# Introduction

Here is part of the introduction from PTA:

> 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 project, you are supposed to solve the aforementioned problem with a combination of various data structures including arrays and voting trees. The basic idea is to build trees, where each pair of points in A and B can form a treenode. Starting from a node (as a root node), you can expand a tree by adding another tree node conditional on that the expanded child node contributes to a valid match between A and B. For example, if you start from ($m_1$, $n_1$) and extend it with ($m_2$,$n_2$), then the following conditions must hold: $m_1 \neq m_2$, $n_1 \neq n_2$, and more importantly the match should be monotonic, that is, the list of indices of { $m_1$,$m_2$,$m_3$, $\cdots$ } should be monotonically increasing or decreasing, so is true with { $n_1$,$n_2$,$n_3$,$\cdots$ }. All these extensions lead to a set of trees within which each tree path represents a potential match between A and B. Then your task is to let these paths **vote** for the confidences of possible matches.
>
> Obviously, if we simply aggregate all valid paths, the search will quickly grow to an exponential complexity. Thus, an efficient expanding strategy is required to stop tree expansion if some requirements are violated.
>
> After the tree expansion, we start the voting process. Make sure that your best solution does not violate the monotonicity property of the match.

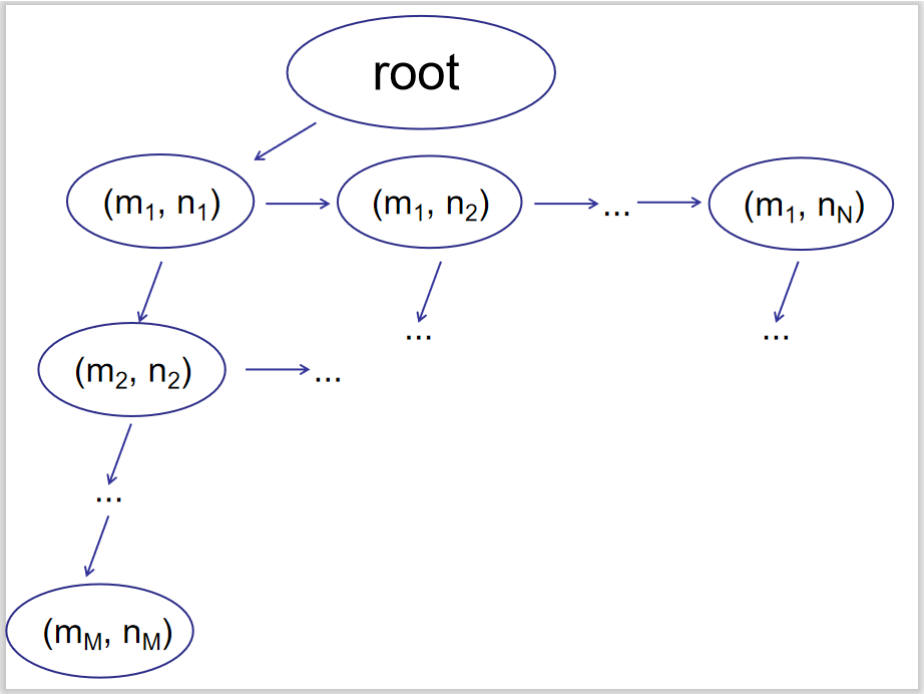
It's to tell whether two shapes are similar to a certain degree. Given that the introduction is too vague to achieve a specific program, so I give some specifications based on my **own** comprehension of the introduction from PTA :

1. We need to find the optimal correspondence until all the points in the shape with less points are matched, for I think to find the 'optimal correspondence' between several points in A and several points in B is somehow meaningless;
2. As a specification of 'optimal', the greatest difference of the square of the lengths of the corresponding line segment allowed is 5% and the greatest difference of the corresponding angle allowed is 3% (which may be too strict but is adjustable);
3. If there is no considerable match, the program will output "No Considerable Match!". Otherwise, it will output the best match;

# Algorithm Specification

## Summary

To conveniently solve the problem, we can use a tree to traverse all the possibilities, which is illustrated as a simple graph as follow:




The treenodes in the same level are different possible combinations, which can be seen as cousins and are constricted by a common 'father'.

