y01 = B[j][1] - B[route[step-1][1]][1];
243     x02 = B[j][0] - B[route[step-2][1]][0];
244     y02 = B[j][1] - B[route[step-2][1]][1];
245     double dis4 = x12 * x12 + y12 * y12;
246     double dis5 = x01 * x01 + y01 * y01;
247     double dis6 = x02 * x02 + y02 * y02;
248
249 //  printf("dis4: %lf\n",dis4);
250 //  printf("dis5: %lf\n",dis5);
251 //  printf("dis6: %lf\n",dis6);
252     double ratio4 = dis4/dis5;
253     double ratio5 = dis4/dis6;
254     double ratio6 = dis5/dis6;
255
256 //  Determine whether the three ratios in the two graphs are similar,
257 //  and return 0 if the difference is large
258     if(ratio1 / ratio4 <0.8 || ratio1 / ratio4 > 1.2
259             || ratio2 / ratio5 < 0.8 || ratio2 / ratio5 > 1.2
260             || ratio3 / ratio6 < 0.8 || ratio3 / ratio6 > 1.2)return 0;
261     else
262         return 1;
263 }
264
265 int getBestmatchGREEDY(int idx) {
266 //  flag records the ordinate, i records the abscissa,
267 //  and j is used to find the appropriate value of the ordinate
268     int flag=0, cnt=0, i, j;
269 //  iterate over the entire array
270     for(i = 0; i < size_A; i++) {
271         for(j = flag; j < size_B; j ++) {
272 //          Find the maximum value that matches the condition in this row
273             if(votingTable[i][j] > votingTable[i][flag]) {
274                 if((cnt == 1&&j >= size_B - 1)||(!cnt && j >= size_B -
    2))continue;
275                 flag = j;
276             }
277         }
278 //      Recorded in the res array according to the serial number
279 //      Add one to flag and cnt respectively
280         if(cnt < size_A && flag < size_B) {
281 //          record score
282             score[idx] += votingTable[i][flag];
283             if(idx == 0) {
284                 res[cnt][0][idx] = i + 1;
285                 res[cnt][1][idx] = ++flag;
```

```
            } else if(idx == 1) {
                res[cnt][0][idx] = i + 1;
                res[cnt][1][idx] = size_B - flag;
                flag++;
            } else if(idx == 2) {
                res[cnt][0][idx] = size_A - i;
                res[cnt][1][idx] = ++flag;
            } else if(idx == 3) {
                res[cnt][0][idx] = size_A - i;
                res[cnt][1][idx] = size_B - flag;
                flag++;
            }
            cnt++;
        }
    }
//  Determine whether a polygon can be formed
    if(cnt > 2)return cnt;
    else return 0;
}

int getBestmatchDP(int idx) {
//  Initialize the dynamic programming array
    int dp[size_A][size_B], i, j;
//  The first row and first column of the dp array are the same as the voting result
    for(i = 0; i <size_B; i++) {
        dp[0][i] = votingTable[0][i];
    }
    for(i = 0; i <size_A; i++) {
        dp[i][0] = votingTable[i][0];
    }

    for(i = 1; i < size_A; i++) {
        for(j = 1; j < size_B; j++) {
//          Go forward if one step forward is better than searching to the right
            if(dp[i][j-1] > dp[i-1][j-1] + votingTable[i][j]) dp[i][j] = dp[i][j-1];
            else dp[i][j] = dp[i-1][j-1] + votingTable[i][j];
        }
    }
//  flag represents the current position of the column number
    int flag = size_B;
    for(i = size_A - 1; i >= 0; i--) {
        for(j = flag-1; j > i - 1; j--) {
//          If you find the edge of the maximum, record it
            if(dp[i][j] > dp[i][j-1] || j == i) {
                flag = j;
//              Record results in different orders
                if(idx == 0) {
                    res[i][0][idx] = i + 1;
                    res[i][1][idx] = j + 1;
                } else if(idx == 1) {
                    res[i][0][idx] = i + 1;
                    res[i][1][idx] = size_B - j;
```

```
                } else if(idx == 2) {
                    res[i][0][idx] = size_A - i;
                    res[i][1][idx] = j + 1;
                } else if(idx == 3) {
                    res[i][0][idx] = size_A - i;
                    res[i][1][idx] = size_B - j;
                }
//              End the traversal of this line if one is found in a line
                break;
            }
        }
    }
    return size_A;

}

//The first parameter represents whether to reverse the order of the A array,
//and the second represents B
void reverse(int order_1, int order_2) {
    int i;
    if(order_1) {
//      Change the front to the back
        for(i=0; i < size_A / 2; i++) {
            double tx = A[i][0];
            A[i][0] = A[size_A-1-i][0];
            A[size_A-1-i][0] = tx;
            double ty = A[i][1];
            A[i][1] = A[size_A-1-i][1];
            A[size_A-1-i][1] = ty;
        }
    }
    if(order_2) {
//      Change the front to the back
        for(i=0; i < size_B / 2; i++) {
            double tx = B[i][0];
            B[i][0] = B[size_B-1-i][0];
            B[size_B-1-i][0] = tx;
            double ty = B[i][1];
            B[i][1] = B[size_B-1-i][1];
            B[size_B-1-i][1] = ty;
        }
    }
}

void print(int idx) {
    int i;
//  If val is 0, there is no matching image
    if(!val[idx]) printf("There is no result!\n");
    else {
        for(i=0; i<val[idx]; i++) {
//          If the isReversed value is 0, it means that the A and B arrays are not converted
            if(!isReversed)printf("(%d, %d)\n", res[i][0][idx], res[i][1][idx]);
```

```
            else printf("(%d, %d)\n", res[i][1][idx], res[i][0][idx]);
        }
    }
}
```

Declaration

I hereby declare that all the work done in this project titled "Voting Tree" is of my independent effort.