Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное
учреждение высшего образования
Национальный исследовательский Нижегородский государственный
университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Отчет по лабораторной работе

«Построение выпуклой оболочки – проход Джарвиса»

Выполнил:
студент группы 381706-2
Антипин Александр Сергеевич

Проверил:
Доцент, кандидат технических наук
Сысоев Александр Владимирович

Нижний Новгород
2019

Содержание

Введение

Вычислительная геометрия занимается изучением разработки и исследования алгоритмов для решения геометрических задач. Задача построения выпуклых оболочек является одной из центральных задач вычислительной геометрии. Важность этой задачи происходит не только из-за огромного количества приложений (в распознавании образов, обработке изображений, базах данных, математической статистике), но также и из-за полезности выпуклой оболочки как инструмента решения множества задач вычислительной геометрии.

Очень широко алгоритмы построения выпуклой оболочки используются в геоинформатике и геоинформационных системах.

Одним из алгоритмов построения выпуклых оболочек в двумерном пространстве является алгоритм, предложенный Джарвисом в 1973 году. Этот алгоритм очень легок для понимания. Он также известен под названием «метод заворачивания подарка», так как этот алгоритм обрабатывает точки выпуклой оболочки одну за другой, как если бы мы оборачивали множество точек листом бумаги.

Джарвис обратил внимание на то, что многоугольник, которым является выпуклая оболочка, с одинаковым успехом можно задать упорядоченным множеством как его ребер, так и его вершин. Если задано множество точек, то довольно трудно быстро определить, является или нет некоторая точка крайней. Однако, если даны две точки, то непосредственно можно проверить, является или нет соединяющий их отрезок ребром выпуклой оболочки.

Постановка задачи

Реализовать последовательный алгоритм Джарвиса поиска минимальной выпуклой оболочки для произвольного количества точек на плоскости, также необходимо учесть все вырожденные случаи (например, когда 2 точки имеют одинаковые координаты). Затем реализовать параллельный вариант данного алгоритма используя средства MPI, провести замеры времени, найти прирост от распараллеливания алгоритма и сделать выводы, почему получились именно такие результаты.

Описание алгоритма

Для удобного представления точек, которые являются ключевыми элементами в данном алгоритме, предлагается перед написанием алгоритма Джарвиса создать простейший класс точки, полями которого будут ее координаты, а методы будут включать «геттеры» и «сеттеры» и перегрузку некоторых операций сравнения. Для сравнения точек необходимо ввести некоторый порядок, при котором одну из них можно считать больше или меньше другой. Предполагается, что если абсцисса одной точки меньше абсциссы другой, то и сама точка меньше (рис. 1). Если же абсциссы равны, то сравнение идет по ординатам, где также, если ордината одной точки меньше ординаты другой, то и сама точка будет меньше.

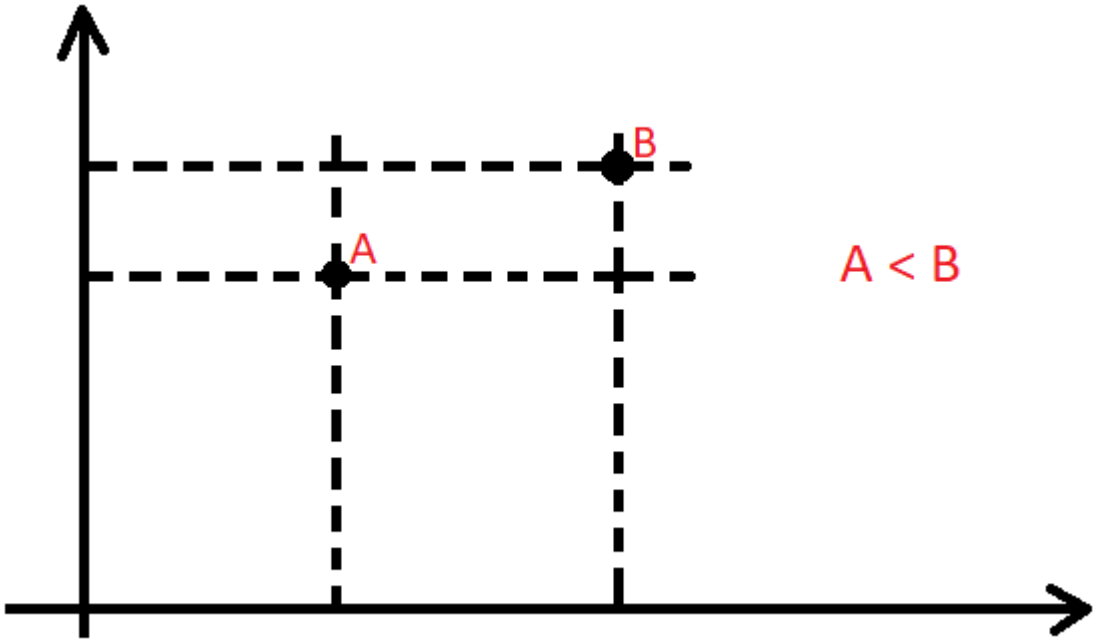

Рис. 1 (иллюстрация случая, в котором точка A < точки B)