The definitions of each node in the tree are as follow:

```
typedef struct tnode{
    int Aindex;
    int Bindex;
    struct tnode *father; // set for voting conveniently
    struct tnode *left; // point to children
    struct tnode *right; // parallel combination, cousins
    int mono;           // 0 represents increase, 1 represents decrease
} tnode;                // the node in the tree
```

- 'Aindex' is the index of the points in shape A, which is always of less points. We can achieve that easily by switching the position of the parameters, which will be illustrated later.
- 'Bindex' is the index of the points in shape B, which is always of more points. If M $\leq$ N then shape B has N points; otherwise, shape B has M points.
- 'father' is a pointer pointing to a treenode that the treenodes of the same level are originated from and constricted by. This pointer is set for voting, which is convenient for the vote process.
- 'left' is a pointer pointing to the children and 'right' is a pointer pointing to the cousins.
- 'mono' is to signal the monotonicality. If it is 0, it means that within a path, the indices of points in B should be monotonically increasing. If it is 1, it means that within a path, the indices of points in B should be monotonically decreasing.

So, to solve the problem, using the tree,  the steps should be:

1. Get the input;
2. Build the tree and vote;
3. Output the result;

```
int main()
{
    int M, N;
    scanf ("%d %d", &M, &N);    // get the input
```

```
    getInput(M, N);

    for (int i=0; i<100; i++)   // clear the vote table
    {
        for (int j=0; j<100; j++)
        {
            table[i][j]=0;
        }
    }

    if (M<=N) // let the shape A be the one with the less points
    {
        buildTree(Marr, 0, M-1, Narr, N-1, root);
        output(M, N, 1);
    }
    else
    {
        buildTree(Narr, 0, N-1, Marr, M-1, root);
        output(N, M, 0);
    }

    return 0;
}
```

# Get the Input

It is easy and so it is omitted.

# Build the Tree and Vote

This step is the most difficult to achieve.

First of all, we need to generate the tree by recursion. For convenience, we can always take the array containing the coordinates of a shorter length as the array of shape A('Aarr'). 'Ast' signals the index that the function should start from. 'Aend' and 'Bend' signal the last index. 'sroot' represents the subroot.

```
void buildTree(point *Aarr, int Ast, int Aend, point *Barr, int Bend, tnode
*sroot);
void buildNode(tnode *ptr, int Ast, int i, tnode *sroot);
int judge(tnode *curnode, int Ain, int Bin, point *Aarr, point *Barr);
```

Then, we cannot just generate a treenode for every combination, which is tough for the memory. So a function judging whether to generate the next treenode is needed. For every treenode to be generated, we need to check its monotonicality and its correspondence. To check correspondence, the judge function calculate the lengths and the angles of the line segments formed by the last three points in A and B. If the square of lengths differ within 5% and the angles differ within 3%, we think we have found a 'optimal correspondence'. At the same time, we should check the monotonicality of the indices.

In pseudo-code for a general situation:

```
void buildTree(point *Aarr, int Ast, int Aend, point *Barr, int Bend, tnode
*sroot)
{
    for(int i=0; i<=Bend; i++)
    {
        if(i<sroot->Bindex && sroot->mono==1 && judge())
        {
            buildNode(ptr, Ast, i, sroot);
            buildTree(Aarr, Ast+1, Aend, Barr, Bend, ptr);
        }
    }
    if(srootIsALeaf)
    {
        vote(sroot);
    }
}
```

Then for the vote function, we should make use of the 'father' pointer. To make the results of voting more beautiful, I simply give the longer path a greater weight, which can be replaced by increasing by 1. Considering the generation of the tree, the program will call the vote function in the `buildTree()` function. In pseudo-code:

```
void vote(tnode *leaf)
{
    tnode *ptr=leaf;
    int weight=0;
    while(ptr!=root)
    {
        weight++;
        ptr=ptr->father;
    }
    while(leaf!=root)
    {
        table[leaf->Aindex][leaf->Bindex]+=weight;
        leaf=leaf->father;
    }
}
```

## Output the Result

It is relatively simple, we just need to mind:

1. If the voting table is all 0, which means there is no optimal correspondence, we should output "No considerable match!";
2. If the voting table is all the same(but not 0), which means shape A and shape B are congruent, we just simply give a feasible correspondence;
3. For a general situation, keep an array to store the index that has been matched to avoid repeating.

