# Appendix

## 1 路径树 | Directory Tree

```
$ tree
.
├── Readme.md
├── _auto.sh
├── _clear.sh
└── src
    ├── _clear.sh
    ├── _compile.sh
    ├── _run.sh
    ├── judger
    │   ├── _clear.sh
    │   ├── _judge.sh
    │   ├── _setup.sh
    │   ├── cache
    │   ├── judger.cpp
    │   ├── painter
    │   │   ├── painter.py
    │   │   └── requirements.txt
    │   ├── section_A
    │   │   ├── Readme.md
    │   │   └── maker.cpp
    │   ├── section_B
    │   │   ├── Readme.md
    │   │   └── maker.cpp
    │   └── section_C
    │       ├── Readme.md
    │       └── maker.cpp
    ├── lib
    │   ├── meta_data.cpp
    │   └── meta_data.h
    └── solver
        ├── _clear.sh
        ├── _setup.sh
        ├── main.cpp
        └── optimal_match
            ├── match_referee.cpp
            ├── match_referee.h
            ├── voting_tree.cpp
            └── voting_tree.h
```

- 只有与问题解决有关的代码会被放在这里，如果您想了解其他内容，请自行查看对应的文件。
- Only files related with the solver will be put here. Check others yourself if you want.

## 2 lib

- `meta_data.h`

```cpp
#ifndef __META_DATA__
#define __META_DATA__

#include <cmath>
#include <vector>
#include <iostream>

/*
 * <[ Class Defination ]>
 * [ Class Name ]:
 * - Point2D
 * [ Description ]:
 * - The class defines a meta data of a point with two double number that
 * - represent coordinates.
 * [ Usage ]:
 * - You can initialize the point with the coordinates and use it as a meta
 * - data. And you can set the mark on the point (eg: You can set the mark
 * - as the UUID of the point, the value of the point, etc.) and get it after
 * - check.
 */
class Point2D{
private:

    bool isMarked;
    int mark;
    double x, y;

public:

    Point2D() = default;

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
```

```
     * - C'tor
     * [ Belonging Class ]:
     * - Point2D
     * [ Description ]:
     * - To initialize the points object.
     * [ Usage ]:
     * - "Point2D p(1.0, 2.0);" defines a plane points at (1.0,
2.0).
     */
    Point2D(double, double);

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - Point2D::getX()
     * [ Belonging Class ]:
     * - Point2D
     * [ Description ]:
     * - Get x coordinate value.
     * - [ Params Description ]:
     * - - No params.
     * - [ Return Description ]:
     * - - (double) // The value of x coordinate value of the
point.
     * [ Usage ]:
     * - "p.getX()" gets a right value represents the x
coordinate value
     * - of Point2D object p.
     */
    double getX();

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - Point2D::getY()
     * [ Belonging Class ]:
     * - Point2D
     * [ Description ]:
     * - Get y coordinate value.
     * - [ Params Description ]:
     * - - No params.
     * - [ Return Description ]:
     * - - (double) // The value of y coordinate value of the
point.
```

```
     * [ Usage ]:
     * - "p.getY()" gets a right value represents the y
coordinate value
     * - of Point2D object p.
     */
    double getY();

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - Point2D::getDis()
     * [ Belonging Class ]:
     * - Point2D
     * [ Description ]:
     * - Get the distance between this and that.
     * - [ Params Description ]:
     * - - No params.
     * - [ Return Description ]:
     * - - (double) // The distance between this and that.
     * [ Usage ]:
     * - "p1.getDis(p2)" gets a right value represents the
distance between
     * - p1 and p2.
     */
    double getDis(Point2D &);

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - Point2D::setMark()
     * [ Belonging Class ]:
     * - Point2D
     * [ Description ]:
     * - Set a mark on the point.
     * - [ Params Description ]:
     * - - [1] (int) // The mark to be set.
     * - [ Return Description ]:
     * - - No retrun.
     * [ Usage ]:
     * - "p1.setMark(1)" sets the mark on p1 is 1.
     */
    void setMark(int);

    /*
```

```
117      * <[ Class Methods Defination ]>
118      * [ Method Name ]:
119      * - Point2D::hasMark()
120      * [ Belonging Class ]:
121      * - Point2D
122      * [ Description ]:
123      * - Query whether the point has a mark.
124      * - [ Params Description ]:
125      * - - No params.
126      * - [ Return Description ]:
127      * - - (bool) // Whether the point is marked.
128      * [ Usage ]:
129      * - "p1.hasMark(1)" returns a boolen value represent
whether
130      * - the point has a mark or not.
131      */
132     bool hasMark();
133
134     /*
135      * <[ Class Methods Defination ]>
136      * [ Method Name ]:
137      * - Point2D::getMark()
138      * [ Belonging Class ]:
139      * - Point2D
140      * [ Description ]:
141      * - Get the mark on the point.
142      * - [ Params Description ]:
143      * - - No params.
144      * - [ Return Description ]:
145      * - - (int) // The mark on the point.
146      * - Important: you are supposed to check whether it has
mark by
147      * - "hasMark()" before "getMark()".
148      * - You are not supposed to get the mark if the point
hasn't mark.
149      * - And I will panic if this happens.
150      * [ Usage ]:
151      * - "p1.getMark()" gets a integer represents the mark on
the point.
152      */
153     int getMark();
154 };
155
156
```

```
157
158 /*
159  * <[ Class Defination ]>
160  * [ Class Name ]:
161  * - Polygon2D
162  * [ Description ]:
163  * - The class defines a polygon with a set of orderred Point2D.
164  * [ Usage ]:
165  * - You can initialize the polygon with the vector consists of
    the
166  * - corner points.
167  */
168 class Polygon2D{
169 private:
170
171     std::vector<Point2D> cornerPoints;
172
173 public:
174
175     Polygon2D() = default;
176
177     /*
178      * <[ Class Methods Defination ]>
179      * [ Method Name ]:
180      * - C'tor
181      * [ Belonging Class ]:
182      * - Polygon2D
183      * [ Description ]:
184      * - To initialize the polygon object.
185      * [ Usage ]:
186      * - "Polygon2D pg((vector<Point2D>)pts);" defines a plane
    polygon
187      * - graph with the corner points stored in
    (vector<Point2D>)pts.
