Yao's 2D-MST Algorithm

Github: https://github.com/LyricZhao/YAO-MST2D

OOP Team Project by **Zhao Chenggang & Zhou Yunshuo**.

- √ OOP Design Style.
- √ High Scalability.
- √ Efficient.
- √ Comfortable Coding Style.
- √ Cross-Platform

1.Introduction

In this project , we implemented Yao's minimum spanning tree algorithm and compare its efficiency with the brute force algorithm.

To start with the algorithm presented by Yao, we need to first solve the eight-neighbors problem (*ENP*), which is "given a set V of n points in the plane, find for each point in V a nearest neighbor in the i region($1 \leq i \leq 8$), if it exists." Then, it is guaranteed that the set of edges E contains an MST on V.

Speaking of ENP, we will find a nearest neighbor to each point in the first region , and simply applies the method to all eight regions.

To find a nearest neighbor to each point p, we divided set V into s^2 cells. The cells can de classified into three classes according their relationships with p;

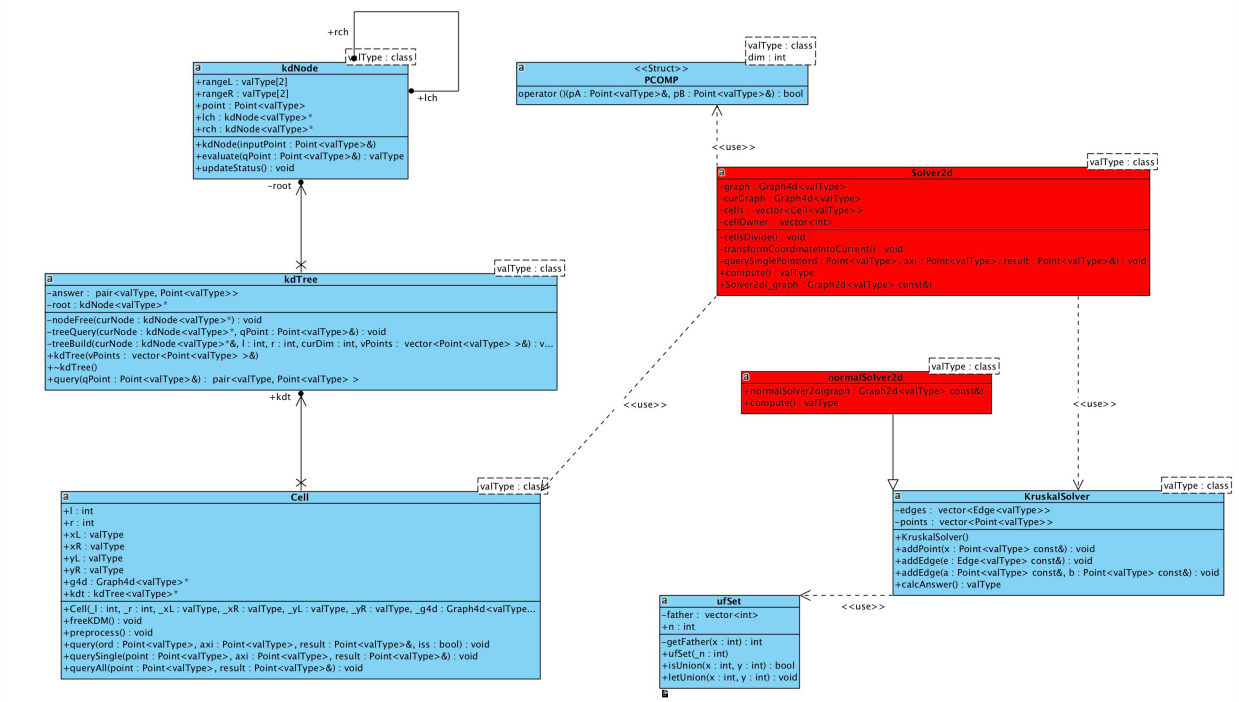
- Class 1, cells all of whose all points are in the region;
- Class 2, cells with no points in the region;
- Class 3, the remaining cells.
- For class 1, we implemented k-d tree algorithm (instead of Voronoi Algorithm, which is not required) to find p's nearest neighbor in the cell while ignoring cells of class 2 (because no point in this kind of class is in the region).
- We then examine each cell in turn for cells in class 3 and compute the distance between all points in the cell and v; this concludes the ENP question.