# Testing Results

# Input 1

The sample given by PTA

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

Desired output:

```
(1, 2)
(2, 3)
(3, 4)
```

Actual output:

```
Best match:
(1, 2)
(2, 3)
(3, 4)
```

Pass

# Input 2

The two shape are approximately congruent and the indices of B should be output in decreasing order. Also, test the cases with 3 points(the least number).

```
3 3
0 0
0 40
30 0
29 0
0 -40
0 0
```

Desired output:

```
(1, 3)
(2, 2)
(3, 1)
```

Actual output:

```
Best Match:
(1, 3)
(2, 2)
(3, 1)
```

Pass

# Input 3

The two shape are congruent.

```
4 4
0 0
0 1
1 1
1 0
0 1
1 1
1 0
0 0
```

Desired output:

```
(1, 1)
(2, 2)
(3, 3)
(4, 4)
```

Actual output:

```
Best Match:
(1, 1)
(2, 2)
(3, 3)
(4, 4)
```

Pass

# Input 4

There is no best match.

```
3 3
-1 2
2 0
0 0
0 0
1 1
1 0
```

Desired output:

```
No Considerable Match!
```

Actual output:

```
No Considerable Match!
```

Pass

# Input 5

A general case between a triangle and a rectangle.

```
3 4
0 0
0 1
1 0
0 0
0 1
1 1
1 0
```

Desired output:

```
(1, 1)
(2, 2)
(3, 4)
```

Actual output:

```
Best Match:
(1, 1)
(2, 2)
(3, 4)
```

Pass

# Input 6

A more complex case.

```
6 6
-100 0
-50 0
0 -50
-75 -75
-120 -50
-120 -25
100 0
50 0
1 49
75 75
121 51
120 25
```

Desired output:

```
(1, 2)
(2, 1)
(3, 3)
(4, 4)
(5, 5)
(6, 6)
```

Actual output:

```
Best Match:
(1, 2)
(2, 1)
(3, 3)
(4, 4)
(5, 5)
(6, 6)
```

Pass

# Analysis and Comments

## Analysis

Given the presence of the judge function, the time complexities and the space complexities of `main()` and `buildTree()` is somehow too difficult to analyze. So I just analyze other functions.

- `buildNode()`

  It has four assignments no matter what the input is.

  time complexity: $O(1)$

  space complexity: $O(1)$

- `judge()`

It has a certain number of assignments and a certain number of calculation no matter what the input is.

time complexity: $O(1)$

space complexity: $O(1)$

- `vote()`

The input can be taken as a linked list with N nodes, and the function needs to traverse the whole linked list twice.

time complexity: $O(N)$

space complexity: $O(1)$

- `output()`

Assume that there are M rows and N columns. For each row, the function needs to traverse the whole row to find the optimal match. Also, it needs to traverse all the rows. So the time complexity is $O(M * N)$. It needs no extra space.

time complexity: $O(M * N)$

space complexity: $O(1)$

## Comments

As far as I can see, combining the comments from the peer review, there are three possible improvements:

1. Modifying the judge function. We can adjust the allowed range of difference to make the correspondence more reasonable. Also, we can add some other criterion to make it better.
2. Adjusting the condition for voting. When testing the program, I found that if the vote function is called when the level is just greater than 3 and if the points is relatively too many, the votes of the lower levels may be dramatically larger than the votes of the higher level, which makes the result not so beautiful.
   So, we can give them weights, which is added in my program, or give a smallest length defined by a rate. For example, only if the length of a path is grater than 30% of the longest, we call the vote function to vote. In this way, we can make the result of the voting table more beautiful even if there are many points. But in my program, I just simply call the vote function when the length is greater then 3, because the samples are simple and there is no need to bother to set a rate.
3. The checking for repeat in the output function is too simple, which is not able to deal with the shape containing more than one similar shapes, for example:

   ```
   3 5
   0 0
   0 6
   8 0
   0 0
   0 3
   0 6
   4 3
   8 0
   ```

   To solve this, I think there are two ways. One is to simply give an extra weight when the shapes formed by the picked points are congruent. The other one is to simply create an array to store the points that could form two congruent shapes, and the output function should display the points according to the array.