188      */
189     Polygon2D(std::vector<Point2D> cps);
190
191     /*
192      * <[ Class Methods Defination ]>
193      * [ Method Name ]:
194      * - Polygon2D::getPointByIdx()
195      * [ Belonging Class ]:
196      * - Polygon2D
197      * [ Description ]:
```

```
198      * - To get the refference of the point by pass in.
199      * - [ Params Description ]:
200      * - - [1] (int) // The index of the target point.
201      * - [ Return Description ]:
202      * - - (Point2D &) // The refference of the target point.
203      * - You are not supposed to pass in an invalid point and I
     will panic
204      * - if this happen.
205      * [ Usage ]:
206      * - "pg.getPointByIdx(idxP)" return the reference of target
     point of p
207      * - whose index is idxP.
208      */
209     Point2D &getPointByIdx(int);
210 
211     /*
212      * <[ Class Methods Defination ]>
213      * [ Method Name ]:
214      * - Polygon2D::getPreByIdx()
215      * [ Belonging Class ]:
216      * - Polygon2D
217      * [ Description ]:
218      * - To get the refference of the previous point by pass in.
219      * - [ Params Description ]:
220      * - - [1] (int) // The index of the current point.
221      * - [ Return Description ]:
222      * - - (Point2D &) // The refference of the previous point.
223      * - You are not supposed to pass in an invalid point and I
     will panic
224      * - if this happen.
225      * [ Usage ]:
226      * - "pg.getPreByIdx(idxP)" return the previous point of p
     whose index
227      * - is idxP.
228      */
229     Point2D &getPreByIdx(int);
230 
231     /*
232      * <[ Class Methods Defination ]>
233      * [ Method Name ]:
234      * - Polygon2D::getNextByIdx()
235      * [ Belonging Class ]:
236      * - Polygon2D
237      * [ Description ]:
```

```
238      * - To get the refference of the next point by pass in.
239      * - [ Params Description ]:
240      * - - [1] (int) // The index of the current point.
241      * - [ Return Description ]:
242      * - - (Point2D &) // The refference of the next point.
243      * - You are not supposed to pass in an invalid point and I
     will panic
244      * - if this happen.
245      * [ Usage ]:
246      * - "pg.getNextByIdx(idxP)" return the next point of p
     whose index
247      * - is idxP.
248      */
249     Point2D & getNextByIdx(int);
250 
251     /*
252      * <[ Class Methods Defination ]>
253      * [ Method Name ]:
254      * - Polygon2D::reset()
255      * [ Belonging Class ]:
256      * - Polygon2D
257      * [ Description ]:
258      * - Reset the point set in the polygon.
259      * - [ Params Description ]:
260      * - - No params.
261      * - [ Return Description ]:
262      * - - No params.
263      * [ Usage ]:
264      * - "pg.reset()" will reset pg.
265      */
266     void reset();
267 
268     /*
269      * <[ Class Methods Defination ]>
270      * [ Method Name ]:
271      * - Polygon2D::getSize()
272      * [ Belonging Class ]:
273      * - Polygon2D
274      * [ Description ]:
275      * - Get the number of points in the polygon.
276      * - [ Params Description ]:
277      * - - No params.
278      * - [ Return Description ]:
279      * - - (int) // The size of points.
```

```
280         * [ Usage ]:
281         * - "pg.getSize()" returns the size of points.
282         */
283        int getSize();
284
285        /*
286         * <[ Class Methods Defination ]>
287         * [ Method Name ]:
288         * - Polygon2D::insertPoint()
289         * [ Belonging Class ]:
290         * - Polygon2D
291         * [ Description ]:
292         * - Insert the point after the end of points, which should
    be the "previous"
293         * - point of the points[0].
294         * - [ Params Description ]:
295         * - - [1] (Point2D) // The Point to be insert.
296         * - - [2] (int) // The index of the point where will new
    point where put behind.
297         * - [ Return Description ]:
298         * - - No params.
299         * [ Usage ]:
300         * - "pg.insertPoint(p, 0)" will insert the point behind the
    point of index 0.
301         */
302        void insertPoint(Point2D, int);
303
304        /*
305         * <[ Class Methods Defination ]>
306         * [ Method Name ]:
307         * - Polygon2D::insertPointBack()
308         * [ Belonging Class ]:
309         * - Polygon2D
310         * [ Description ]:
311         * - Insert the point after the end of points, which should
    be the "previous"
312         * - point of the points[0].
313         * - [ Params Description ]:
314         * - - [1] (Point2D) // The Point to be insert.
315         * - [ Return Description ]:
316         * - - No params.
317         * [ Usage ]:
318         * - "pg.insertPointBack(p)" will insert the point after the
    end of points.
```

```
319      */
320     void insertPointBack(Point2D);
321 
322     /*
323      * <[ Class Methods Defination ]>
324      * [ Method Name ]:
325      * - Polygon2D::removePointBack()
326      * [ Belonging Class ]:
327      * - Polygon2D
328      * [ Description ]:
329      * - Remove the point after the end of points, which should
    be the "previous"
330      * - point of the points[0].
331      * - [ Params Description ]:
332      * - - No params.
333      * - [ Return Description ]:
334      * - - No params.
335      * [ Usage ]:
336      * - "pg.removePointBack(p)" will remove the point after the
    end of points.
337      */
338     void removePointBack();
339 };
340 
341 
342 
343 // Change it if the elements is not this.
344 using TableEleType = double;
345 // Pre declaration.
346 class Table2D;
347 
348 /*
349  * <[ Class Defination ]>
350  * [ Class Name ]
351  * - TableSlice
352  * [ Description ]:
353  * - The class defines a slice in 2d-table.
354  * [ Usage ]:
355  * - Actually, you shouldn't use it directly.
356  */
357 class TableSlice{
358 private:
359 
360     int idx1;
```

```
361     Table2D * table2D;
362 
363 public:
364 
365     /*
366      * <[ Class Methods Defination ]>
367      * [ Method Name ]:
368      * - C'tor
369      * [ Belonging Class ]:
370      * - TableSlice
371      * [ Description ]:
372      * - To initialize the table slice object.
373      * [ Usage ]:
374      * - "TableSlice tb(&t, 4);" indexed the fifth layer of
Table2D t.
375      */
376     TableSlice(Table2D *, int);
377 
378     /*
379      * <[ Class Overload Defination ]>
380      * [ Overload Option ]:
381      * - []
382      * [ Belonging Class ]:
383      * - TableSlice
384      * [ Description ]:
385      * - To get the element of the slice.