Теперь рассмотрим алгоритм Джарвиса. Как уже было сказано ранее, его смысл очень схож с «заворачиваем подарка». Первым делом необходимо найти самую левую нижнюю точку, то есть минимальную по введенному ранее порядку. Назовем эту точку p1. Затем ищется точка p2 такая, что относительно вектора (p1, p2) все точки из заданного множества не будут лежать правее его. Затем ведущей точкой становится p2, и уже от нее ищется следующая точка выпуклой оболочки. Данная процедура продолжается, пока очередная найденная точка не окажется первой найденной точкой p1.

Как находится такая точка p_i, что относительно вектора (p_{i-1}, p_i) все оставшиеся точки из искомого множества не будут находиться справа? Заводится временная минимальная точка, которой присваивается либо 0-ая, либо 1-ая точка искомого множества (в зависимости от того, какой была точка p_{i-1}). В цикле от 0 до n (n – общее количество точек) прогоняются все точки через функцию их сравнения (если точка лежит справа от вектора (p_{i-1}, min), то она становится минимальной).

Функция сравнения точек работает следующим образом. На вход поступает предыдущая найденная точка, текущая минимальная точка и точка, которую необходимо проверить. От предыдущей точки p_{i-1} строится прямая, проходящая через проверяемую точку. Минимальная точка проецируется на построенную прямую по оси ординат. Тем самым у нас получаются две точки на одной прямой, и если спроектированная окажется меньше поступившей, то функция возвращает false, если же больше, то true (рис. 2).

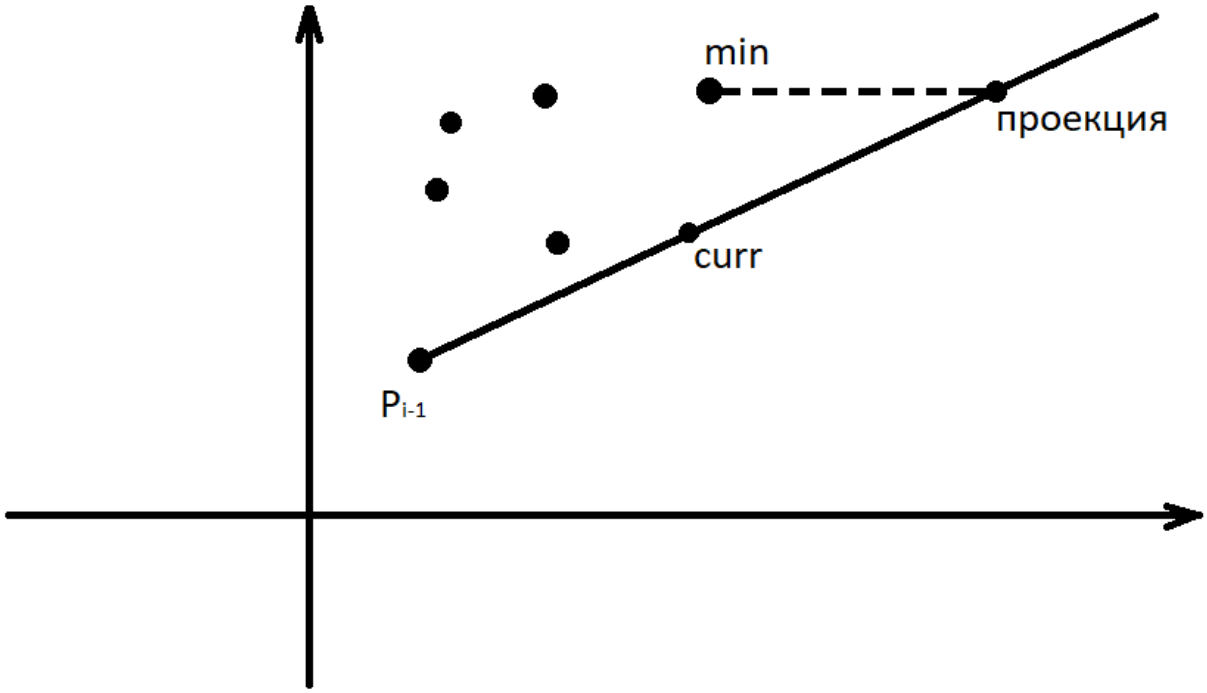

Рис. 2 (иллюстрация работы функции сравнения точек)