# Program

The code of the program is as follow, which can also be obtained in voting_tree.cpp:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
typedef struct point{
    float x;
    float y;
} point;                // the coordinates of each point
typedef struct tnode{
    int Aindex;
```

```
    int Bindex;
    struct tnode *father; // set for voting conveniently
    struct tnode *left; // point to children
    struct tnode *right; // parallel combination, cousins
    int mono;           // 0 represents increase, 1 represents decrease
} tnode;                // the node in the tree

tnode *root; // a dummy root
point Marr[100], Narr[100]; // store the input
int table[100][100]; // the vote table

void buildTree(point *Aarr, int Ast, int Aend, point *Barr, int Bend, tnode
*sroot);
void buildNode(tnode *ptr, int Ast, int i, tnode *sroot);
int judge(tnode *curnode, int Ain, int Bin, point *Aarr, point *Barr);
void vote(tnode *leaf);
void output(int A, int B, int flag);

int main()
{
    int M, N;
    scanf ("%d %d", &M, &N);    // get the input
    for (int i=0; i<M; i++)
    {
        scanf("%f %f", &Marr[i].x, &Marr[i].y);
    }
    for (int i=0; i<N; i++)
    {
        scanf("%f %f", &Narr[i].x, &Narr[i].y);
    }

    for (int i=0; i<100; i++)   // clear the vote table
    {
        for (int j=0; j<100; j++)
        {
            table[i][j]=0;
        }
    }

    root=(struct tnode*)malloc(sizeof(struct tnode));
    root->left=root->right=NULL;
    if (M<=N) // let the shape A be the one with the less points
    {
        buildTree(Marr, 0, M-1, Narr, N-1, root);
        output(M, N, 1);
    }
    else
    {
        buildTree(Narr, 0, N-1, Marr, M-1, root);
        output(N, M, 0);
    }

    return 0;
}

void buildTree(point *Aarr, int Ast, int Aend, point *Barr, int Bend, tnode
*sroot)
{
```

```
    tnode *ptr;
    if (Ast==0) // level 2, in which the indices of all points of A is 0(Ast)
    {
        for (int i=0; i<=Bend; i++)
        {
            if (sroot->left==NULL)  // the subroot does not have a child yet
            {
                sroot->left=ptr=(struct tnode*)malloc(sizeof(struct tnode));
                buildNode(ptr, Ast, i, sroot);
            }
            else
            {
                ptr->right=(struct tnode*)malloc(sizeof(struct tnode));
                ptr=ptr->right;
                buildNode(ptr, Ast, i, sroot);
            }
            buildTree(Aarr, 1, Aend, Barr, Bend, ptr);
        }
    }
    else if(Ast==1) // level 3, in which the indices of all points of A is
1(Ast)
    {
        for (int i=0; i<=Bend; i++)
        {
            if (sroot->left==NULL && i!=sroot->Bindex)  // the subroot does not
have a child and the point cannot repeat
            {
                sroot->left=ptr=(struct tnode*)malloc(sizeof(struct tnode));
                buildNode(ptr, Ast, i, sroot);
                if(ptr->Bindex<ptr->father->Bindex) // set the monotonicality
                {
                    ptr->mono=1;
                }
                else
                {
                    ptr->mono=0;
                }
                buildTree(Aarr, 2, Aend, Barr, Bend, ptr);
            }
            else if(i!=sroot->Bindex)
            {
                ptr->right=(struct tnode*)malloc(sizeof(struct tnode));
                ptr=ptr->right;
                buildNode(ptr, Ast, i, sroot);
                if(ptr->Bindex<ptr->father->Bindex)
                {
                    ptr->mono=1;
                }
                else
                {
                    ptr->mono=0;
                }
                buildTree(Aarr, 2, Aend, Barr, Bend, ptr);
            }
        }
    }
    else if(Ast==Aend)  // the last level, in which all the nodes does not have
a child
```