386      * - Actually, you shouldn't use it directly.
387      * - [ Params Description ]:
388      * - - [1] (int) // The index of element.
389      * - [ Return Description ]:
390      * - - (TableEleType) // The slice of the 2d-table.
391      * [ Usage ]:
392      * - "tb[1]" returns the second layer of 2d-table (start
from 0).
393      * - "tb[2][3]" returns the element at table[2][3].
394      */
395     TableEleType & operator[](int);
396 };
397 
398 
399 
400 
401 /*
402  * <[ Class Defination ]>
```

```
 * [ Class Name ]
 * - Table2D
 * [ Description ]:
 * - The class defines a 2d-table to store the votes of the match.
 * [ Usage ]:
 * - You should first set the relevant params before use. Then pass in the
 * - "Point2D" objects, the relevant function will return the match result.
 */
class Table2D{
// Private elements can be accessed by TableSlice.
friend class TableSlice;
private:

    std::pair<int,int> shape;
    std::vector<TableEleType> table;

public:

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - C'tor
     * [ Belonging Class ]:
     * - Table2D
     * [ Description ]:
     * - To initialize the 2d-table object.
     * [ Usage ]:
     * - "Table2D tb(3, 4);" defines a 3*4 2d-table.
     */
    Table2D(int ,int, TableEleType);

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - Table2D::getShape()
     * [ Belonging Class ]:
     * - Table2D
     * [ Description ]:
     * - To get the shape of the table.
     * - [ Params Description ]:
     * - - No params.
```

```
     * - [ Return Description ]:
     * - - (std::pair<int,int>) // The shape of the 2d-table is rt.first * rt.second.
     * [ Usage ]:
     * - "tb.getShape();" returns the shape of the 2d-table.
     */
    std::pair<int,int> getShape();

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - Table2D::getElements()
     * [ Belonging Class ]:
     * - Table2D
     * [ Description ]:
     * - To get all the elements of the table.
     * - [ Params Description ]:
     * - - No params.
     * - [ Return Description ]:
     * - - (std::vector<TableEleType>) // All the elements of the table.
     * [ Usage ]:
     * - "tb.getElements();" returns all the elements of the table.
     */
    std::vector<TableEleType> getElements();

    /*
     * <[ Class Overload Defination ]>
     * [ Overload Option ]:
     * - []
     * [ Belonging Class ]:
     * - Table2D
     * [ Description ]:
     * - To get the slice of the table.
     * - Actually, you are supposed to use it with "[x][y]" form.
     * - [ Params Description ]:
     * - - [1] (int) // The layer of slice.
     * - [ Return Description ]:
     * - - (TableSlice) // The slice of the 2d-table.
     * [ Usage ]:
     * - "tb[1]" returns the second layer of 2d-table (start from 0).
```

```
     * - "tb[2][3]" returns the element at table[2][3].
     */
    TableSlice operator[](int);
};


#endif
```

- `meta_data.cpp`

```cpp
#include "meta_data.h"

/*
 * <[ Class Methods Implementations ]>
 * [ Class Name ]:
 * - Point2D
 */

// Detail comments are in `.h` file!
Point2D::Point2D(double x, double y): x(x), y(y), isMarked(false){}

// Detail comments are in `.h` file!
double Point2D::getX(){
    return this->x;
}

// Detail comments are in `.h` file!
double Point2D::getY(){
    return this->y;
}

// Detail comments are in `.h` file!
double Point2D::getDis(Point2D &that){
    double dx = this->x - that.getX();
    double dy = this->y - that.getY();
    return sqrt(dx     * dx + dy     * dy);
}

// Detail comments are in `.h` file!
void Point2D::setMark(int m){
    this->mark = m;
    this->isMarked = true;
}
```

```cpp

// Detail comments are in `.h` file!
bool Point2D::hasMark(){
    return this->isMarked;
}

// Detail comments are in `.h` file!
int Point2D::getMark(){
    // Panic for invalid option.
    if(!isMarked){
        std::cerr << __FILE__ << "/" << __LINE__ <<  " [FATAL] The point is not marked!\n";
        exit(0);
    }
    return this->mark;
}



/*
 * <[ Class Methods Implementations ]>
 * [ Class Name ]:
 * - Polygon2D
 */

// Detail comments are in `.h` file!
Polygon2D::Polygon2D(std::vector<Point2D> cps)
    :   cornerPoints(cps){
    // Set alias to make codes below briefly.
    std::vector<Point2D> &vec = Polygon2D::cornerPoints;
    // Set the mark of the point to be the index in the vector.
    // It will make the other option more convenient.
    for(int i = 0; i < vec.size(); ++i){
        vec[i].setMark(i);
    }
}

// Detail comments are in `.h` file!
Point2D & Polygon2D::getPointByIdx(int idxP){
    // Store the size to make codes below briefly.
    int pgSize = this->cornerPoints.size();
    // Panic if the param is invalid.
    if(idxP >= pgSize){
        std::cerr << __FILE__ << "/" << __LINE__ <<
```

```
            " [FATAL] Invalid idxP! idxP = [ " << idxP << " ] while pgSize = [ " << pgSize << " ]\n";
        exit(0);
    }
    return this->cornerPoints[idxP];
}

// Detail comments are in `.h` file!
Point2D & Polygon2D::getPreByIdx(int idxP){
    // Store the size to make codes below briefly.
    int pgSize = this->cornerPoints.size();
    // Panic if the param is invalid.
    if(idxP >= pgSize){
        std::cerr << __FILE__ << "/" << __LINE__ <<
            " [FATAL] Invalid idxP! idxP = [ " << idxP << " ] while pgSize = [ " << pgSize << " ]\n";
        exit(0);
    }
    // Calculate the next index.
    int nextIdx = (idxP + pgSize - 1) % pgSize;
    return this->getPointByIdx(nextIdx);
}

// Detail comments are in `.h` file!