  Since the Kruskal algorithm is not the key point of this study, further explanation of that would be omitted.

  And we also implement *OpenMP* to accelerate the calcuation.

2. Implement Design

- To show the relationship better, we draw a UML class diagram.



- The two main solver is *Solver2d* and *normalSolver2d*, the main design style is by **composition**.

3.Details

3.0 Algorithm Flow

- *a*. Read a graph. (which is a vector of lots of points)
- *b*. Coordinate transformation.
- *c*. Divide the graph into several cells.
- *d*. Build k-d tree on each cell.
- *e*. Query every point in the cells and save the result
- *f*. Coordinate rotation.
- *g*. Repeat *b to f* for 8 times.
- *h*. Run Kruskal algorithm.
- *i*. Print the result.

3.1 Source (Library)

3.1.1 graph.cpp/h

This file is the implement of the basic representation of graphs, including *Point, Edge, Graph, Point4d, Graph4d*.

- class *Point* with template:
  - the implement of a 2-d point structure.
  - *operator +*, *operator -*, *length()* are supported.
  - *valType& d(int sw)* is for get the value of dimension *sw*, e.g. *p.d(1) += 1;*
- class *Edge* with template:
  - the implement of an edge structure.
  - *operator <* is supported.
- function *dot* with template:
  - return the dot product of a pair of vector
- *Graph2d*:

```
template <class valType>
using Graph2d = std:: vector<Point<valType> >;
```

  - a simple c++11 implement of a vector of several points.
- *Point4d*:

```
template <class valType>
using Point4d = std:: pair<Point<valType>, Point<valType> >;
```

  - a simple c++11 implement of a pair of points.
- *Graph4d*:

```
template <class valType>
using Graph4d = std:: vector<std:: pair<Point<valType>, Point<valType> >
>;
```

  - a simple c++11 implement of a pair of *Point4d*.

3.1.2 ufs.cpp/h

This file is Union-Find Set Algorithm implement.

- class *ufSet*:
  - *bool isUnion(int x, int y);* determines whether two points have the same father so that to check if a loop is generated during the process of growing a tree.
  - *void letUnion(int x, int y)* will connect two points.
  - *int getFather(int x); (private)* the details of the implement.

3.1.3 kruksal.cpp/h

The file is for the implement of Kruksal algorithm implement.

- class KruskalSolver with template:
  - *void addPoint(const Point<valType> &x);* : add a point to the tree by evaluating the smallest weight of edges in the current graph.
  - *void addEdge(const Edge<valType> &e); or void addEdge(const Point<valType> &a, const*

Point<valType> &b); : add an edge to the tree.
- *valType calcAnswer();* returns the result.

3.1.4 2d_Solver.cpp/h

The main part of Yao's algorithm.

- Macros for cell dividing:

```
# define BLOCK_LOW(id, p, n) ((id) * (n) / (p))
# define BLOCK_HGH(id, p, n) (BLOCK_LOW((id) + 1, p, n) - 1)
# define BLOCK_SIZ(id, p, n) (BLOCK_HGH(id, p, n) - BLOCK_LOW(id, p, n) +
1)
# define BLOCK_OWN(j, p, n) (((p) * ((j) + 1) - 1) / (n))
```

- Macro for *OpenMP*:

```
# define ENABLE_OPENMP
```

  - You can add this line to enable OpenMP multithread accelerating.
- class *Solver2d* with template:
  - *void cellsDivide(); private*: divide the graph into several cells.
  - *void transformCoordinateIntoCurrent(); private*: coordinate transformation for all points.
  - *void querySinglePoint(Point<valType> ord, Point<valType> axi, Point<valType> &result); private*: query the nearest point of a point.
  - *valType compute();* runs the algorithm and returns the result.
- function *PointTransform* with template:
  - Coordinate transformation for a single point.
- function *PointRotate* with template:
  - Rotation for a single point.