Схема распараллеливания

Схема распараллеливания достаточна проста и очевидна. В последовательном алгоритме для нахождения следующей точки в выпуклой оболочке нужно было пройтись по всему искомому множеству точек, что занимало n операций. В параллельной же реализации алгоритма в каждом процессе мы будем проходиться только по части искомого множества точек, ища следующую точку локально, потом найденные локальные точки передаются в нулевой процесс, где уже из них выбирается нужная точка и записывается в результирующий массив.

Приведем пример: пусть у нас имеется множество из n точек, и программа запускается на 4 процессах, тогда для каждого процесса область поиска будет следующей (рис. 3):

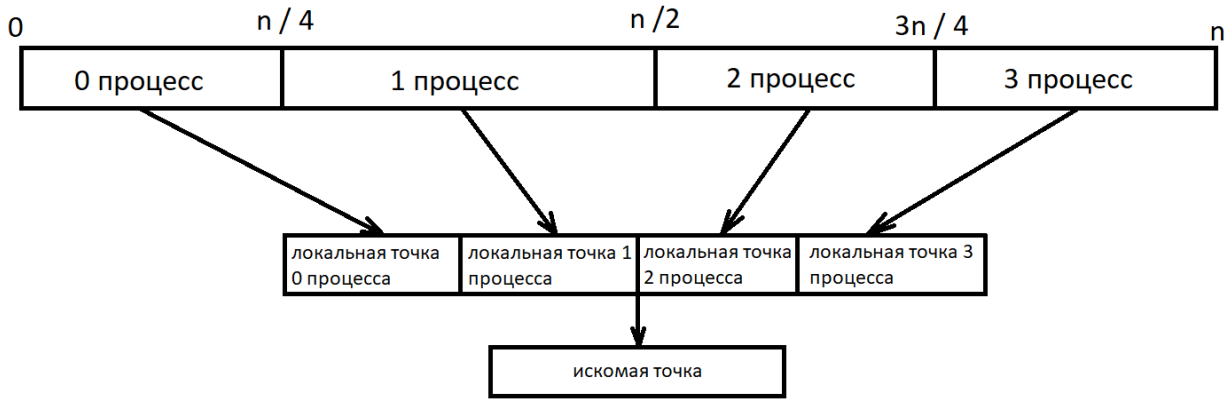

Рис. 3 (области поиска для каждого процесса)

Если количество точек в множестве не делится нацело на число процессов, то остаток от деления отправляется в последний процесс.

Описание программной реализации

Описание класса точка

Поля:

double x	координата по оси абсцисс
double y	координата по оси ординат

Методы:

point()	конструктор по умолчанию
point(const point& p)	конструктор копирования
point(const double x, const double y)	конструктор инициализации
~point()	деструктор
void setX(const double x)	метод изменения координаты X
void setY(const double y)	метод изменения координаты Y
double getX() const	метод получения координаты X
double getY() const	метод получения координаты Y
bool operator==(const point& p) const	перегрузка оператора сравнения
bool operator<=(const point& p) const	перегрузка оператора меньше равно
bool operator>(const point& p) const	перегрузка оператора больше

Описание алгоритма Джарвиса

void getRandomFieldOfPoints(std::vector <point>* field, const int maxX, const int minX, const int minY, const int maxY)	метод, который позволяет задать случайные точки в промежутках minX <= X < maxX и minY <= Y < maxY
bool isMaxRightPoint(const std::vector <point>& field, const int min, const int start, const int end)	метод, который позволяет определить, лежит ли точка правее текущей или нет
void getSequentialSolution(const std::vector<point>& field, std::vector<int>* result)	последовательный вариант алгоритма Джарвиса
void getParallelSolution(const std::vector <point>& field, std::vector<int>* result)	параллельный вариант алгоритма Джарвиса

Подтверждение корректности

Для подтверждения корректности работы данного алгоритма были написаны unit тесты. Так как основная сложность в алгоритме определение, является ли точка крайне справа по отношению к остальным, то упор в тестировании был сделан на рассмотрение всех возможных случаев взаимного расположения точек в пространстве. Для такой цели в большинстве тестов рассматривался последовательный метод.