```
    {
        for (int i=0; i<=Bend; i++)
        {
            if (sroot->left==NULL && i!=sroot->Bindex)
            {
                if(i<sroot->Bindex && sroot->mono==1 && judge(sroot, sroot-
>Aindex+1, i, Aarr, Barr))   // if sufficient, build the next node and
vote(decrease)
                {
                    sroot->left=ptr=(struct tnode*)malloc(sizeof(struct tnode));
                    buildNode(ptr, Ast, i, sroot);
                    vote(ptr);
                }
                else if (i>sroot->Bindex && sroot->mono==0 && judge(sroot,
sroot->Aindex+1, i, Aarr, Barr)) // if sufficient, build the next node and
vote(increase)
                {
                    sroot->left=ptr=(struct tnode*)malloc(sizeof(struct tnode));
                    buildNode(ptr, Ast, i, sroot);
                    vote(ptr);
                }
            }
            else if(i!=sroot->Bindex)
            {
                if(i<sroot->Bindex && sroot->mono==1 && judge(sroot, sroot-
>Aindex+1, i, Aarr, Barr))   // if sufficient, build the next node and
vote(decrease)
                {
                    ptr->right=(struct tnode*)malloc(sizeof(struct tnode));
                    ptr=ptr->right;
                    buildNode(ptr, Ast, i, sroot);
                    vote(ptr);
                }
                else if (i>sroot->Bindex && sroot->mono==0 && judge(sroot,
sroot->Aindex+1, i, Aarr, Barr)) // if sufficient, build the next node and
vote(increase)
                {
                    ptr->right=(struct tnode*)malloc(sizeof(struct tnode));
                    ptr=ptr->right;
                    buildNode(ptr, Ast, i, sroot);
                    vote(ptr);
                }
            }
        }
    }
    else  // other levels
    {
        for (int i=0; i<=Bend; i++)
        {
            if (sroot->left==NULL && i!=sroot->Bindex)
            {
                if(i<sroot->Bindex && sroot->mono==1 && judge(sroot, sroot-
>Aindex+1, i, Aarr, Barr))
                {
                    sroot->left=ptr=(struct tnode*)malloc(sizeof(struct tnode));
                    buildNode(ptr, Ast, i, sroot);
                    ptr->mono=1;
                    buildTree(Aarr, Ast+1, Aend, Barr, Bend, ptr);
```

```
                }
                else if (i>sroot->Bindex && sroot->mono==0 && judge(sroot,
sroot->Aindex+1, i, Aarr, Barr))
                {
                    sroot->left=ptr=(struct tnode*)malloc(sizeof(struct tnode));
                    buildNode(ptr, Ast, i, sroot);
                    ptr->mono=0;
                    buildTree(Aarr, Ast+1, Aend, Barr, Bend, ptr);

                }
            }
            else if(i!=sroot->Bindex)
            {
                if(i<sroot->Bindex && sroot->mono==1 && judge(sroot, sroot-
>Aindex+1, i, Aarr, Barr))
                {
                    ptr->right=(struct tnode*)malloc(sizeof(struct tnode));
                    ptr=ptr->right;
                    buildNode(ptr, Ast, i, sroot);
                    ptr->mono=1;
                    buildTree(Aarr, Ast+1, Aend, Barr, Bend, ptr);
                }
                else if (i>sroot->Bindex && sroot->mono==0 && judge(sroot,
sroot->Aindex+1, i, Aarr, Barr))
                {
                    ptr->right=(struct tnode*)malloc(sizeof(struct tnode));
                    ptr=ptr->right;
                    buildNode(ptr, Ast, i, sroot);
                    ptr->mono=0;
                    buildTree(Aarr, Ast+1, Aend, Barr, Bend, ptr);

                }
            }
        }
        if (sroot->left==NULL && Ast>=2) // if a node has no children, then the
path ends and begin to vote; the latter condition is to limit the votes of the
former combination
        {
            vote(sroot);
        }
    }
}

void buildNode(tnode *ptr, int Ast, int i, tnode *sroot) // to build a node for
a combination
{
    ptr->Aindex=Ast;
    ptr->Bindex=i;
    ptr->father=sroot;
    ptr->left=ptr->right=NULL;
}

int judge(tnode *curnode, int Ain, int Bin, point *Aarr, point *Barr)   // to
judge whether to build a new node
{
    float Ax1=Aarr[curnode->father->Aindex].x, Ay1=Aarr[curnode->father-
>Aindex].y, Bx1=Barr[curnode->father->Bindex].x, By1=Barr[curnode->father-
>Bindex].y;
```