3.1.5 cell.cpp/h

- Macro for *k-d tree* optimization:

```
# define ENABLE_KD_OPT
```

  - You can add this line to enable k-d tree optimization.
- class *cell* with template:
  - *void freeKDM();* free the memory of k-d tree.
  - *void preprocess();*: to create kd tree for the use of queries in cells of class 1.
- *void query(Point<valType> ord, Point<valType> axi, Point<valType> &result, bool iss);*
  - Query the nearest point in a cell.
- *void querySingle(Point<valType> point, Point<valType> axi, Point<valType> &result);*: we examine the distance of points in cells of class 3 using this function, which is to compute and compare each of every distances, since it is guaranteed that the amount of cells of 3 is limited to at most *2s*.

3.1.6 KD-Opt/kdTree.cpp/h

The construction of a kd Tree is needed to solve the cells of class 1, this including the building function of a kd tree and the method to query a single node.

- class *kdNode* with template:
  - *valType evaluate(Point<valType> &qPoint);* : evaluating function for the search.
  - *void updateStatus();*: maintain the information of a node on k-d tree.
- class *kdTree* with template:
  - *void treeBuild(kdNode<valType>* &curNode, int l, int r, int curDim, std:: vector<Point<valType> > &vPoints);*: using recursion to create a kd tree with nodes.
  - *void nodeFree(kdNode<valType> *curNode);*: to eliminate a particular node on the tree.
  - *void treeBuild(kdNode<valType>* &curNode, int l, int r, int curDim, std:: vector<Point<valType> > &vPoints);*: return the minimum distance from a point to a kd tree graph.
  - *void treeQuery(kdNode<valType> *curNode, Point<valType> &qPoint);*: implementing recursion and std:: nth_element to find the distance between a certain point and a tree node
- struct PCOMP with template:

```
template <class valType, int dim>
struct PCOMP {
    bool operator () (Point<valType> &pA, Point<valType> &pB) {
        return pA.d(dim) < pB.d(dim);
    }
};
```

  - function object which is helpful to reduce the code.
  - Use:

```
std:: nth_element(&vPoints[l], &vPoints[mid], &vPoints[r + 1],
PCOMP<valType, 0> ());
```

3.2 Testing

At first, we generated a random graph with function generateRG() under rand.cpp , using std:: default_random_engine and std:: uniform_real_distribution<double>. To evaluate the efficiency of the algorithms, time.h is used to calculate the internal time while different platform naming Windows, Linux and Unix are supported. The results would then be printed on the screen.

3.2.1 normal_Solver.cpp/h

The brute force algorithm implement is for checking the result and comparing the time.

- class *normalSolver2d* (which is derived from KruskalSolver):
  - *valType compute():* return the result.

3.2.2 rand.cpp/h

A random engine implement which can support *double/float/int*.

- class randomEngine:
  - Give it a seed or not, you can get a random number by calling *getRandom()*
  - We use *std:: default_random_engine* and *std:: uniform_real_distribution *distribution* to do the work.
- function *void generateRG(Graph2d &G, int nodes, double l, double r)*:
  - Generate a graph.

3.2.3 time.cpp/h

This file is for timing, which could support different systems.

- function *unsigned long long get_wall_time()*: get wall time.
- function *double getTimeDiff(unsigned long long t0, unsigned long long t1)*: get the time between two time stamp.

3.2.4 main.cpp/h

Do the testing work.