Список тестов

- TEST(jarvis_algorithm, can_generate_random_field_of_points) – тест, который проверяет случайное задание точек в пространстве;
- TEST(jarvis_algorithm, can_get_convex_hill_by_sequential_method_0) – тест, который проверяет, сможет ли последовательный метод найти выпуклую оболочку в простом случае и не вызвать исключение;
- TEST(jarvis_algorithm, can_get_convex_hill_by_sequential_method_0_correct) – тест, который проверяет корректность списка точек, который вернул последовательный метод;
- TEST(jarvis_algorithm, can_get_convex_hull_by_sequential_method_1) – вариант расположения точек, когда есть такие, что они расположены на одной абсциссе и ординате;
- TEST(jarvis_algorithm, can_get_convex_hull_by_sequential_method_2) – несколько точек расположены на одной прямой;
- TEST(jarvis_algorithm, can_get_convex_hull_by_sequential_method_3) – вариант, когда есть 2 точки с одинаковыми координатами;
- TEST(jarvis_algorithm, can_get_convex_hull_by_sequential_method_4) – случайное задание точек и применение последовательного метода;
- TEST(jarvis_algorithm, can_get_convex_hull_by_parallel_method_0) – тестирование параллельного метода на простом примере, чтобы метод не вызвал исключение;
- TEST(jarvis_algorithm, can_get_convex_hull_by_parallel_method_0_correct) – проверка работы параллельного метода на примере с известными ответами;
- TEST(jarvis_algorithm, can_get_convex_hull_by_parallel_method_1) – проверка параллельного метода на случайных точках.
- TEST(jarvis_algorithm, can_get_convex_hull_by_parallel_method_1_correct) – сравнение ответов последовательного и параллельного методов.

Результаты экспериментов

Был проведен ряд экспериментов для проверки времени работы как последовательного, так и параллельного алгоритмов. Так как сложность алгоритма O(n*m), где n – общее количество точек в исходном множестве точек, а m – количество точек в получившейся выпуклой оболочке, то придется учитывать не только время, но это количество точек. Результаты экспериментов приведены в таблицах:

100000 точек в исходном множестве			
№ эксперимента:	Время последовательного метода (сек):	Время параллельного метода (сек):	Количество точек в выпуклой оболочке:
1	0.801432	0.345024	9
2	0.892377	0.384847	10
3	0.985963	0.41809	11
4	1.24488	0.547762	14
5	1.09639	0.456474	12

1000000 точек в исходном множестве			
№ эксперимента:	Время последовательного метода (сек):	Время параллельного метода (сек):	Количество точек в выпуклой оболочке:
1	4.6866	1.9763	5
2	5.4961	2.3186	6
3	3.65245	1.63066	4
4	6.34819	2.6539	7
5	7.28673	3.0303	8

10000000 точек в исходном множестве			
№ эксперимента:	Время последовательного метода (сек):	Время параллельного метода (сек):	Количество точек в выпуклой оболочке:
1	108.805	47.0601	12
2	86.4324	35.632	10
3	114.049	45.9312	13
4	87.0294	35.3638	10
5	104.335	42.1622	12

В экспериментах на сто тысяч и миллион элементов генерация случайных точек была в пределе от 5 до 1000 по обоим осям, в эксперименте на 10 миллионов от 5 до 10000. Все тесты параллельного метода проводились на 4 процессах.

Как видно из экспериментов у нас есть стабильное уменьшение времени работы программы в более 2-2.5 раза, что доказывает эффективность параллельного алгоритма. Такой результат достигается за счет достаточно простого и равномерного распределения данных между процессами в начале работы параллельного алгоритма и пересылками небольших порций данных во время работы алгоритма.

Тем не менее нужно сказать, что максимальный прирост производительности будет достигнут только в случае использования всех ядер процессора (и всех потоков процессора, если архитектура процессора поддерживает multi-threading). Если число процессов, на которых запускается программа, меньше чем максимальное число ядер (потоков) процессора, то соответственно процессор будет недогружен и время выполнения программы дольше. Если же число процессов, на которых запускается программа, будет превышать число ядер (потоков) процессора, то будут производиться дополнительные переключения контекста одного процесса на другой на одном ядре, а также хранение самого контекста потока, что влечет использование дополнительной памяти. Эти причины можно описать как дополнительные накладные расходы. И при слишком большом количестве процессов, на котором решили запустить программу, скорость выполнения может стать ниже чем при запуске последовательного метода.

Тестирование проводилось на компьютере с характеристиками:

- CPU – Intel Core i5 7200U (~3.1 GHz, 2 cores, 4 flows);
- RAM – 12 GB DDR4 (~2133 MHZ);
- OS – Windows 10 Home.

Заключение

В заключении можно сказать, что все поставленные цели данной работы были выполнены, а именно разработаны последовательный и параллельный методы нахождения выпуклой оболочки методом Джарвиса, написаны Unit тесты для подтверждения корректности работы программы, выполнены замеры времени, и сделаны выводы о приросте производительности.