```
    float Ax2=Aarr[curnode->Aindex].x, Ay2=Aarr[curnode->Aindex].y,
Bx2=Barr[curnode->Bindex].x, By2=Barr[curnode->Bindex].y;
    float Ax3=Aarr[Ain].x, Ay3=Aarr[Ain].y, Bx3=Barr[Bin].x, By3=Barr[Bin].y;

    float Aleng1=((Ax1-Ax2)*(Ax1-Ax2)+(Ay1-Ay2)*(Ay1-Ay2)); // the length of the
line segment in A
    float Aleng2=((Ax2-Ax3)*(Ax2-Ax3)+(Ay2-Ay3)*(Ay2-Ay3));
    float Bleng1=((Bx1-Bx2)*(Bx1-Bx2)+(By1-By2)*(By1-By2)); // the length of the
line segment in B
    float Bleng2=((Bx2-Bx3)*(Bx2-Bx3)+(By2-By3)*(By2-By3));
    float Aangle=((Ax1-Ax2)*(Ax3-Ax2)+(Ay1-Ay2)*(Ay3-
Ay2))/(sqrt(Aleng1)*sqrt(Aleng2));
    float Bangle=((Bx1-Bx2)*(Bx3-Bx2)+(By1-By2)*(By3-
By2))/(sqrt(Bleng1)*sqrt(Bleng2));
    Aangle=acos(Aangle); // the angle of the points in A
    Bangle=acos(Bangle); // the angle of the points in B

    return (fabs(Aleng1/Aleng2 - Bleng1/Bleng2) < 0.05*(Aleng1/Aleng2))&&
(fabs(Aangle-Bangle) < 0.03*Aangle); // check whether the difference is allowed
}

void vote(tnode *leaf)
{
    tnode *ptr=leaf;
    int ct=0;        // name 'ct' as 'weight' may be better
    while(ptr!=root) // the combinations in a longer path have a greater weight,
because the combinations are more credible
    {
        ct++;
        ptr=ptr->father;
    }
    while(leaf!=root)// vote reversely with the 'father' pointer
    {
        table[leaf->Aindex][leaf->Bindex]+=ct;
        leaf=leaf->father;
    }
}

void output(int A, int B, int flag)
{
    int Bo, zeroflag=0, eflag=1, trash[100];
    for (int i=0; i<A; i++) //check whether the votes of all the combiantions
are 0
    {
        for (int j=0; j<B; j++)
        {
            zeroflag=zeroflag+table[i][j];
        }
    }

    if (!zeroflag)
    {
        printf("No Considerable Match!");
    }
    else
    {
        for(int i=0; i<A; i++)  // check whether the two shapes are congruent
        {
```

```
            for (int j=1; j<B; j++)
            {
                if (table[i][j-1]!=table[i][j])
                {
                    eflag=0;
                }
            }
        }
        if (eflag)
        {
            printf("Best Match:\n");
            for (int i=1; i<=A; i++)
            {
                printf("(%d, %d)\n", i, i);
            }
        }
        else
        {
            printf("Best Match:\n");
            for (int i=0; i<A; i++)
            {
                Bo=0;
                for(int j=0; j<B; j++)
                {
                    // printf("%d ", table[i][j]);
                    int k;
                    for (k=0; k<i; k++)
                    {
                        if (trash[k]==j && Bo==j) // abandon the repeated
indices
                        {
                            Bo++;
                            break;
                        }
                    }
                    if (k==i && table[i][j]>table[i][Bo])   // renew the index
of B to be output
                    {
                        Bo=j;
                    }
                }
                trash[i]=Bo; // store the indices of B that have been used
                //putchar('\n');
                if (flag) // put A, B and M, N in accordance
                {
                    printf("(%d, %d)\n", i+1, Bo+1);
                }
                else
                {
                    printf("(%d, %d)\n", Bo+1, i+1);
                }
            }
        }
    }
}
```

**Declaration**

***I hereby declare that all the work done in this project titled "Project 1 Voting Tree Report" is of my independent effort.***