Point2D & Polygon2D::getNextByIdx(int idxP){

    // Store the size to make codes below briefly.
    int pgSize = this->getSize();
    // Panic if the param is invalid.
    if(idxP >= pgSize){
        std::cerr << __FILE__ << "/" << __LINE__ <<
            " [FATAL] Invalid idxP! idxP = [ " << idxP << " ] while pgSize = [ " << pgSize << " ]\n";
        exit(0);
    }
    // Calculate the next index.
    int nextIdx = (idxP + 1) % pgSize;
    return this->getPointByIdx(nextIdx);
}

// Detail comments are in `.h` file!
void Polygon2D::reset(){
    while(this->getSize()) cornerPoints.pop_back();
    return;
```

```
}

// Detail comments are in `.h` file!
int Polygon2D::getSize(){
    return this->cornerPoints.size();
}

// Detail comments are in `.h` file!
void Polygon2D::insertPoint(Point2D p, int idx){
    // Panic if the param is invalid.
    if(idx >= this->getSize()){
        std::cerr << __FILE__ << "/" << __LINE__ <<
            " [FATAL] Invalid idxP! idxP = [ " << idx << " ]
while pgSize = [ " << this->getSize() << " ]\n";
        exit(0);
    }
    auto & vec = this->cornerPoints;
    vec.insert( vec.begin() + idx, p );
    return;
}

// Detail comments are in `.h` file!
void Polygon2D::insertPointBack(Point2D p){
    this->cornerPoints.push_back(p);
    ( this->cornerPoints.end()-1 )->setMark( this-
>cornerPoints.size()-1 );
    return;
}

// Detail comments are in `.h` file!
void Polygon2D::removePointBack(){
    if(this->cornerPoints.size() > 0){
        this->cornerPoints.pop_back();
    }
    return;
}



/*
 * <[ Class Methods Implementations ]>
 * [ Class Name ]:
 * - TableSlice
 */
```

```cpp

// Detail comments are in `.h` file!
TableSlice::TableSlice(Table2D * t, int idx): idx1(idx), table2D(t){}

// Detail comments are in `.h` file!
TableEleType & TableSlice::operator[](int idx2){
    int offset = idx1 * table2D->shape.second + idx2;
    return table2D->table[offset];
}



/*
 * <[ Class Methods Implementations ]>
 * [ Class Name ]:
 * - Table2D
 */

// Detail comments are in `.h` file!
Table2D::Table2D(int m, int n, TableEleType dVal): shape(m,n){
    // Reshape the vector to m * n.
    table.resize(m * n);
    for(auto it = table.begin(); it != table.end(); ++it){
        *it = dVal;
    }
}

// Detail comments are in `.h` file!
std::pair<int,int> Table2D::getShape(){
    return this->shape;
}

// Detail comments are in `.h` file!
std::vector<TableEleType> Table2D::getElements(){
    return table;
}

// Detail comments are in `.h` file!
TableSlice Table2D::operator[](int idx){
    return TableSlice(this, idx);
}
```

## 3 solver

- main.cpp

```cpp
#include <iomanip>
#include <iostream>
#include <ctime>

#include "optimal_match/voting_tree.h"

#define WIDTH 4

void printResult( std::vector< std::pair<int,int> > & res){
    std::cout << res.size() << std::endl;
    for(auto it = res.begin(); it != res.end(); ++it){
        std::cout << std::setw(WIDTH) << it->first << std::setw(WIDTH) << it->second << std::endl;
    }
}

void solve(){
    MatchReferee judger1(0.05, 0.05, 1.0);
    MatchReferee judger2(0.05, 0.05, 1.0);
    auto inA = VotingTree::readPts(std::cin);
    auto inB = VotingTree::readPts(std::cin);
    VotingTree vTree(inA, inB, judger1, judger2, 5);

    clock_t timerI = clock();
    vTree.dealOptimalMatch();
    clock_t timerF = clock();
    double delTime = (double)(timerF-timerI)/CLOCKS_PER_SEC;
    std::cerr << "It takes : [ " << delTime << " ] Seconds!\n";

    // // Debug::Print table.
    // auto vTable = vTree.getVotingTable();
    // std::cerr << "\n\n";
    // for(auto i = 0; i < vTable.getShape().first; ++i){
    //     for(auto j = 0; j < vTable.getShape().second; ++j){
    //         std::cerr << std::setw(WIDTH) << vTable[i][j] << " ";
    //     }
    //     std::cerr << std::endl;
    // }
    // std::cerr << "\n\n";
```

```

    auto result = vTree.getOptimalMatch();
    printResult(result);
}

int main(){
    std::ios::sync_with_stdio(false);
    int _;
    std::cin >> _;
    for(int i = 1; i <= _; ++i){
        std::cerr << " Test case " << i << " Start.\n";
        solve();
        std::cerr << " Test case " << i << " Finished.\n";
    }
    return 0;
}
```

- match_referee.h

```
#ifndef __MATCH_REFEREE__
#define __MATCH_REFEREE__

// Head files.
#include <set>
#include <cmath>
#include <iostream>
#include "../../lib/meta_data.h"

/*
 * <[ Class Defination ]>
 * [ Class Name ]
 * - MatchReferee
 * [ Description ]:
 * - The class defines a judement system for optimal match.
 * [ Usage ]:
 * - You should first set the relevant params before use. Then pass in the
 * - "Point2D" objects, the relevant function will return the match result.
 */
class MatchReferee{
private:

    double angleTolerance;
```

```
24      double edgeRatioTolerance;
25      double weight;
26
27  public:
28
29      /*
30       * <[ Class Methods Defination ]>
31       * [ Method Name ]:
32       * - C'tor
33       * [ Belonging Class ]:
34       * - MatchReferee
35       * [ Description ]:
36       * - To initialize the points object.
37       * [ Usage ]:
38       * - "MatchReferee p(0.1, 0.2, 1.0);" initialize the referee
  with
39       * - angleTolerance = 0.1 and edgeRatioTolerance = 0.2 and
  weight = 1.0.
40       */
41      MatchReferee(double, double, double);
42
43      // Functional Methods
44
45      /*
46       * <[ Class Methods Defination ]>
47       * [ Method Name ]:
48       * - MatchReferee::isMatch()
49       * [ Belonging Class ]:
50       * - MatchReferee
51       * [ Description ]:
52       * - Get the weight of the juder.
53       * - [ Params Description ]:
54       * - No params.
55       * - [ Return Description ]:
56       * - - (double) // The weight of the judger.
57       * [ Usage ]:
58       * - "mr.getWeight();" gets the weight of the judger.