Что же касается самого алгоритм Джарвиса, то он является достаточно простым в понимании и реализации. Его сложность O(n*m), где n – общее количество точек в исходном множестве точек, а m – количество точек в получившейся выпуклой оболочке, что в общем случае дает хороший результат по времени выполнения, но надо помнить, что в плохом случае эта сложность может стать $O(n^2)$.

Литература

- Хабр, Построение минимальных выпуклых оболочек: сайт [электронный ресурс] – URL: https://habr.com/ru/post/144921/ (дата обращения: 04.12.2019);
- Университет ИТМО, Статические выпуклые оболочки: Джарвис, Грэхем, Эндрю, Чен, QuickHull: сайт [электронный ресурс] – URL: https://neerc.ifmo.ru/wiki/index.php?title=Статические_выпуклые_оболочки:_Джарвис,_Грэхем,_Эндрю,_Чен,_QuickHull (дата обращения: 04.12.2019);
- YouTube, Алгоритмы и структуры данных (продвинутый поток). Семинар 5. Выпуклые оболочки: сайт [электронный ресурс] – URL: https://www.youtube.com/watch?v=JPypmkh77S4 (дата обращения: 04.12.2019)

Приложение

jarvis_algorithm.h

```
// Copyright 2019 Antipin Alexander
#ifndef
MODULES_TASK_3_ANTIPIN_A_JARVIS_ALGORITHM_JARVIS_ALGORITHM_H_
#define
MODULES_TASK_3_ANTIPIN_A_JARVIS_ALGORITHM_JARVIS_ALGORITHM_H_

#include <mpi.h>
#include <time.h>
#include <iostream>
#include <limits>
#include <vector>
#include <random>

class point {
    double x;
    double y;
 public:
    point();
    point(const point& p);
    point(const double x, const double y);
    ~point();
    void setX(const double x);
    void setY(const double y);
    double getX() const;
    double getY() const;
    bool operator==(const point& p) const;
    bool operator<=(const point& p) const;
    bool operator>(const point& p) const;
};

void getRandomFieldOfPoints(std::vector<point>* field, const int
maxX, const int minX, const int minY, const int maxY);
bool isMaxRightPoint(const std::vector<point>& field, const int
min, const int start, const int end);
void getSequentialSolution(const std::vector<point>& field,
std::vector<int>* result);
void getParallelSolution(const std::vector<point>& field,
std::vector<int>* result);

#endif  // MOD-
ULES_TASK_3_ANTIPIN_A_JARVIS_ALGORITHM_JARVIS_ALGORITHM_H_
```

