Ryan Helmlinger

Period 4

#include <iostream>

#include <cmath>

#include <vector>

#include<stdlib.h>

#include<algorithm>

#include "Point.h"

using namespace std;

bool sortByX (Point a, Point b) {

return (a.getY() < b.getY());

}

bool sortByY (Point a, Point b) {

return (a.getX() < b.getX());

}

void printToPPM (int sizeGrid, int numPoints, vector<Point> scaled, Point firstPointScaled, Point secondPointScaled){

int grid [sizeGrid][sizeGrid];

for (int row = 0; row < sizeGrid; row++){

for (int col = 0; col < sizeGrid; col++){

grid[row][col] = 1;

}

}

for (int index = 0; index < numPoints; index++){

grid[int(scaled[index].getX())][int(scaled[index].getY())] = 0;

}

for (int x = firstPointScaled.getX()-1; x <= firstPointScaled.getX()+1; x++){

for(int y = firstPointScaled.getY()-1; y <= firstPointScaled.getY()+1; y++){

grid[x][y]=2;

}

}

for (int x = secondPointScaled.getX()-1; x <= secondPointScaled.getX()+1; x++){

for(int y = secondPointScaled.getY()-1; y <= secondPointScaled.getY()+1; y++){

grid[x][y]=2;

}

}

sort (scaled.begin(), scaled.end(), sortByX);

cout << "P3 " << sizeGrid << " " << sizeGrid << " 1" << endl;

for (int row = 0; row < sizeGrid; row++){

for (int col = 0; col < sizeGrid; col++){

if (grid[row][col] == 1)

cout << "1 1 1 "; //white

else if (grid[row][col] == 0)

cout << "0 0 0 "; //black

else if (grid[row][col] == 2)

cout << "0 0 1 "; //blue

}

cout << endl;

}

}

double calculateDistance(Point first, Point second){

double distance = sqrt(pow(second.getX()-first.getX(),2) + pow(second.getY()-first.getY(),2));

return (distance);

}

vector<Point> bruteForce(vector<Point> points){

int numPoints = points.size();

double minDistance = 2.0;

Point firstPoint = Point();

Point secondPoint = Point();

for (int first = 0; first < numPoints-1; first++){

for (int second = first+1; second < numPoints; second++){

double distance = calculateDistance(points[first], points[second]);

if (distance < minDistance){

firstPoint = points[first];

secondPoint = points[second];

minDistance = distance;

}

}

}

vector<Point> firstAndSecond;

firstAndSecond.push\_back(firstPoint);

firstAndSecond.push\_back(secondPoint);

return (firstAndSecond);

}

vector<Point> recursiveDivide(vector<Point> points){

int pointsLength = points.size();

if (pointsLength==3){

return (bruteForce(points));

}

if (pointsLength==2){

return points;

}

int midpoint = int(pointsLength/2);

vector<Point> leftpoints;

vector<Point> rightpoints;

for (int index = 0; index < midpoint; index++){

leftpoints.push\_back(points[index]);

}

for (int index = midpoint; index < pointsLength; index++){

rightpoints.push\_back(points[index]);

}

vector<Point> leftpair = recursiveDivide(leftpoints);

vector<Point> rightpair = recursiveDivide(rightpoints);

vector<Point> smallerpair;

double leftdistance = calculateDistance(leftpair[0], leftpair[1]);

double rightdistance = calculateDistance(rightpair[0], rightpair[1]);

double mindistance = 2;

if (leftdistance < rightdistance){

smallerpair = leftpair;

mindistance = leftdistance;

}

else{

smallerpair = rightpair;

mindistance = rightdistance;

}

vector<Point> straddling;

for (int index = 0; index < pointsLength; index++){

if (abs(points[index].getX()-points[midpoint].getX())<mindistance){

straddling.push\_back(points[index]);

}

}

if (straddling.size()>2){

vector<Point> straddlingpair = bruteForce(straddling);

if (calculateDistance(straddlingpair[0], straddlingpair[1]) < mindistance)

return (straddlingpair);

}

if (straddling.size()==2){

if (calculateDistance(straddling[0], straddling[1]) < mindistance)

return (straddling);

}

return (smallerpair);