59       */
60      double getWeight();
61
62      /*
63       * <[ Class Methods Defination ]>
64       * [ Method Name ]:
65       * - MatchReferee::isMatch()
```

```
     * [ Belonging Class ]:
     * - MatchReferee
     * [ Description ]:
     * - To judge whether the angle with edge can match.
     * - [ Params Description ]:
     * - - [1-3] (Point2D &) // The points form the first corner.
     * - - [4-6] (Point2D &) // The points form the second
corner.
     * - [ Return Description ]:
     * - - (bool) // Whether the two corner can be matched.
     * [ Usage ]:
     * - "mr.isMatch(U, V, W, X, Y, Z);" gets a boolen value
     * - represents whether the △UVW are approximately similar
     * - with △XYZ.
     */
    bool isMatch(Point2D &, Point2D &, Point2D &, Point2D &,
Point2D &, Point2D &);
};

#endif
```

- match_referee.cpp

```
#ifndef __MATCH_REFEREE__
#define __MATCH_REFEREE__

// Head files.
#include <set>
#include <cmath>
#include <iostream>
#include "../../lib/meta_data.h"

/*
 * <[ Class Defination ]>
 * [ Class Name ]
 * - MatchReferee
 * [ Description ]:
 * - The class defines a judement system for optimal match.
 * [ Usage ]:
 * - You should first set the relevant params before use. Then
pass in the
 * - "Point2D" objects, the relevant function will return the
match result.
 */
```

```
class MatchReferee{
private:

    double angleTolerance;
    double edgeRatioTolerance;
    double weight;

public:

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - C'tor
     * [ Belonging Class ]:
     * - MatchReferee
     * [ Description ]:
     * - To initialize the points object.
     * [ Usage ]:
     * - "MatchReferee p(0.1, 0.2, 1.0);" initialize the referee with
     * - angleTolerance = 0.1 and edgeRatioTolerance = 0.2 and weight = 1.0.
     */
    MatchReferee(double, double, double);

    // Functional Methods

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - MatchReferee::isMatch()
     * [ Belonging Class ]:
     * - MatchReferee
     * [ Description ]:
     * - Get the weight of the juder.
     * - [ Params Description ]:
     * - No params.
     * - [ Return Description ]:
     * - - (double) // The weight of the judger.
     * [ Usage ]:
     * - "mr.getWeight();" gets the weight of the judger.
     */
    double getWeight();

```

```
62      /*
63       * <[ Class Methods Defination ]>
64       * [ Method Name ]:
65       * - MatchReferee::isMatch()
66       * [ Belonging Class ]:
67       * - MatchReferee
68       * [ Description ]:
69       * - To judge whether the angle with edge can match.
70       * - [ Params Description ]:
71       * - - [1-3] (Point2D &) // The points form the first corner.
72       * - - [4-6] (Point2D &) // The points form the second
corner.
73       * - [ Return Description ]:
74       * - - (bool) // Whether the two corner can be matched.
75       * [ Usage ]:
76       * - "mr.isMatch(U, V, W, X, Y, Z);" gets a boolen value
77       * - represents whether the △UVW are approximately similar
78       * - with △XYZ.
79       */
80      bool isMatch(Point2D &, Point2D &, Point2D &, Point2D &,
Point2D &, Point2D &);
81  };
82
83  #endif
```

- voting_tree.h

```
1   #ifndef __VOTING_TREE__
2   #define __VOTING_TREE__
3
4   // Head file.
5   #include <vector>
6   #include <iostream>
7   #include "../../lib/meta_data.h"
8   #include "match_referee.h"
9
10  // Pre declaration.
11  class VotingTree;
12
13  /*
14   * <[ Class Defination ]>
15   * [ Class Name ]
16   * - CurStage
17   * [ Description ]:
```

```
 * - The class store the current stage.
 */
class CurStage{
// Private elements can be accessed by TableSlice.
friend class VotingTree;
private:

    Polygon2D curA, curB;
    std::vector< std::pair<int,int> > curPath;

public:

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - CurStage::reset()
     * [ Belonging Class ]:
     * - CurStage
     * [ Description ]:
     * - It will reset the current stage.
     * - [ Params Description ]:
     * - - No params.
     * - [ Return Description ]:
     * - - No params.
     * [ Usage ]:
     * - "cur.reset()" will do the things above.
     */
    void reset();

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - CurStage::storeStage()
     * [ Belonging Class ]:
     * - CurStage
     * [ Description ]:
     * - It will store the current stage.
     * - [ Params Description ]:
     * - - [1] (VotingTree *) // The voting tree store the data.
     * - - [2-3] (int) // The index of points in pgA and pgB to
be store.
     * - [ Return Description ]:
     * - - No params.
     * [ Usage ]:
```

```
     * - "cur.storeStage(1, 2)" will store the pA[1] and pB[2] to the stage.
     */
    void storeStage(VotingTree *, int, int);

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - CurStage::recoverStage()
     * [ Belonging Class ]:
     * - CurStage
     * [ Description ]:
     * - It will recover the current stage.
     * - [ Params Description ]:
     * - - No params.
     * - [ Return Description ]:
     * - - No params.
     * [ Usage ]:
     * - "cur.recoverStage()" will do the things above.
     */
    void recoverStage();

};

/*
 * <[ Class Defination ]>
 * [ Class Name ]
 * - VotingTree
 * [ Description ]:
 * - The class defines a voting tree object to solve the problem.
 * [ Usage ]:
 * - You should construct the voting tree object with the input that consists of
 * - two vector.
 */
class VotingTree{
// Private elements can be accessed by TableSlice.
friend class CurStage;
public:
    // Define a memeber struct.
    struct MatchPair{
        int x, y;
        double ele;
```

```
        // Actually, it DON'T means '<', I actually define the comparation function
        // for std::sort(), it act just like the custom cmp() you usually defined.
        // After this, the elements will be sorted in monotonically decreasing order.
        bool operator<( const MatchPair & rhs) const;
    };
private:

    Polygon2D pgA, pgB;
    Table2D votingTable;
    MatchReferee &judger1;
    MatchReferee &judger2;
    std::vector< std::pair<int,int> > optimalMatch;
    int credibleLowerLimit; // At least 3. Because any 2points-2points pair will match.
    double mutationRatio; // The ratio of the mutation in 'matchAccordingTalbe' progress.

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - VotingTree::voteByDfs()
     * [ Belonging Class ]:
     * - VotingTree
     * [ Description ]:
     * - It will dfs the possible matching and get the sum of vote of each match.