jarvis_algorithm.cpp

```
// Copyright 2019 Antipin Alexander
```

```
#include
"../../../modules/task_3/antipin_a_jarvis_algorithm/jarvis_algorit
hm.h"
#include <vector>
#include <limits>

point::point() {
    x = 0.0;
    y = 0.0;
}

point::point(const point & p) {
    x = p.x;
    y = p.y;
}

point::point(const double x, const double y) {
    this->x = x;
    this->y = y;
}

point::~point() {
}

void point::setX(const double x) {
    this->x = x;
}

void point::setY(const double y) {
    this->y = y;
}

double point::getX() const {
    return x;
}

double point::getY() const {
    return y;
}

bool point::operator==(const point & p) const {
    if (x == p.x && y == p.y) {
        return true;
    } else {
        return false;
    }
}

bool point::operator<=(const point & p) const {
    if (x < p.x) {
        return true;
    } else if (x == p.x && y <= p.y) {
        return true;
```

```
    } else {
        return false;
    }
}

bool point::operator>(const point & p) const {
    if (x > p.x) {
        return true;
    } else if (x == p.x && y > p.y) {
        return true;
    } else {
        return false;
    }
}

void getRandomFieldOfPoints(std::vector<point>* field, const int
maxX, const int minX,
    const int maxY, const int minY) {
    std::mt19937 gen;
    gen.seed(static_cast<unsigned int>(time(NULL)));
    for (unsigned int i = 0; i < field->size(); ++i) {
        (*field)[i].setX(static_cast<double>(gen() % (maxX - minX)
+ minX));
        (*field)[i].setY(static_cast<double>(gen() % (maxY - minY)
+ minY));
    }
}

bool isMaxRightPoint(const std::vector<point>& field, const int
min, const int start, const int end) {
    if (field[start] == field[min]) {
        return true;
    }
    double k = (field[end].getY() - field[start].getY()) /
(field[end].getX() - field[start].getX());
    double k2 = (field[min].getY() - field[start].getY()) /
(field[min].getX() - field[start].getX());
    double b = field[end].getY() - k * field[end].getX();
    bool isUp = field[start].getY() < field[end].getY();
    double newX;

    // if the direction is vertical

    if (k == std::numeric_limits<double>::infinity() || k == -
std::numeric_limits<double>::infinity()) {
        newX = field[end].getX();
        if (field[end].getX() == field[min].getX()) {
            if (k == std::numeric_limits<double>::infinity()) {
                return field[end].getY() > field[min].getY() ?
true : false;
            } else if (k == -
std::numeric_limits<double>::infinity()) {
```

```
                return field[end].getY() < field[min].getY() ?
true : false;
            }
        }
    } else if (k == 0.0) {  // if the direction is horizontal
        newX = field[min].getX();
        if (field[start].getX() > field[end].getX()) {
            if (field[min].getY() != field[end].getY()) {
                return point(newX, field[end].getY()) <=
field[min] ? false : true;
            } else {
                return field[min] > field[end] ? true : false;
            }
        } else if (field[start].getX() < field[end].getX()) {
            if (field[min].getY() != field[end].getY()) {
                return point(newX, field[end].getY()) <=
field[min] ? true : false;
            } else {
                return field[min] > field[end] ? false : true;
            }
        }
    } else if (k == k2) {  // if 2 points on one line
        if (isUp) {
            return field[end].getY() > field[min].getY() ? true :
false;
        } else {
            return field[end].getY() < field[min].getY() ? true :
false;
        }
    } else {  // simple happend
        newX = (field[min].getY() - b) / k;
    }
    if (isUp && k > 0.0) {
        return point(newX, field[min].getY()) <= field[min] ?
false : true;
    } else if (isUp && k < 0.0) {
        return point(newX, field[min].getY()) > field[min] ? true
: false;
    } else if (!isUp && k > 0.0) {
        return point(newX, field[min].getY()) <= field[min] ? true
: false;
    } else if (!isUp && k < 0.0) {
        return point(newX, field[min].getY()) <= field[min] ? true
: false;
    }
    return false;
}

void getSequentialSolution(const std::vector<point>& field,
std::vector<int>* result) {
    if (field.size() < 3) {
        throw 1;
    }
```

```
    int startPoint = 0;
    int currPoint = 0;
    for (unsigned int i = 1; i < field.size(); ++i) {
        if (field[i] <= field[startPoint]) {
            startPoint = i;
        }
    }
    currPoint = startPoint;
    int j = 1;
    (*result).resize(field.size());
    (*result)[0] = currPoint;
    while (true) {
        int currMin = currPoint == 0 ? 1 : 0;
        for (unsigned int i = 0; i < field.size(); ++i) {
            if (i == static_cast<unsigned int>(currPoint)) {
                continue;
            }
            if (field[i] == field[currPoint]) {
                continue;
            }
            if (isMaxRightPoint(field, currMin, currPoint, i)) {
                currMin = i;
            }
        }
        (*result)[j] = currMin;
        currPoint = currMin;
        j++;
        if (currPoint == startPoint || field[currPoint] ==
field[startPoint]) {
            break;
        }
    }
    (*result).resize(j - 1);
}

void getParallelSolution(const std::vector<point>& field,
std::vector<int>* result) {
    if (field.size() < 3) {
        throw 1;
    }
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (field.size() < static_cast<unsigned int>(size)) {
        if (rank == 0) {
            getSequentialSolution(field, result);
            return;
        } else {
            return;
        }
    }
    int startPoint = 0;
    int currPoint = 0;
```

```
    if (rank == 0) {
        for (unsigned int i = 1; i < field.size(); ++i) {
            if (field[i] <= field[startPoint]) {
                startPoint = i;
            }
        }
    }
    MPI_Bcast(&startPoint, 1, MPI_INT, 0, MPI_COMM_WORLD);
    currPoint = startPoint;
    int j = 1;
    (*result).resize(field.size());
    int pointCount = rank == size - 1 ? field.size() / size +
field.size() % size : field.size() / size;
    (*result)[0] = currPoint;
    while (true) {
        int currMin = static_cast<unsigned int>(currPoint) ==
(field.size() / size) * static_cast<unsigned int>(rank) ?
            (field.size() / size) * rank + 1 : (field.size() /
size) * rank;
        for (unsigned int i = currMin; i < (rank == (size - 1) ?
field.size() : pointCount * (rank + 1)); ++i) {
            if (i == static_cast<unsigned int>(currPoint)) {
                continue;
            }
            if (field[i] == field[currPoint]) {
                continue;
            }
            if (isMaxRightPoint(field, currMin, currPoint, i)) {
                currMin = i;
            }
        }
        if (rank == 0) {
            for (int i = 1; i < size; ++i) {
                int locPoint;
                MPI_Status status;
                MPI_Recv(&locPoint, 1, MPI_INT, i, 0,
MPI_COMM_WORLD, &status);
                if (isMaxRightPoint(field, currMin, currPoint,
locPoint)) {
                    currMin = locPoint;
                }
            }
            (*result)[j] = currMin;
            currPoint = currMin;
            j++;
        } else {
            MPI_Send(&currMin, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
        MPI_Bcast(&currPoint, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (currPoint == startPoint || field[currPoint] ==
field[startPoint]) {
            break;
        }
```

```
    }
    (*result).resize(j - 1);
}
```