}

class HashEntry {

private:

vector<Point> key;

double value;

public:

HashEntry(vector<Point> key, double value) {

this->key = key;

this->value = value;

}

vector<Point> getKey() {

return key;

}

double getValue() {

return value;

}

};

const int tableSize = 128;

class HashMap {

private:

HashEntry \*\*table;

public:

HashMap() {

table = new HashEntry\*[tableSize];

for (int i=0; i<tableSize; i++){

table[i] = NULL;

}

}

double get(vector<Point> key) {

int hash = (key % tableSize);

while (table[hash] != NULL && table[hash]->getKey() != key)

hash = (hash+1)%tableSize;

if(table[hash] == NULL)

return -1;

else

return table[hash]->getValue();

}

void put(vector<Point> key, double value) {

int hash = (key%tableSize);

while(table[hash] != NULL && table[hash]->getKey() != key)

hash = (hash+1)%tableSize;

if (table[hash] != NULL)

delete table[hash];

table[hash] = new HashEntry(key,value);

}

~HashMap() {

for(int i=0; i<tableSize; i++){

if (table[i] != NULL)

delete table[i];

}

}

};

vector<Point> sieveAlgorithm(vector<Point> points){

/\*

\* Go through all points

\* pick randpoint, store dist to each point

\* min distance is D

\* take D, D/3 -> size of mesh

\* divide unit square into D/3

\* loop through points, where point goes into map

\* which box it goes in, store where that goes in

\* once you have hashtable

\* with every box label points to all points in box

\* go through hash, if all boxes around it are empty and that box just has box

\*

\* Recur until subset becomes 0

\* look at D you used

\* last seive algorithm

\* build new hash table

\* go through each point, find points

\* find points in neighborhood, brute force

\* compare min distances

\*/

//std::map <int,int> pointDistDict;

int pointsLength = points.size();

if (pointsLength==3){

return (bruteForce(points));

}

if (pointsLength==2){

return points;

}

HashMap randPointHashDict = new HashMap();

double D = 2.0;

srand (time(NULL));

for (int index=0; index<pointsLength; index++)

{

double xValue = (double(rand()) / double(RAND\_MAX));

double yValue = (double(rand()) / double(RAND\_MAX));

vector<Point> randPoint = Point(xValue, yValue);

double distBet = calculateDistance(points[index],randPoint);

randPointHashDict.put(points[index],distBet);

if(distBet<D){

D = distBet;

}

}

double miniD = D/3.0;

/\*vector<Point> newLeftPointsX(pointsLength);

vector<Point> newMidPointsX(pointsLength);

vector<Point> newRightPointsX(pointsLength);

for(int index=0; index<pointsLength; index++){

oldXVal = points[index].getX();

oldYVal = points[index].getY();

if(oldXVal < miniD){

newLeftPointsX[index] = points[index];

}

else if(oldXVal > miniD && oldXVal < miniD\*2){

newMidPointsX[index] = points[index];

}

else{

newRightPointsX[index] = points[index];

}

}

vector<Point> newBottomPointY(pointsLength);

vector<Point> newMidPointsY(pointsLength);

vector<Point> newTopPointsY(pointsLength);

for(int index=0; index<pointsLength; index++){

oldXVal = points[index].getX();

oldYVal = points[index].getY();

if(oldYVal < miniD){

newBottomPointsY[index] = points[index];

}

else if(oldYVal > miniD && oldYVal < miniD\*2){

newMidPointsY[index] = points[index];

}

else{

newTopPointsY[index] = points[index];

}

}\*/

vector<Point> newTopLeftPoints(pointsLength);

vector<Point> newTopMidPoints(pointsLength);

vector<Point> newTopRightPoints(pointsLength);

vector<Point> newMidLeftPoints(pointsLength);

vector<Point> newCenterPoints(pointsLength);

vector<Point> newMidRightPoints(pointsLength);

vector<Point> newBottomLeftPoints(pointsLength);

vector<Point> newBottomMidPoints(pointsLength);

vector<Point> newBottomRightPoints(pointsLength);

for(int index=0; index<pointsLength; index++){

oldXVal = points[index].getX();

oldYVal = points[index].getY();

if(oldXVal<0.5-miniD){

if(oldYVal<0.5-miniD){

newTopLeftPoints[index] = points[index];