     * - [ Params Description ]:
     * - - (CurStage) // The current stage.
     * - [ Return Description ]:
     * - - std::pair<double, bool> // The vote of the current match and the success sign of the path.
     * - - That means, if the path is impossible, the second value will be false, vice versa.
     * [ Usage ]:
     * - "vt.voteByDfs()" will dfs and get the vote of current match (also the son of the match).
     */
    std::pair<double, bool> voteByDfs(CurStage &);

public:

```

```
137     /*
138      * <[ Class Static Methods Defination ]>
139      * [ Method Name ]:
140      * - VotingTree::readPts()
141      * [ Belonging Class ]:
142      * - VotingTree
143      * [ Description ]:
144      * - To get the input data from istream.
145      * - [ Params Description ]:
146      * - - [1] (std::istream) // The input stream where we get
input from.
147      * - [ Return Description ]:
148      * - - (std::vector<Point2D>) // The points.
149      * [ Usage ]:
150      * - "VotingTree::readPts(cin)" returns a vector of Points2D
read from
151      * - cin stream.
152      */
153     static std::vector<Point2D> readPts(std::istream &);
154 
155     /*
156      * <[ Class Methods Defination ]>
157      * [ Method Name ]:
158      * - C'tor
159      * [ Belonging Class ]:
160      * - VotingTree
161      * [ Description ]:
162      * - To initialize the polygon object.
163      * [ Usage ]:
164      * - "VotingTree vt(A, B, j1, j2, 5, 1.0);" defines a voting
tree solve the situation with
165      * - graph with the corner points stored in
(vector<Point2D>)pts A and B, and the match
166      * - judgement will be done by j1 and j2. The number of
points in polygon shouldn't be
167      * - less than 5. What's more, the mutationRatio is 1.0
168      */
169     VotingTree(std::vector<Point2D> &, std::vector<Point2D> &,
MatchReferee &, MatchReferee &, int, double);
170 
171     /*
172      * <[ Class Methods Defination ]>
173      * [ Method Name ]:
174      * - VotingTree::searchAndVote()
```

```
     * [ Belonging Class ]:
     * - VotingTree
     * [ Description ]:
     * - It will search for the possible matching schemes and vote for them.
     * - The votes will be updated to votingTable.
     * - [ Params Description ]:
     * - - No params.
     * - [ Return Description ]:
     * - - No params.
     * [ Usage ]:
     * - "vt.searchAndVote()" will do the things above.
     */
    void searchAndVote();

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - VotingTree::getVotingTable()
     * [ Belonging Class ]:
     * - VotingTree
     * [ Description ]:
     * - It will return the copy of the voting table.
     * - [ Params Description ]:
     * - - No params.
     * - [ Return Description ]:
     * - - (Table2D) // The copy of the voting table.
     * [ Usage ]:
     * - "vt.getVotingTable()" returns the copy of the voting table.
     */
    Table2D getVotingTable();

    /*
     * <[ Class Methods Defination ]>
     * [ Method Name ]:
     * - VotingTree::matchAccordingTalbe()
     * [ Belonging Class ]:
     * - VotingTree
     * [ Description ]:
     * - It will deal the voting table and get the matching relationship.
     * - The answer will be stored into the optimalMatch.
     * - [ Params Description ]:
```

```
216      * - - No params.
217      * - [ Return Description ]:
218      * - - No params.
219      * [ Usage ]:
220      * - "vt.matchAccordingTalbe()" will do the things above.
221      */
222     void matchAccordingTalbe();
223 
224     /*
225      * <[ Class Methods Defination ]>
226      * [ Method Name ]:
227      * - VotingTree::dealOptimalMatch()
228      * [ Belonging Class ]:
229      * - VotingTree
230      * [ Description ]:
231      * - To calculate the optimal match of the two polygon.
232      * - [ Params Description ]:
233      * - - No params.
234      * - [ Return Description ]:
235      * - - No params.
236      * [ Usage ]:
237      * - "vt.dealOptimalMatch()" calculates the optimal match
    and store it in
238      * - optimalMatch.
239      */
240     void dealOptimalMatch();
241 
242     /*
243      * <[ Class Methods Defination ]>
244      * [ Method Name ]:
245      * - VotingTree::getOptimalMatch()
246      * [ Belonging Class ]:
247      * - VotingTree
248      * [ Description ]:
249      * - To get the optimal match calculated.
250      * - You will get nothing if you getOptimalMatch() before
    dealOptimalMatch().
251      * - [ Params Description ]:
252      * - - No params.
253      * - [ Return Description ]:
254      * - - (std::vector< std::pair<int,int> >) // The optimal
    match.
255      * [ Usage ]:
256      * - "vt.getOptimalMatch()" returns the optimal match.
```

```
257      */
258     std::vector< std::pair<int,int> > getOptimalMatch();
259 };
260 
261 
262 #endif
```

- voting_tree.cpp

```
1  #include "voting_tree.h"
2  
3  /*
4   * <[ Class Methods Implementations ]>
5   * [ Class Name ]:
6   * - CurStage
7   */
8  
9  // Detail comments are in `.h` file!
10 void CurStage::reset(){
11     this->curA.reset();
12     this->curB.reset();
13     this->curPath.resize(0);
14 }
15 
16 // Detail comments are in `.h` file!
17 void CurStage::storeStage(VotingTree * vt, int ia, int ib){
18     curA.insertPointBack( vt->pgA.getPointByIdx(ia) );
19     curB.insertPointBack( vt->pgB.getPointByIdx(ib) );
20     curPath.push_back( std::make_pair(ia, ib) );
21 }
22 
23 // Detail comments are in `.h` file!
24 void CurStage::recoverStage(){
25     curA.removePointBack();
26     curB.removePointBack();
27     curPath.pop_back();
28 }
29 
30 
31 /*
32  * <[ Class Methods Implementations ]>
33  * [ Class Name ]:
34  * - VotingTree
35  */
```

```cpp

// Detail comments are in `.h` file!
bool VotingTree::MatchPair::operator<( const MatchPair & rhs) const {
    return this->ele > rhs.ele;
}

// Detail comments are in `.h` file!
std::vector<Point2D> VotingTree::readPts(std::istream & input){
    // Initialize the variable to be used.
    int size;
    double x, y;
    std::vector<Point2D> vec;
    // Read in the size of the polygon.
    input >> size;
    for(int i = 0; i < size; ++i){
        input >> x >> y;
        vec.push_back( Point2D(x, y) );
    }
    return vec;
}

// Detail comments are in `.h` file!