- Macro for *OpenMP* setting:

```
# define THREADS_NUM 24
```

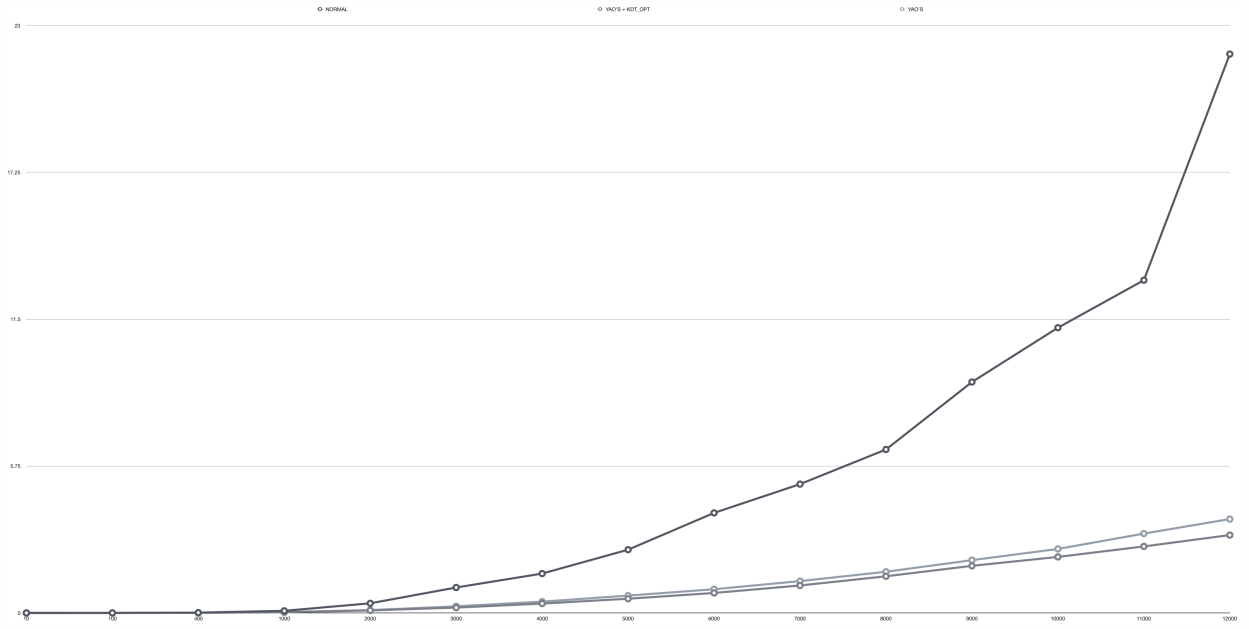
  - If *OpenMP* is enabled, you can change this to change the number of threads.
- Macro for algorithm:

```
# define ENABLE_YAO
# define ENABLE_NORMAL
```

  - Determine which algorithm to test or both.

4.Experiments

- We did some tests to check the result and also compare the efficiency.
- Here is the result, we can see that the Yao's algorithm($O(n^2)$) is must faster than the bruteforce ($O(n^2 logn)$), and kd-tree version($O(n^{\frac{5}{3}} logn)$) is also faster than the normal Yao's algorithm.



5.Conclusion

The project we did on implementing Yao's algorithm for finding minimum spanning tree is mainly about creating a testing environment, realizing Yao's algorithm and comparing its efficiency to bruteforce, and k-d tree methods.

In our project, we fully utilized Object-Oriented Programming design style by employing template types and function-object classes in order to lower code redundancy as well as to strengthen its scalability as a whole. To scrutinize and verify the generated answer we carried out, several tests have been conducted and the results are shown in the above chart, where it proposes that Yao's algorithm is must faster than the others. Therefore , the experiments turn out to be successful and correct.

However, there still left some space for improvements; for instance, to further ensure the complexity using Voronoi Algorithm as well as to visualize the results of the generated minimum spanning tree.

In a nutshell, this project implementing Yao's algorithm by applying OOP design style and validation on improving the efficiency of MST construction methods.