CS-F402 Computational Geometry
Programming Assignment

Kotha Rohit Reddy
2020A7PS1890H

April 24, 2023

1 Line Segment Intersection

The Sweep Line Algorithm was implemented as mentioned in the course. This only handles the non degenrate cases successfully.

The test cases were generated manually using GeoGebra. I was able to test for inputs around 5 to 15 line segments.

The code runs as expected in $O((n+I)logn)$ time and using only $O(n)$ space. The timings recorded for the few inputs are given below. This was tested with the CPU AMD Ryzen 7 4800H on Windows 11.

No.	Line Segments	Intersections	Time
1	8	12	0.04s
2	5	1	0.01s
3	5	7	0.03s
4	14	18	0.11s

2 Map Overlay

The Map Overlay for the worst case is when the n segments intersect all the other n segments of other map. I have generated a case that forms a grid with n horizontal and n vertical segments. When passed the value of n as a command line argument, it outputs the runtime and solution for the overlay.

Looking at the data, we see that both time and space would be a bottleneck at bigger values of n. But we can parallelize the code which can reduce the time significantly. Hence, when the algorithm is implemented on multiple therads, time would not be a bottleneck but space will be.

The code runs in $O((n+I)logn)$ time and uses $O(n^2)$ space. The timings recorded for the inputs are given below. This was tested with the CPU AMD Ryzen 7 4800H on Windows 11.

No.	n	Time
1	2	0.001s
2	5	0.002s
3	10	0.006s
4	20	0.026s
5	50	0.130s
6	100	0.521s
7	300	5.21s

Appendix

The source code can also be viewed at github here

timer.h

```
#include <chrono>
#include <iostream>
#include <string>

class Timer {
public:
    void start(std::string name);
    void display();

private:
    std::chrono::system_clock::time_point startTime;
    std::chrono::system_clock::time_point curTime;
    std::string name;
};

void Timer::start(std::string name) {
    this->name = name;
    startTime = std::chrono::system_clock::now();
}
void Timer::display() {
    curTime = std::chrono::system_clock::now();
    std::chrono::duration<float> dur = curTime - startTime;
    std::cout << name << " Completed in " << dur.count() << "s" << std::endl;
}
```