VotingTree::VotingTree(std::vector<Point2D> & A, std::vector<Point2D> & B, MatchReferee & judger1, MatchReferee & judger2, int cll, double mr)
    :   pgA(A),
        pgB(B),
        votingTable(A.size(), B.size(), 0),
        judger1(judger1),
        judger2(judger2),
        credibleLowerLimit(cll > 3 ? cll : 3),
        mutationRatio(mr){}

// Detail comments are in `.h` file!
std::pair<double, bool> VotingTree::voteByDfs(CurStage & cur){
    double retVote = 0;
    bool retSuccess = false;
    auto curMatch = *(cur.curPath.end()-1);
    int iaCur = curMatch.first, ibCur = curMatch.second;
    // That is, if we want the current match to be the leaf node, we will check this.
    bool enoughPoints = ( cur.curPath.size() >= this->credibleLowerLimit );
```

```
75
76      if(cur.curPath.size() >= 3){
77           // The pgA and the newest point of pgA in current path.
78          auto & curA = cur.curA;
79          auto & pA = curA.getPointByIdx( cur.curA.getSize() - 1
    );
80          // The pgB and the newest point of pgB in current path.
81          auto & curB = cur.curB;
82          auto & pB = curB.getPointByIdx( cur.curB.getSize() - 1
    );
83
84          // Log.
85          // std::cerr << "[" << __FILE__ << "/" << __LINE__ << "]
    : " <<  "Now dfs at :\n" << "id: " << pA.getMark() << ", pos: ("
    << pA.getX() << ", " << pA.getY() << ")" << std::endl <<  "id: "
    << pB.getMark() << ", pos: (" << pB.getX() << ", " << pB.getY()
    << ")" << std::endl;
86
87          // Judge whether the point is matched.
88          Point2D * Ap, * Am, * An, * Bp, * Bm, * Bn;
89
90          // Judgement 1.
91          // Where cur point is the end of the corner.
92          An = &pA,
93          Am = &curA.getPreByIdx( An->getMark() ),
94          Ap = &curA.getPreByIdx( Am->getMark() );
95          Bn = &pB,
96          Bm = &curB.getPreByIdx( Bn->getMark() ),
97          Bp = &curB.getPreByIdx( Bm->getMark() );
98
99          bool isMatched1 = judger1.isMatch(
100             *Ap, *Am, *An,
101             *Bp, *Bm, *Bn
102         );
103
104         // Judgement 2.
105         // Where cur point is the vertex of the corner.
106         Am = &pA,
107         An = &curA.getNextByIdx( An->getMark() ),
108         Ap = &curA.getPreByIdx( Am->getMark() );
109         Bm = &pB,
110         Bn = &curB.getNextByIdx( Bn->getMark() ),
111         Bp = &curB.getPreByIdx( Bm->getMark() );
112
```

```
113         bool isMatched2 = judger2.isMatch(
114             *Ap, *Am, *An,
115             *Bp, *Bm, *Bn
116         );
117 
118         if( isMatched1 && isMatched2 ){
119             // Totally match, that means the current match is
    ok. That means we
120             // should add the vote to the table. But we still
    want to find deeper
121             // match, so we should go on.
122             retVote = judger1.getWeight() + judger2.getWeight();
123 
124             // Update the votes.
125             votingTable[iaCur][ibCur] += retVote * retSuccess;
126         } else if( isMatched1 || isMatched2 ){
127             // Not totally match, so we won't let it go on. But
    it do has some good
128             // feature, so return but with vote.
129             retVote = judger1.getWeight() * isMatched1 +
    judger2.getWeight() * isMatched2;
130             retSuccess = enoughPoints;
131             // Update the votes.
132             votingTable[iaCur][ibCur] += retVote * retSuccess;
133             return std::pair<double,bool>(retVote,
    enoughPoints);
134         } else {
135             // Not match.
136             return std::pair<double,bool>(0, false);
137         }
138     }
139 
140     // The point is leagal, means we should search more deeper.
141     // Here I iterate the following point, be carefull that the
    boundary of A and B
142     // is different. We should iterate all the possible points
    in B but only points
143     // that have not been the endpoint of the root node. So that
    we can have all the
144     // situation.
145     // Note that we can't use pA.getMark() or pB.getMark() here
    because it will get
146     // the index in curPolygon system, but we want to get the
    index in origin polygon
```

```
147     // system.
148 
149     // iaLast is the index of the point got from cur path
    history. That is, the last
150     // selected point in A. "last" is for the next point. "cur"
    is for the current
151     // stage.
152     int & iaLast = iaCur;
153     // iaStart is just the index of point after iaLast in a ring
    sequence (which means
154     // the previous point of the head is the end).
155     int iaStart = this->pgA.getNextByIdx(iaLast).getMark();
156     // ibLast is the index of the point got from cur path
    history. That is, the last
157     // selected point in B. "last" is for the next point. "cur"
    is for the current
158     // stage.
159     int & ibLast = ibCur;
160     // ibEldest is the index of the point got from cur path
    history. That is, the first
161     // selected point in B.
162     int ibEldest = cur.curPath.begin()->second;
163     // ibStart is just the index of point after iaLast in a ring
    sequence (which means
164     // the previous point of the head is the end).
165     int ibStart = this->pgB.getNextByIdx(ibLast).getMark();
166     // Now we should iterate all the possible situation.
167     // Iterate all the possible ia, that is from the next of
    iaCur to the bigges index.
168     for(int ia = iaStart; ia > iaLast; ia =
    pgA.getNextByIdx(ia).getMark()){
169         // Iterate all the possible ib, that is from the next of
    ibCur to the previous
170         // of the first ib.
171         for(int ib = ibStart; ib != ibEldest; ib =
    pgB.getNextByIdx(ib).getMark()){
172             cur.storeStage(this, ia, ib);
173             auto response = this->voteByDfs(cur);
174             double vote = response.first;
175             double success = response.second;
176             // The vote of (ia, ib) contribute to the vote of
    its father match, i.e.
177             // the current point.
178             retVote += vote * success;
```

```
            // If one way succeed, that means the path will contribute.
            retSuccess = retSuccess || success;
            cur.recoverStage();
        }
    }

    if(retSuccess == 0){
        // No legal son node, so check if cur is legal leaf node or not.