main.cpp

```
// Copyright 2019 Antipin Alexander
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <iostream>
#include <vector>
#include "./jarvis_algorithm.h"

TEST(jarvis_algorithm, can_generate_random_field_of_points) {
    std::vector<point> field(10);
    ASSERT_NO_THROW(getRandomFieldOfPoints(&field, 25, 5, 25, 5));
}

TEST(jarvis_algorithm, can_get_convex_hull_by_sequential_method_0)
{
    std::vector<point> field(10);
    field[0].setX(15); field[0].setY(6);
    field[1].setX(20); field[1].setY(19);
    field[2].setX(9); field[2].setY(20);
    field[3].setX(9); field[3].setY(23);
    field[4].setX(17); field[4].setY(15);
    field[5].setX(21); field[5].setY(5);
    field[6].setX(15); field[6].setY(24);
    field[7].setX(18); field[7].setY(23);
    field[8].setX(23); field[8].setY(13);
    field[9].setX(13); field[9].setY(9);
    std::vector<int> res;
    ASSERT_NO_THROW(getSequentialSolution(field, &res));
}

TEST(jarvis_algorithm,
can_get_convex_hull_by_sequential_method_0_correct) {
    std::vector<point> field(10);
    field[0].setX(14); field[0].setY(14);
    field[1].setX(19); field[1].setY(19);
    field[2].setX(12); field[2].setY(14);
    field[3].setX(10); field[3].setY(15);
    field[4].setX(6); field[4].setY(20);
    field[5].setX(20); field[5].setY(16);
    field[6].setX(8); field[6].setY(11);
    field[7].setX(17); field[7].setY(6);
    field[8].setX(19); field[8].setY(22);
    field[9].setX(15); field[9].setY(13);
    std::vector<int> res;
    getSequentialSolution(field, &res);
    EXPECT_EQ(res[0], 4);
    EXPECT_EQ(res[1], 6);
    EXPECT_EQ(res[2], 7);
```

```
    EXPECT_EQ(res[3], 5);
    EXPECT_EQ(res[4], 8);
}

TEST(jarvis_algorithm, can_get_convex_hull_by_sequential_method_1)
{
    std::vector<point> field(10);
    field[0].setX(22); field[0].setY(9);
    field[1].setX(7); field[1].setY(19);
    field[2].setX(10); field[2].setY(18);
    field[3].setX(20); field[3].setY(23);
    field[4].setX(20); field[4].setY(18);
    field[5].setX(13); field[5].setY(5);
    field[6].setX(23); field[6].setY(10);
    field[7].setX(23); field[7].setY(22);
    field[8].setX(7); field[8].setY(21);
    field[9].setX(23); field[9].setY(23);
    std::vector<int> res;
    ASSERT_NO_THROW(getSequentialSolution(field, &res));
}

TEST(jarvis_algorithm, can_get_convex_hull_by_sequential_method_2)
{
    std::vector<point> field(10);
    field[0].setX(9); field[0].setY(8);
    field[1].setX(18); field[1].setY(13);
    field[2].setX(19); field[2].setY(11);
    field[3].setX(23); field[3].setY(9);
    field[6].setX(18); field[6].setY(24);
    field[5].setX(9); field[5].setY(15);
    field[4].setX(19); field[4].setY(22);
    field[7].setX(19); field[7].setY(15);
    field[8].setX(23); field[8].setY(14);
    field[9].setX(24); field[9].setY(22);
    std::vector<int> res;
    ASSERT_NO_THROW(getSequentialSolution(field, &res));
}

TEST(jarvis_algorithm, can_get_convex_hull_by_sequential_method_3)
{
    std::vector<point> field(10);
    field[0].setX(14); field[0].setY(9);
    field[1].setX(5); field[1].setY(6);
    field[2].setX(16); field[2].setY(12);
    field[3].setX(5); field[3].setY(22);
    field[4].setX(10); field[4].setY(10);
    field[5].setX(19); field[5].setY(18);
    field[6].setX(5); field[6].setY(17);
    field[7].setX(20); field[7].setY(15);
    field[8].setX(5); field[8].setY(6);
    field[9].setX(20); field[9].setY(6);
    std::vector<int> res;
    ASSERT_NO_THROW(getSequentialSolution(field, &res));
```

```
}

TEST(jarvis_algorithm, can_get_convex_hull_by_sequential_method_4)
{
    std::vector<point> field(10);
    getRandomFieldOfPoints(&field, 25, 5, 25, 5);
    std::vector<int> res;
    ASSERT_NO_THROW(getSequentialSolution(field, &res));
}