Line Segment Intersection

geometry.h

```
#pragma once

#include <iostream>
#include <fstream>
#include <utility>
#include <cmath>

#define PREC 0.000001

extern double sweep_x;

int compare(const double& a, const double& b) {
    if (fabs(a - b) < PREC)
        return 0;
    else if (a < b)
        return -1;
    else
        return 1;
```

```cpp
}

enum class PTYPE { left, right, intersection };

struct Point {
    Point() {};
    Point(double x, double y) : x {x}, y {y} {};
    Point(double x, double y, PTYPE cat) : x {x}, y {y}, type {cat} {};

    double x, y;
    PTYPE type;
};

struct Line {
    Line() {};
    Line(Point a, Point b) : left {a}, right {b} {
        if (compare(a.x, b.x) == 0) {
            isVertical = true;
            k = a.x;
        } else {
            isVertical = false;
            m = (a.y - b.y) / (a.x - b.x);
            c = a.y - m * a.x;
        }
    };

    double evaly(double val) const { return m * val + c; };

    Point left, right;
    bool isVertical;
    double k;     // x = k if vertical
    double m, c;  // y = mx + c;
};

std::ostream& operator<<(std::ostream& os, const Point& a) {
    os << '(' << a.x << ',' << a.y << ')';
    return os;
}

std::istream& operator>>(std::istream& is, Point& a) {
    is >> a.x >> a.y;
    return is;
}

Point operator+(const Point& a, const Point& b) {
    return Point(a.x + b.x, a.y + b.y);
}

Point operator-(const Point& a, const Point& b) {
    return Point(a.x - b.x, a.y - b.y);
}

bool operator<(const Point& a, const Point& b) {
    return (compare(a.x, b.x) == -1 || (compare(a.x, b.x) == 0 && compare(a.y, b.y
    ) == -1));
```

```
}

bool operator==(const Point& a, const Point& b) {
    return compare(a.x, b.x) == 0 && compare(a.x, b.x) == 0;
}

bool operator>(const Point& a, const Point& b) {
    return b < a;
}

double cross(const Point& a, const Point& b) {
    return a.x * b.y - b.x * a.y;
}

// 1 anticlockwise, 0 collinear, -1 clockwise
int orientation(const Point& a, const Point& b, const Point& c) {
    double cr = cross(b - a, c - b);
    if (compare(cr, 0) == 1)
        return 1;
    else if (compare(cr, 0) == 0)
        return 0;
    else
        return -1;
}

std::ostream& operator<<(std::ostream& os, const Line& a) {
    os << '{' << a.left << ',' << a.right << '}';
    return os;
}

std::istream& operator>>(std::istream& is, Line& cur) {
    Point a, b;
    is >> a >> b;
    if (compare(a.x, b.x) == -1 || (compare(a.x, b.x) == 0 && compare(a.x, b.x) == -1)) {
        cur = Line(a, b);
    } else {
        cur = Line(b, a);
    }
    cur.left.type = PTYPE::left;
    cur.right.type = PTYPE::right;

    return is;
}

bool operator<(const Line& a, const Line& b) {
    return compare(a.evaly(sweep_x), b.evaly(sweep_x)) == -1 ||
           (compare(a.evaly(sweep_x), b.evaly(sweep_x)) == 0 && orientation(a.left, b.left, a.right) < 0);
}

bool operator==(const Line& a, const Line& b) {
    return a.left == b.left && a.right == b.right;
}

```

```
bool operator>(const Line& a, const Line& b) {
    return b < a;
}

// one line is part of another with different endpoints
bool checkSameLine(const Line& a, const Line& b) {
    if (a.isVertical && b.isVertical) { return compare(a.k, b.k) == 0; }
    return (compare(a.m, b.m) == 0 && compare(a.c, b.c) == 0);
}

// must be two different lines and non vertical
bool checkIntersection(const Line& a, const Line& b) {
    if (orientation(a.left, b.left, a.right) == orientation(a.left, b.right, a.right)) return false;
    if (orientation(b.left, a.left, b.right) == orientation(b.left, a.right, b.right)) return false;
    return true;
}

Point intersect(const Line& a, const Line& b) {
    double x, y;

    x = (b.c - a.c) / (a.m - b.m);
    y = (b.c * a.m - a.c * b.m) / (a.m - b.m);

    Point temp = Point(x, y, PTYPE::intersection);
    return temp;
}
```