}

else if(oldYVal>0.5+miniD){

newBottomLeftPoints[index] = points[index];

}

else{

newMidLeftPoints[index] = points[index];

}

}

else if(oldXVal>0.5+miniD){

if(oldYVal<0.5-miniD){

newTopRightPoints[index] = points[index];

}

else if(oldYVal>0.5+miniD){

newBottomRightPoints[index] = points[index];

}

else{

newMidRightPoints[index] = points[index];

}

}

else{

if(oldYVal<0.5-miniD){

newTopMidPoints[index] = points[index];

}

else if(oldYVal>0.5+miniD){

newBottomMidPoints[index] = points[index];

}

else{

newCenterPoints[index] = points[index];

}

}

}

for(int index=0; index<pointsLength; index++){

randPointHashDict.get(points[index]);

centerLength = newCenterPoints.size();

if(newTopLeftPoints.size()==0 && newTopMidPoints.size()==0 && newTopRightPoints.size()==0 && newMidLeftPoints.size()==0 && newMidRightPoints.size()==0 && newBottomLeftPoints.size()==0 && newBottomMidtPoints.size()==0 && newBottomRightPoints.size()==0){

for(int i=0; i<centerLength; i++){

vector<Point> brutePoints(centerLength);

brutePoints[i] = newCenterPoints[i];

}

return bruteForce(brutePoints);

}

else{

return sieveAlgorithm(newCenterPoints);

}

}

}

int main(void) {

int numPoints = 3;

while (true){

vector<Point> values(numPoints);

sort (values.begin(), values.end(), sortByX);

srand (time(NULL));

for (int point = 0; point < numPoints; point++){

double xValue = (double(rand()) / double(RAND\_MAX));

double yValue = (double(rand()) / double(RAND\_MAX));

values[point] = Point(xValue, yValue);

}

vector<Point> pairBrute;

vector<Point> pairRecursive;

vector<Point> pairSieve;

cout << numPoints << "\t";

const clock\_t initialTime = clock();

pairBrute = bruteForce(values);

cout << float ( clock () - initialTime ) / CLOCKS\_PER\_SEC << "\t\t";

const clock\_t initialTime2 = clock();

pairRecursive = recursiveDivide(values);

cout << float( clock () - initialTime2 ) / CLOCKS\_PER\_SEC << "\t\t";

const clock\_t initialTime3 = clock();

pairSieve = sieveAlgorithm(values);

cout << float( clock () - initialTime3 ) / CLOCKS\_PER\_SEC << endl;

if (numPoints > 10000)

numPoints \*= 1.25;

else if (numPoints > 5000)

numPoints \*= 1.5;

else

numPoints \*= 2;

/\*

Point firstBrute = pairBrute[0];

Point secondBrute = pairBrute[1];

Point firstRecursive = pairRecursive[0];

Point secondRecursive = pairRecursive[1];

int sizeGrid = 500;

vector<Point> scaled(numPoints);

for (int index = 0; index < numPoints; index++){

Point unscaled = values[index];

scaled[index] = Point(double(int(unscaled.getX()\*sizeGrid)), double(int(unscaled.getY()\*sizeGrid)));

}

Point firstBruteScaled = Point(double(int(firstBrute.getX()\*sizeGrid)), double(int(firstBrute.getY()\*sizeGrid)));

Point secondBruteScaled = Point(double(int(secondBrute.getX()\*sizeGrid)), double(int(secondBrute.getY()\*sizeGrid)));

Point firstRecursiveScaled = Point(double(int(firstRecursive.getX()\*sizeGrid)), double(int(firstRecursive.getY()\*sizeGrid)));

Point secondRecursiveScaled = Point(double(int(secondRecursive.getX()\*sizeGrid)), double(int(secondRecursive.getY()\*sizeGrid)));

cout << "Brute:" << endl;

cout << firstBruteScaled << endl;

cout << secondBruteScaled << endl;

cout << "Recursive:" << endl;

cout << firstRecursiveScaled << endl;

cout << secondRecursiveScaled << endl;

//printToPPM(sizeGrid, numPoints, scaled, firstRecursiveScaled, secondRecursiveScaled);

\*/

}

}