TEST(jarvis_algorithm, can_get_convex_hull_by_parallel_method_0) {
    std::vector<point> field(10);
    field[0].setX(14); field[0].setY(14);
    field[1].setX(19); field[1].setY(19);
    field[2].setX(12); field[2].setY(14);
    field[3].setX(10); field[3].setY(15);
    field[4].setX(6); field[4].setY(20);
    field[5].setX(20); field[5].setY(16);
    field[6].setX(8); field[6].setY(11);
    field[7].setX(17); field[7].setY(6);
    field[8].setX(19); field[8].setY(22);
    field[9].setX(15); field[9].setY(13);
    std::vector<int> res;
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ASSERT_NO_THROW(getParallelSolution(field, &res));
}

TEST(jarvis_algorithm,
can_get_convex_hull_by_parallel_method_0_correct) {
    std::vector<point> field(10);
    field[0].setX(14); field[0].setY(14);
    field[1].setX(19); field[1].setY(19);
    field[2].setX(12); field[2].setY(14);
    field[3].setX(10); field[3].setY(15);
    field[4].setX(6); field[4].setY(20);
    field[5].setX(20); field[5].setY(16);
    field[6].setX(8); field[6].setY(11);
    field[7].setX(17); field[7].setY(6);
    field[8].setX(19); field[8].setY(22);
    field[9].setX(15); field[9].setY(13);
    std::vector<int> res;
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    getParallelSolution(field, &res);
    if (rank == 0) {
        EXPECT_EQ(res[0], 4);
        EXPECT_EQ(res[1], 6);
        EXPECT_EQ(res[2], 7);
        EXPECT_EQ(res[3], 5);
        EXPECT_EQ(res[4], 8);
    }
}
```

```
TEST(jarvis_algorithm, can_get_convex_hull_by_parallel_method_1) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<point> field(10);
    double* arrX = new double[10];
    double* arrY = new double[10];
    if (rank == 0) {
        getRandomFieldOfPoints(&field, 25, 5, 25, 5);
        for (int i = 0; i < 10; ++i) {
            arrX[i] = field[i].getX();
            arrY[i] = field[i].getY();
        }
    }
    MPI_Bcast(arrX, 10, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(arrY, 10, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    std::vector<int> res;
    if (rank != 0) {
        for (int i = 0; i < 10; ++i) {
            field[i].setX(arrX[i]);
            field[i].setY(arrY[i]);
        }
    }
    ASSERT_NO_THROW(getParallelSolution(field, &res));
}

TEST(jarvis_algorithm,
can_get_convex_hull_by_parallel_method_1_correct) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<point> field(10);
    double* arrX = new double[10];
    double* arrY = new double[10];
    if (rank == 0) {
        getRandomFieldOfPoints(&field, 25, 5, 25, 5);
        for (int i = 0; i < 10; ++i) {
            arrX[i] = field[i].getX();
            arrY[i] = field[i].getY();
        }
    }
    MPI_Bcast(arrX, 10, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(arrY, 10, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    std::vector<int> res;
    if (rank != 0) {
        for (int i = 0; i < 10; ++i) {
            field[i].setX(arrX[i]);
            field[i].setY(arrY[i]);
        }
    }
    getParallelSolution(field, &res);
    if (rank == 0) {
        std::vector<int> res2;
        getSequentialSolution(field, &res2);
```

```
        for (unsigned int i = 0; i < res.size(); ++i) {
            EXPECT_EQ(res[i], res2[i]);
        }
    }
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    srand(time(NULL));
    MPI_Init(&argc, &argv);

    ::testing::AddGlobalTestEnvironment(new GTestMPIListen-
er::MPIEnvironment);
    ::testing::TestEventListeners& listeners =
        ::testing::UnitTest::GetInstance()->listeners();

    listeners.Release(listeners.default_result_printer());
    listeners.Release(listeners.default_xml_generator());

    listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);
    return RUN_ALL_TESTS();
}
```