        // Note that reVote will be overloaded because no leagl son node exsits.
        retVote = judger1.getWeight() + judger2.getWeight();
        retSuccess = enoughPoints;
    }

    votingTable[iaCur][ibCur] += retVote * retSuccess;
    return std::pair<double,bool>(retVote, retSuccess);
}

// Detail comments are in `.h` file!
void VotingTree::searchAndVote(){
    // Initialize the variable to be used.
    for(int ia = 0; ia < pgA.getSize(); ++ia){
        for(int ib = 0; ib < pgB.getSize(); ++ib){
            // Create a new current stage.
            CurStage cur;
            cur.storeStage(this, ia, ib);
            // Go deeper.
            this->voteByDfs(cur);
            // The CurStage object will be destructed so we needn't recover stage.
            // cur.recoverStage();
        }
    }
}

// Detail comments are in `.h` file!
Table2D VotingTree::getVotingTable(){
    return this->votingTable;
}

// // Detail comments are in `.h` file!
// void VotingTree::matchAccordingTalbe(){
```

```
219 //     // Reset the result space.
220 //     this->optimalMatch.clear();
221 //     // The vector to store the elements in talbe.
222 //     std::vector<MatchPair> vec;
223 //     // The set data structure to note whether the point is
    visited.
224 //     std::set<int> visA, visB;
225 //     // Set alias to make code briefly.
226 //     auto & vt = this->votingTable;
227 //     auto shape = vt.getShape();
228 //     // Extract elements from the table.
229 //     for(int i = 0; i < shape.first; ++i){
230 //         for(int j = 0; j < shape.second; ++j){
231 //             MatchPair te = {i, j, vt[i][j]};
232 //             vec.push_back(te);
233 //         }
234 //     }
235 //     // The comparation rule is defined in the struct
    defination.
236 //     std::sort(vec.begin(), vec.end());
237 //     double ave = vec[0].ele + vec[ vec.size()-1 ].ele;
238 //     ave /= 2;
239 //     for(int i = 0; i < vec.size(); ++i){
240 //         if(vec[i].ele < ave) break;
241 //         auto cur = vec[i];
242 //         if(visA.find(cur.x) == visA.end() && visB.find(cur.y)
    == visB.end()){
243 //             visA.insert(cur.x), visB.insert(cur.y);
244 //             this->optimalMatch.push_back(std::pair<int,int>
    (cur.x+1, cur.y+1));
245 //         }
246 //     }
247 //     // Sort it in accressment order.
248 //     std::sort(optimalMatch.begin(), optimalMatch.end());
249 //     // this->checkOrderUsingLIS();
250 //     if(optimalMatch.size() < this->credibleLowerLimit)
    optimalMatch.clear();
251 //     return;
252 // }
253
254 // Detail comments are in `.h` file!
255 void VotingTree::matchAccordingTalbe(){
256     // Reset the result space.
257     this->optimalMatch.clear();
```

```
258     auto & vt = this->votingTable;
259     auto shape = vt.getShape();
260     double maxEle = -1e9+9, minEle = 1e9+9;
261     // Extract elements from the table.
262     for(int i = 0; i < shape.first; ++i){
263         for(int j = 0; j < shape.second; ++j){
264             maxEle = std::max(maxEle, vt[i][j]);
265             minEle = std::min(minEle, vt[i][j]);
266         }
267     }
268     // Find the optimal match.
269     double ave = (maxEle + minEle) * 0.3;
270     bool oneRound = false;
271 
272     std::vector< std::pair<int,int> > tmp;
273     int tmpN = -1;
274     // Iterate the bound, that is, where the second index will
    from n-1 to 0.
275     // But actually, bound is the dividing line of the first
    index.
276         // 'j' is not mutated yet.
277     for(int bound = 0; bound < shape.first; ++bound){
278         // J ~ last chosen j.
279         // fJ ~ first chosen j, fI ~ first chosen i.
280         int J = 0, fJ = -1, fI = -1;
281         // Iterate i before the boud.
282         // 'j' is not mutated yet.
283         for(int i = bound; i < shape.first; ++i){
284             for(int j = J; j < shape.second; ++j){
285                 if(vt[i][j] > ave){
286                     J = j;
287                     if(fJ == -1) fJ = j;
288                     if(fI == -1) fI = i;
289                     optimalMatch.push_back(std::pair<int,int>
    (i+1,j+1));
290                     break;
291                 }
292             }
293         }
294         // Iterate i after the boud.
295         // 'j' is not mutated yet.
296         for(int i = 0; i < bound; ++i){
297             for(int j = J; j < shape.second; ++j){
298                 if(vt[i][j] > ave){
```

```
                    J = j;
                    if(fJ == -1) fJ = j;
                    if(fI == -1) fI = i;
                    optimalMatch.push_back(std::pair<int,int>
(i+1,j+1));
                    break;
                }
            }
        }
        // Iterate i before the boud.
        // 'j' is not mutated already.
        for(int i = bound; i < std::min(shape.first, fI); ++i){
            for(int j = 0; j < fJ; ++j){
                if(vt[i][j] > ave){
                    J = j;
                    optimalMatch.push_back(std::pair<int,int>
(i+1,j+1));
                    break;
                }
            }
        }
        // Iterate i after the boud.
        // 'j' is not mutated already.
        for(int i = 0; i < std::min(bound, fI); ++i){
            for(int j = J; j < fJ; ++j){
                if(vt[i][j] > ave){
                    J = j;
                    optimalMatch.push_back(std::pair<int,int>
(i+1,j+1));
                    break;
                }
            }
        }
        // Memorize the better match.
        if(int(optimalMatch.size()) > tmpN){
            tmpN = optimalMatch.size();
            tmp = optimalMatch;
        }
        optimalMatch.clear();
    }
    if(tmpN > 0){
        optimalMatch = tmp;
    }
    return;
```

```cpp
}

// Detail comments are in `.h` file!
void VotingTree::dealOptimalMatch(){
    // Search the matching programmes with dfs and initialize the voting table.
    this->searchAndVote();
    // Calculate the answer from the table.
    this->matchAccordingTalbe();
    return;
}

// Detail comments are in `.h` file!
std::vector< std::pair<int,int> > VotingTree::getOptimalMatch(){
    return this->optimalMatch;
}
```