sweep.h

```
#pragma once

#include "geometry.h"

#include <iostream>
#include <fstream>
#include <utility>
#include <map>
#include <set>
#include <vector>

extern double sweep_x;

class event_queue {
public:
    void processLines(const std::vector<Line>& arr);

    void push(std::pair<Point, Line> cur) { data.insert(cur); };
    void erase(std::pair<Point, Line> cur);
    std::pair<Point, Line> top() { return *(data.begin()); };
    void pop() { data.erase(data.begin()); };

    int size() const { return data.size(); };
    bool empty() const { return data.empty(); };
    std::multimap<Point, Line> data;
};

class sweep_status {
public:
    void push(Line cur) { data.insert(cur); };
    void erase(Line cur);

    int size() const { return data.size(); };
    bool empty() const { return data.empty(); };

    bool existPred(Line cur) const;
    Line getPred(Line cur) const;
    bool existSucc(Line cur) const;
    Line getSucc(Line cur) const;

    std::multiset<Line> data;
};

void event_queue::processLines(const std::vector<Line>& arr) {
    if (arr.size() == 0) return;
    for (int i = 0; i < arr.size(); i++) {
        this->push(std::make_pair(arr[i].left, arr[i]));
        this->push(std::make_pair(arr[i].right, arr[i]));
    }
}

void event_queue::erase(std::pair<Point, Line> cur) {
    typedef std::multimap<Point, Line>::iterator iterator;
    std::pair<iterator, iterator> iterpair = data.equal_range(cur.first);

    iterator it = iterpair.first;
    for (; it != iterpair.second;) {
        if (it->second == cur.second) {
            it = data.erase(it);
        } else
            it++;
    }
}

void sweep_status::erase(Line cur) {
    auto it = data.find(cur);
    data.erase(it);
}

bool sweep_status::existPred(Line cur) const {
    auto it = data.find(cur);
    if (it == data.begin())
        return false;
    else
        return true;
}

Line sweep_status::getPred(Line cur) const {
    auto it = data.find(cur);
    it--;
    return *it;
}

bool sweep_status::existSucc(Line cur) const {
    auto it = data.find(cur);
    it++;
    if (it == data.end())
        return false;
    else
        return true;
}

Line sweep_status::getSucc(Line cur) const {
    auto it = data.find(cur);
    it++;
    return *it;
}

std::ostream& operator<<(std::ostream& ofs, const event_queue& events) {
    ofs << events.size() << std::endl;
    for (auto i: events.data) { ofs << i.first << ' ' << i.second << std::endl; }
    return ofs;
}

std::ostream& operator<<(std::ostream& ofs, const sweep_status& sweepline) {
    ofs << sweepline.size() << std::endl;
    for (auto i: sweepline.data) { ofs << i << ' ' << i.evaly(sweep_x) << std::
    endl; }
    return ofs;
}

std::vector<Line> readInput(std::istream& instream) {
    int n;
    instream >> n;
    std::vector<Line> ans(n);
    for (int i = 0; i < n; i++) { instream >> ans[i]; }
    return ans;
}

void processEvent(Line fir, Line sec, event_queue& events) {
    // fir is below line i.e least y before intersection
    if (checkIntersection(fir, sec)) {
        Point ans = intersect(fir, sec);
        if (compare(ans.x, sweep_x) == 1) events.push(std::make_pair(ans, fir));
    }
}

void removeEvent(Line fir, Line sec, event_queue& events) {
    // fir is below line i.e least y before intersection
    if (checkIntersection(fir, sec)) {
        Point ans = intersect(fir, sec);
        if (compare(ans.x, sweep_x) == 1) { events.erase(std::make_pair(ans, fir))
    ; }
    }
}

void processLeftEvents(Line cur, const sweep_status& sweepline, event_queue& events) {
    if (sweepline.existPred(cur) && sweepline.existSucc(cur)) {
        // cur in the middle
        removeEvent(sweepline.getPred(cur), sweepline.getSucc(cur), events);
        processEvent(sweepline.getPred(cur), cur, events);
        processEvent(cur, sweepline.getSucc(cur), events);
    } else if (sweepline.existSucc(cur)) {
        // is in the bottom
        processEvent(cur, sweepline.getSucc(cur), events);
    } else if (sweepline.existPred(cur)) {
        // is in the top
        processEvent(sweepline.getPred(cur), cur, events);
    }
}

void processRightEvents(Line cur, const sweep_status& sweepline, event_queue& events) {
    if (sweepline.existPred(cur) && sweepline.existSucc(cur)) {
        // cur in the middle
        processEvent(sweepline.getPred(cur), sweepline.getSucc(cur), events);
    }
}

void processInterEvents(Line cur, const sweep_status& sweepline, event_queue& events) {
    Line below, top;
    if (sweepline.existSucc(cur)) {
        below = cur;
        top = sweepline.getSucc(cur);
    } else if (sweepline.existPred(cur)) {
        below = sweepline.getPred(cur);
        top = cur;
    }

    if (sweepline.existSucc(top)) {
        removeEvent(top, sweepline.getSucc(top), events);
        processEvent(below, sweepline.getSucc(top), events);
    }

    if (sweepline.existPred(below)) {
        removeEvent(sweepline.getPred(below), below, events);
        processEvent(sweepline.getPred(below), top, events);
    }
}
```

intersection.cpp

```
#include "geometry.h"
#include "sweep.h"
#include "timer.h"

```

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <set>

using namespace std;

double sweep_x;

int main(int argc, char** argv) {
    ifstream ifs;
    ofstream ofs;

    // final answer containing intersections
    vector<Point> ans;

    // event queue stored in a map
    event_queue events;

    // sweep status stored in stl set based on balanced tree
    sweep_status sweepline;

    // handle degenrates before passing to event queue
    vector<Line> prearr;

    if (argc == 2) {
        ifs.open(argv[1]);
        prearr = readInput(ifs);
        ifs.close();
    } else
        prearr = readInput(cin);

    Timer t1;
    t1.start("Line Sweep");
    events.processLines(prearr);

    int afterInter = 0;
    Line inter1, inter2;

    while (!events.empty()) {
        Line cur = events.top().second;
        sweep_x = events.top().first.x;

        if (afterInter == 1) {
            sweepline.push(inter1);
            sweepline.push(inter2);
            afterInter = 0;
        }
        if (events.top().first.type == PTYPE::left) {
            sweepline.push(cur);
            processLeftEvents(cur, sweepline, events);
        } else if (events.top().first.type == PTYPE::right) {
            processRightEvents(cur, sweepline, events);
            sweepline.erase(cur);
        } else if (events.top().first.type == PTYPE::intersection) {
            ans.push_back(events.top().first);
            processInterEvents(cur, sweepline, events);
            Line other = sweepline.getSucc(cur);
            afterInter = 1;
            inter1 = cur;
            inter2 = other;
            sweepline.erase(cur);
            sweepline.erase(other);
        }
        events.pop();

        ofs.open("event.debug", ios_base::out);
        ofs << "Event Queue" << endl;
        ofs << events;
        ofs.close();

        ofs.open("sweep.debug", ios_base::out);
        ofs << "Sweep Status" << endl;
        ofs << sweepline;
        ofs.close();
    }

    cout << ans.size() << endl;
    for (auto i: ans) cout << i << endl;
    t1.display();
}
```

Map Overlay

overlay.cpp

```
#include "arr_exact_construction_segments.h"
#include "arr_print.h"

#include <CGAL/basic.h>
#include <CGAL/Arr_overlay_2.h>

#include "timer.h"

#include <iostream>
#include <string>

using namespace std;

void createVerticalArrangement(Arrangement& arr1, int n) {
    for (int i = 1; i <= n; i++) { insert_non_intersecting_curve(arr1, Segment(
    Point(i, 1), Point(i, n))); }
}

void createHorizontalArrangement(Arrangement& arr1, int n) {
    for (int i = 1; i <= n; i++) { insert_non_intersecting_curve(arr1, Segment(
    Point(1, i), Point(n, i))); }
}

```

```cpp
int main(int argc, char** argv) {
    if (argc != 2) {
        cout << "value of n needed as argument" << endl;
        return -1;
    }

    int n = atoi(argv[1]);

    Arrangement arr1;
    // Construct the first arrangement, vertical
    createVerticalArrangement(arr1, n);

    // Construct the second arrangement, horizontal
    Arrangement arr2;
    createHorizontalArrangement(arr2, n);

    // Compute the overlay of the two arrangements.
    // n * (n - 1) / 2 edges
    // space increases n^2, time as nlogn
    Arrangement overlay_arr;

    Timer t1;
    t1.start("Overlay");
    CGAL::overlay(arr1, arr2, overlay_arr);
    print_arrangement_size(overlay_arr);
    t1.display();
    return 0;
}
```